

Харви Дейтел, Пол Дейтел
КАК ПРОГРАММИРОВАТЬ НА С++

Книга предлагает полный курс изучения наиболее популярного и перспективного языка программирования — С++ и рассчитана как на начинающих, не владеющих никаким языком программирования, так и на опытных пользователей. Для начинающих — это полноценный курс, в котором изучается все, начиная с устройства компьютера и кончая новейшими достижениями С++: шаблонами функций и классов, обработкой исключений и т.п. Опытный программист может пропустить часть известного ему материала, но получит множество полезных сведений в остальных частях книги. Особое внимание авторы уделяют современным методикам разработки программного обеспечения: наследованию, полиморфизму, объектно-ориентированному проектированию и объектно-ориентированному программированию, не забывая и о классическом структурном программировании. Приведено множество полезных советов.

Книга рассчитана на широкий круг читателей, от начинающих осваивать азы программирования до опытных разработчиков.

Содержание

Предисловие	15
Введение в объектное ориентирование начинается с главы 1!	16
Об этой книге	16
Обзор книги	21
Глава 1. Введение в компьютеры и программирование на С++	31
1.1. Введение	32
1.2. Что такое компьютер?	35
1.3. Организация компьютера	36
1.4. Эволюция операционных систем	37
1.5. Персональные вычисления, распределенные вычисления и вычисления на платформе клиент/сервер	38
1.6. Машинные языки, языки ассемблера и языки высокого уровня	38
1.7. История С++	40
1.8. Библиотеки классов С++ и стандартная библиотека С	41
1.9. Параллельный С++	42
1.10. Другие языки высокого уровня	43
1.11. Структурное программирование	43
1.12. Общее описание типичной среды программирования на С++	44
1.13. Общие замечания о С++ и об этой книге	47
1.14. Введение в программирование на С++	48
1.15. Простая программа: печать строки текста	48
1.16. Другая простая программа: сложение двух целых чисел	52
1.17. Концепции памяти	56
1.18. Арифметика	57

1.19. Принятие решений: операции проверки на равенство и отношения	61
1.20. Размышления об объектах	65
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 2. Управляющие структуры	87
2.1. Введение	89
2.2. Алгоритмы	89
2.3. Псевдокод	89
2.4. Управляющие структуры	90
2.5. Структура выбора if (ЕСЛИ)	93
2.6. Структура выбора if/else (ЕСЛИ-ИНАЧЕ)	94
2.7. Структура повторения while (ПОКА)	99
2.8. Разработка алгоритмов: учебный пример 1 (повторение, управляемое счетчиком)	100
2.9. Нисходящая разработка алгоритмов с пошаговой детализацией: учебный пример 2 (повторение, управляемое меткой)	102
2.10. Нисходящая разработка алгоритмов с пошаговой детализацией: учебный пример 3 (вложенные управляющие структуры)	109
2.11. Операции присваивания	113
2.12. Операции инкремента и декремента	114
2.13. Основы повторения, управляемого счетчиком	116
2.14. Структура повторения for (ЦИКЛ)	119
2.15. Пример использования структуры for	122
2.16. Структура множественного выбора switch	127
2.17. Структура повторения do/while	133
2.18. Операторы break и continue	135
2.19. Логические операции	137
2.20. Ошибки случайной подмены операций проверки равенства (==) и присваивания (=)	139
2.21. Заключение по структурному программированию	141
2.22. Размышления об объектах: идентификация объектов задачи	147
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 3. Функции	177
3.1. Введение	178
3.2. Программные модули в C++	179
3.3. Математические библиотечные функции	180

3.4. Функции	181
3.5. Определения функций	182
3.6. Прототипы функций	187
3.7. Заголовочные файлы	189
3.8. Генерация случайных чисел	189
3.9. Пример: азартная игра	195
3.10. Классы памяти	198
3.11. Правила, определяющие область действия	201
3.12. Рекурсия	205
3.13. Пример использования рекурсии: последовательность чисел Фибоначчи	208
3.14. Рекурсии или итерации	211
3.15. Функции с пустыми списками параметров	213
3.16. Встраиваемые функции	214
3.17. Ссылки и ссылочные параметры	216
3.18. Аргументы по умолчанию	219
3.19. Унарная операция разрешения области действия	221
3.20. Перегрузка функций	222
3.21. Шаблоны функции	224
3.22. Размышления об объектах: идентификация атрибутов объектов	226
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 4. Массивы	259
4.1. Введение	260
4.2. Массивы	261
4.3. Объявление массивов	263
4.4. Примеры использования массивов	263
4.5. Передача массивов в функции	276
4.6. Сортировка массивов	280
4.7. Учебный пример: вычисление среднего значения, медианы и моды с использованием массивов	282
4.8. Поиск в массивах: линейный поиск и двоичный поиск	285
4.9. Многомерные массивы	290
4.10. Размышления об объектах: идентификация поведений объектов	297
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения • Упражнения на рекурсию	
Глава 5. Указатели и строки	321

5.1. Введение	322
5.2. Объявления и инициализация переменных указателей	323
5.3. Операции над указателями	324
5.4. Вызов функций по ссылке	326
5.5. Использование спецификатора const с указателями	330
5.6. Пузырьковая сортировка, использующая вызов по ссылке	336
5.7. Выражения и арифметические действия с указателями	340
5.8. Взаимосвязи между указателями и массивами	344
5.9. Массивы указателей	348
5.10. Учебный пример: моделирование тасования и раздачи карт	349
5.11. Указатели на функции	354
5.12. Введение в обработку символов и строк	358
5.13. Размышления об объектах: взаимодействие объектов	367
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения • Специальный раздел: построение вашего собственного компьютера Дополнительные упражнения на указатели • Упражнения на работу со строками • Специальный раздел: упражнения повышенной сложности на работу со строками • Головоломный проект работы со строками	
Глава 6. Классы и абстрагирование данных	405
6.1. Введение	406
6.2. Определения структур	408
6.3. Доступ к элементам структуры	408
6.4. Использование определенного пользователем типа Time с помощью Struct	409
6.5. Использование абстрактного типа данных Time с помощью класса	411
6.6. Область действия класс и доступ к элементам класса	418
6.7. Отделение интерфейса от реализации	419
6.8. Управление доступом к элементам	423
6.9. Функции доступа и обслуживающие функции-утилиты	426
6.10. Инициализация объектов класса: конструкторы	429
6.11. Использование конструкторов с аргументами по умолчанию	429
6.12. Использование деструкторов	433
6.13. Когда вызываются конструкторы и деструкторы	433
6.14. Использование данных-элементов и функций-элементов	436
6.15. Тонкий момент: возвращение ссылки на закрытые данные-элементы	441
6.16. Присваивание побитовым копированием по умолчанию	443
6.17. Повторное использование программного обеспечения	445
6.18. Размышления об объектах: программирование классов для моделирования лифта	445

Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 7. Классы: часть II	459
7.1. Введение	460
7.2. Константные объекты и функции-элементы	461
7.3. Композиция: классы как элементы других классов	467
7.4. Дружественные функции и дружественные классы	471
7.5. Использование указателя this	474
7.6. Динамическое распределение памяти с помощью операций new и delete	479
7.7. Статические элементы класса	480
7.8. Абстракция данных и скрытие информации	485
7.9. Классы контейнеры и итераторы	489
7.10. Размышления об объектах: использование композиции и динамического управления объектом в модели лифта	489
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 8. Перегрузка операций	497
8.1. Введение	498
8.2. Основы перегрузки операций	499
8.3. Ограничения на перегрузку операции	501
8.4. Функции-операции как элементы класса и как дружественные функции	502
8.5. Перегрузка операций поместить в поток и взять из потока	504
8.6. Перегрузка унарных операций	506
8.7. Перегрузка бинарных операций	507
8.8. Учебный пример: класс массив	508
8.9. Преобразования типов	519
8.10. Учебный пример: класс строка	520
8.11. Перегрузка ++ и --	531
8.12. Учебный пример: класс дата	532
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы, по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 9. Наследование	551
9.1. Введение	552
9.2. Базовые классы и производные классы	554
9.3. Защищенные элементы	556
9.4. Приведение типов указателей базовых классов к указателям производных классов	556

9.5. Использование функций-элементов	562
9.6. Переопределение элементов базового класса в производном классе	562
9.7. Открытые, защищенные и закрытые базовые классы	566
9.8. Прямые и косвенные базовые классы	568
9.9. Использование конструкторов и деструкторов в производных классах	568
9.10. Неявное преобразование объектов производных классов в объекты базовых классов	571
9.11. Проектирование программного обеспечения с помощью наследования	573
9.12. Композиция и наследование	574
9.13. Отношения «использует А» и «знает А»	575
9.14. Учебный пример: точка, круг, цилиндр	575
9.15. Множественное наследование	580
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 10. Виртуальные функции и полиморфизм	595
10.1. Введение	596
10.2. Поля типов и операторы switch	597
10.3. Виртуальные функции	597
10.4. Абстрактные базовые классы и конкретные классы	599
10.5. Полиморфизм	600
10.6. Учебный пример: система расчета заработной платы	602
10.7. Новые классы и динамическое связывание	612
10.8. Виртуальные деструкторы	613
10.9. Учебный пример: интерфейс наследования и его реализация	614
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 11. Потоки ввода-вывода в C++	629
11.1. Введение	631
11.2. Потоки	632
11.3. Вывод потоков	635
11.4. Ввод потоков	639
11.5. Неформатированный ввод-вывод с использованием read, gcount и write	645
11.6. Манипуляторы потоков	645
11.7. Состояния формата потоков	650
11.8. Состояния ошибок потока	660
11.9. Ввод-вывод определенных пользователем типов данных	661
11.10. Связывание выходного потока с входным	664
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности •	

Замечания по технике программирования • Упражнения для самопроверки •	
Ответы на упражнения для самопроверки • Упражнения	
Глава 12. Шаблоны	683
12.1. Введение	684
12.2. Шаблоны функций	685
12.3. Перегрузка шаблонных функций	688
12.4. Шаблоны классов	689
12.5. Шаблоны классов и нетиповые параметры	694
12.6. Шаблоны и наследование	695
12.7. Шаблоны и друзья	695
12.8. Шаблоны и статические элементы	696
Резюме • Терминология • Типичные ошибки программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 13. Обработка исключений	703
13.1. Введение	704
13.2. Когда должна использоваться обработка исключений	708
13.3. Другие методы обработки ошибок	708
13.4. Основы обработки исключений в C++	709
13.5. Простой пример обработки исключений: деление на нуль	710
13.6. Блоки try	713
13.7. Генерация исключений	713
13.8. Перехват исключений	714
13.9. Повторная генерация исключений	718
13.10. Создание условного выражения	719
13.11. Спецификация исключений	719
13.12. Обработка непредусмотренных исключений	720
13.13. Конструкторы, деструкторы и обработка исключений	721
13.14. Исключения и наследование	722
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 14. Обработка файлов и ввод-вывод потоков строк	737
14.1. Введение	738
14.2. Иерархия данных	739
14.3. Файлы и потоки	741
14.4. Создание файла последовательного доступа	742
14.5. Чтение данных из файла последовательного доступа	746
14.6. Обновление файлов последовательного доступа	750
14.7. Файлы произвольного доступа	751

14.8. Создание файла произвольного доступа	753
14.9. Произвольная запись данных в файл произвольного доступа	754
14.10. Последовательное чтение данных из файла произвольного доступа	756
14.11. Пример: программа обработки запросов	758
14.12. Обработка потоков строк	763
14.13. Ввод-вывод объектов	767
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 15. Структуры данных	781
15.1. Введение	782
15.2. Классы с самоадресацией	783
15.3. Динамическое выделение памяти	784
15.4. Связные списки	786
15.5. Стеки	799
15.6. Очереди	804
15.7. Деревья	807
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения • Специальный раздел: создание вашего собственного компилятора	
Глава 16. Биты, символы, строки и структуры	847
16.1. Введение	848
16.2. Описание структур	849
16.3. Инициализация структур	851
16.4. Использование структур в функциях	852
16.5. Создание синонимов	852
16.6. Пример: эффективное моделирование тасования и раздачи карт	853
16.7. Поразрядные операции	855
16.8. Битовые поля	863
16.9. Библиотека обработки символов	866
16.10. Функции преобразования строк	872
16.11. Функции поиска из библиотеки обработки строк	876
16.12. Функции работы с памятью из библиотеки обработки строк	881
16.13. Другие функции библиотеки обработки строк	884
Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 17. Препроцессор	903
17.1. Введение	904

17.2. Директива препроцессора #include	905
17.3. Директива препроцессора #define: символические константы	906
17.4. Директива препроцессора #define: макросы	908
17.5. Условная компиляция	908
17.6. Директивы препроцессора #error и #pragma	910
17.7. Операции # и ##	910
17.8. Нумерация строк	911
17.9. Предопределенные символические константы	911
17.10. Макрос assert	912
Резюме • Терминология • Хороший стиль программирования • Советы по повышению эффективности • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Глава 18. Другие темы	919
18.1. Введение	920
18.2. Переназначение ввода-вывода в системах UNIX и DOS	921
18.3. Список параметров переменной длины	922
18.4. Использование аргументов командной строки	924
18.5. Замечания по компиляции программ, состоящих из нескольких исходных файлов	924
18.6. Завершение программы при помощи функций exit и atexit	927
18.7. Спецификатор типа volatile	928
18.8. Сuffixы целочисленных и вещественных констант	929
18.9. Обработка сигналов	929
18.10. Динамическое выделение памяти: функции calloc и realloc	932
18.11. Безусловный переход: оператор goto	933
18.12. Объединения	934
18.13. Спецификации связывания	938
18.14. Заключительные замечания	939
Резюме • Терминология • Типичные ошибки программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Приложение А. Стандартная библиотека	949
Приложение Б. Таблица приоритета операций	988
Приложение В. Набор символов ASCII	989
Приложение Г. Системы счисления	991
Г. 1. Введение	992
Г. 2. Сокращенная запись двоичных чисел в восьмеричной и шестнадцатеричной системах счисления	995
Г. 3. Преобразование восьмеричных и шестнадцатеричных чисел в двоичные	997
Г. 4. Преобразование двоичных, восьмеричных и шестнадцатеричных чисел в десятичные	997

Г. 5. Преобразование десятичных чисел в двоичные, восьмеричные и шестнадцатеричные	998
Г.6. Представление отрицательных двоичных чисел: дополнение до двух	999
Резюме • Терминология • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения	
Предметный указатель	1007

Предметный указатель

! операция логическое НЕ
 (отрицание), 139

!= не равно, операция отношения, 62

#define, 421, 422, 685, 905, 906-908,
 909, 911

- DEBUG, 909

- NDEBUG, 484, 912

#elif, 909

#else, 909

#endif, 421, 422, 909

#error, 910

#if, 909

#ifdef DEBUG, 909

#ifndef, 909

#ifndef, 421, 422, 909

#include, 49, 189, 267, 905

- <iomanip.h>, 650

- <iostream.h>, 52

- <math.h>, 180

- <string.h>, 926

#line, 911, 911

#pragma, 910

#undef, 908, 911

% операция вычисления остатка,

& операция адреса, 218, 324

— — — с битовыми полями, 866

& операция поразрядного И, 858

&& логическое И, 137, 140, 211

&= операция поразрядного присваивания И, 862

* операция разыменования, косвенной адресации 323, 324

* операция умножения, 59

+ операция сложения, 55

+ флаг, 59

++ операция инкремента, 114-116

-- операция декремента, 114—116

:: унарная операция разрешения областей действия, 416

< меньше, операция отношения, 62

<< операция сдвига влево, 861

<<= операция присваивания сдвига влево, 862

<= меньше или равно, 62

<ctrl>-c, 930

<ctrl>-d, 130, 642, 745

<ctrl>-z, 130, 642, 745

= операция присваивания, 55, 62

== равно, операция отношения, 62

> больше, операция отношения, 62

>= больше или равно, операция отношения, 62

>> операция сдвига вправо, 861, 862

>>= операция присваивания сдвига вправо, 862

? : операция,

\ символ продолжения макроса в новой строке, 908

\" управляемая последовательность двойных кавычек, 50

\\" управляемая последовательность обратного слэша, 50

\a управляемая последовательность звонка (звуковой сигнал), 50

\n управляемая последовательность новой строки, 50, 636

\r управляемая последовательность возврата каретки, 50

\t управляемая последовательность горизонтальной табуляции, 50

\^ операция поразрядного исключающего ИЛИ, 861

\wedge = операция поразрядного присваивания исключающего ИЛИ, 862
DATE, предопределенная символическая константа, 911, **_FILE_**, предопределенная символическая константа, 911, **_LINE_**, предопределенная символическая константа, 911, **_STDC_**, предопределенная символическая константа, 911, **_TIME_**, предопределенная символическая константа, 911, | операция поразрядного ИЛИ, 860 |= операция присваивания поразрядного ИЛИ, 860, 862 || операция логическое ИЛИ, 137, 140, 211 ~ операция поразрядного НЕ (дополнения, отрицания), 861

A

a.out, 44, 46
abort, 484, 710, 715, 720
abs, 980
acos, 956
ANSI, 487
— C, 33, 41, 331
— —, стандартная библиотека функций, 41
ASCII, 360, 365, 837, 900
assert макрос, 912
assert.h заголовочный файл, 190, 484, 912, 950
atan, 955
atan2, 956
atexit, 927, 928, 975
atof, 873, 971
atoi, 873, 842, 971
atol, 873, 874, 971
auto, 92, 199

B

bad функция-элемент, 660
badbit, 639, 661

Borland C++, 44, 223, 278, 856, 910, 927
break, 92, 130, 132, 135, 136, 717, 933
C

calloc, 932, 974
case метка, 92, 127, 130, 131, 201
catch блок, 713, 714, 715, 718
catch(...), 715, 718, 719, 734
- {throw;}, 734
ceil функция, 182, 957
cerr (стандартный небуферизованный вывод ошибок), 46, 633, 741, 910
char, 53, 92, 127, 189, 779
- *argv[], 924
cin (стандартный ввод), 46, 633, 741
cin, 54, 55, 504, 641, 642
cin.clear, 661
cin.eof(), 642, 661
cin.get() функция, 127, 129
cin.get, 641
cin.getline, 360
cin.tie, 664
class, 146, 270, 368, 407, 412, 596, 739, 848, 863, 905, 936
clear, 750
— функция-элемент, 661
clearerr, 970
clock, 982
clock_t, 982
clog (стандартный буферизованный вывод ошибок), 633, 741
const, 92, 215, 266, 278, 280, 332, 335, 409, 461, 474, 928
— char*, 346
continue оператор, 92, 135, 136
cos, 182, 956
cosh, 956
cout (<<), 46, 49, 50, 52, 639, 640, 633, 635, 741
CPU, 45
ctune, 983
ctype.h заголовочный файл, 190, 332, 867, 908, 951

D

DEBUG, 909

dec манипулятор потока, 646

default, 92, 127, 130, 131, 192, 334

delete операция, 479, 480, 613

— [] операция, 479

dequeue, 805, 807

div, 980

do/while, 92, 133, 134, 135, 146

DOS, 921, 924, 930

double, 92, 189, 779

E

EBCDIC, 365

else, 92, 94-98

endl манипулятор потока, 56, 108, 636

ends манипулятор потока строк, 764

enqueue, 805, 807

enum ключевое слово, 92, 195, 198

eof, 756, 757

— функция-элемент, 642, 661

EOF, 129, 130, 641, 644, 867

eofbit, 660, 661

errno, 709, 949

errno.h заголовочный файл, 190, 949

exit, 927, 928, 975

EXIT_FAILURE, 927, 975

EXIT_SUCCESS, 927, 975

exp функция, 956

extern, 92, 200, 201, 925

— "C", 938

F

F суффикс, 929

f суффикс, 929

fabs функция, 182, 957

fail функция-элемент, 660

failbit, 639, 660, 744, 765

fclose, 963

feof, 970

ferror, 970

fflush, 963

fgetc, 966

fgetpos, 969

fgets, 967

FIFO (первый вошел — первый вышел), 488, 807

FILE, 962

fill функция-элемент, 653, 655

fixed флаг, 657

flags функция-элемент, 650, 658, 659

float, 92, 105, 108, 189, 779

float.h заголовочный файл, 190

fmod функция, 182, 957

fopen, 963

for структура повторения, 92, 119-127, 135, 136, 146, 263, 294

fprintf, 965

fputc, 967

fputs, 967

fread, 968

free, 479, 480, 932, 932, 974

freopen, 964

frexp, 956

friend, 92, 423, 429, 436, 437, 488, 502, 503, 521, 577, 685, 695, 696

fscanf, 965

fseek, 969

fsetpos, 969

fstream, 634, 741, 743, 758, 759

fstream.h заголовочный файл, 633, 741, 742

ftell, 970

fwrite, 969

G

gcount функция-элемент, 645, 646

get функция-элемент, 425, 436, 641, 643

getc, 967

getchar, 967

getchar(), 908

getenv, 975

getline функция-элемент, 360, 504, 504, 643, 664, 798

gets, 967

gmtime, 983

good функция-элемент, 660

goodbit, 660

goto, 92, 933, 934

Н

hex манипулятор потока, 646

И

IBM, 38, 43

if, 61, 67, 92-94, 97, 98, 127, 146

if/else, 92, 94-98, 127, 146

ifstream, 635, 741, 743, 756

ignore функция-элемент, 661

inline, 92, 214, 215, 220, 258, 417, 907

int, 53, 92, 189, 779

internal флаг, 654

iomanip.h заголовочный файл, 108, 263, 633, 646, 905

ios базовый класс, 635

ios::adjustfield, 654, 656, 657, 659

ios::app, 743, 764

ios::ate, 743, 755

ios::badbit, 660

ios::basefield статический элемент, 656

ios::beg, 748

ios::cur, 748

ios::dec, 656, 659

ios::end, 748

ios::eofbit, 660

ios::failbit, 660

ios::fixed флаг, 108, 126, 657

ios::fixed, 657

ios::floatfield, 657, 659

ios::goodbit, 660

ios::hex, 656, 659

ios::in, 746, 759

ios::internal, 654, 659

ios::left, 654, 659, 750, 761

ios::nocreate, 743

ios::noreplace, 743

ios::oct, 656, 659

ios::out, 743, 744, 759, 761, 764

ios::right, 654, 659, 750, 761

ios::scientific флаг, 657

ios::showbase, 654, 657

ios::showpoint, 108, 126, 651, 750

ios::showpos, 654

ios::skipws флаг, 681

ios::trunc, 743

ios::uppercase, 658

iostream класс, 632, 634, 635

iostream.h заголовочный файл, 49, 52, 324, 741, 763, 905

isalnum, 867, 868, 869, 951

isalpha, 867, 868, 869, 951

iscntrl, 867, 871, 951

isdigit, 867, 868, 869, 951

isgraph, 867, 870, 871, 951,

islower, 867, 869, 870, 951

isprint, 867, 870, 871, 951

ispunct, 867, 870, 951

isspace, 867, 870, 951

isupper, 867, 869, 870, 951

isxdigit, 867, 868, 869, 951

Л

L суффикс, 929

l суффикс, 929

L-величина (lvalue, левое значение),

141, 218, 262, 324, 441, 519, 521, 530

labs, 976

ldexp, 956

ldiv, 977

left флаг, 654

LIFO последним вошел — первым вышел, 485, 692, 799

limits.h заголовочный файл, 190, 985

locale.h заголовочный файл, 190, 952

localeconv, 953

localtime, 983

log функция, 182, 956

log10 функция, 182, 956

long double, 189, 779

long int, 92, 133, 188, 189, 779

longjmp, 709, 958

lvalue (левое значение), 141, 218, 262,

324, 441, 519, 521, 530

М

main(), 50

make утилита, 927
makefile, 927
malloc, 479
math.h заголовочный файл, 125, 126,
 180, 188, 190, 955
mblen, 977
mbstowcs, 977
mbtowc, 977
memchr, 881, 883, 980
memcmp, 881, 882, 979
memcpy, 881, 882, 978
memmove, 881, 882, 978
memset, 881, 884, 981
mktime, 982
modf, 957 MS DOS, 130

N

new операция, 92, 446, 479, 480, 484,
 489, 516, 517, 528, 734, 784, 796,
 932

new_handler, 709 NULL, 324, 484,
 852, 875, 877, 880, 883, 909, 950,
 963, 978, 982

O

ост манипулятор потока, 646

ofstream, 635, 741, 743, 744, 745, 748,
 756

open, 744

operator функция-элемент класса ios,
 744

ostream, 763, 765, 767

P

pd, 223

peek функция-элемент, 644

perror, 970

pf, 223

pop, 485, 800

pow, 125, 253

pragma, 910

precision функция-элемент, 647, 657

print, 800, 807

printf, 965

private:, 425

protected:, 92, 423

public:, 425

push, 485, 800

put функция-элемент, 635, 638, 639,
 642

putback функция-элемент, 644

putc, 967

putchar, 908, 968

puts, 968

R

R-величина (rvalue, правое значение),
 141, 218, 515

raise, 958

rand, 191, 973

RAND_MAX, 190, 194

rdstate функция-элемент, 660, 765

read функция-элемент, 645, 646

realloc, 932, 933, 974

register, 92, 199, 200, 324

resetiosflags манипулятор потока, 650

return оператор, 712, 718

right флаг, 654

rvalue (правое значение), 141, 218,
 515

S

scanf, 965

scientific флаг, 657

SEEK_CUR, 961

SEEK_END, 961

SEEK_SET, 961

seekg, 748, 750, 758, 761, 762

seekp, 748, 754, 762

set_new_handler, 709

set_new_handler (), 709

set_terminate, 715, 720, 735

set_terminate (), 715, 720, 735

set_unexpected, 720, 735

set_unexpected (), 720, 735

setbase манипулятор потока, 646

setbuf, 966

setf функция-элемент, 650, 654, 656

setfill манипулятор потока, 655

setiosflags манипулятор потока, 108,

 126, 650, 654, 750, 758, 761

setjmp, 709, 957
setjmp.h заголовочный файл, 190,
709, 957
setjmp/longjmp, 735
setlocale, 953
setprecision манипулятор потока, 108,
126, 750
setprecision, 108, 126, 750
setvbuf, 966
setw манипулятор потока, 126, 263,
647, 654, 750, 758, 761
short, 92, 133, 188, 189
— int, 133, 189, 779
showbase флаг, 657
showpoint флаг, 652
showpoint, 652
SIGABRT, 930, 930, 958
SIGFPE, 930, 958
SIGGILL, 930, 958
SIGINT, 930, 958
signal, 930, 959
signal.h заголовочный файл, 190, 930,
930, 958
signal_handler, 930
SIGSEGV, 930, 959
SIGTERM, 930, 959
sin, 182
sinh, 956
size_t, 361, 877, 950, 963, 982, 982
sizeof, 92, 339, 340
skipws флаг, 651
sprintf, 965
sqrt, 180, 182, 188, 957
srand, 192, 194, 195, 974
sscanf, 968
static, 92, 199, 200, 201, 203, 219, 273,
480, 927, 936
stdarg.h заголовочный файл, 190, 922,
961
stddef.h заголовочный файл, 190, 361,
950
stderr, 962
stdin, 962

stdio.h заголовочный файл, 190, 908,
961
stdiostream.h заголовочный файл, 633
stdlib библиотека утилит общего
назначения 484, 872, 912, 927,
932
stdlib.h заголовочный файл, 190, 484,
741, 872, 927, 932, 971
stdout, 962
str функция-элемент, 763, 764
strcat, 362, 363, 528, 926, 979
strchr, 877, 878, 899, 980
strcmp, 362, 364, 365, 926, 979
strcoll, 979
strcpy, 361, 362, 363, 528, 978
strcspn, 877, 878, 980
stream.h заголовочный файл, 633, 763
strerror, 884, 885, 981
strftime, 988, 983
string.h заголовочный файл, 190, 361,
779, 978
strlen, 362, 367, 981
strncat, 363, 979
strncmp, 362, 364, 365, 979
strncpy, 361, 362, 363, 978
strupr, 877, 879, 980
 strrchr, 877, 879, 980
strspn, 877, 878, 980
strstr, 877, 880, 899, 981
strtod, 873, 874, 875, 971
strtok, 362, 365, 366, 367, 832, 842,
981
strtol, 873, 875, 876, 972
strtoul, 873, 876, 973
struct tm, 982
struct, 92, 408, 425, 739, 783
strxfrm, 980
switch оператор, 597, 660, 717
— структура выбора, 92, 92, 127-133,
135, 201, 390
system, 975
T
tan, 182

tanh, 956
tellg, 748
telp, 748
template, 92, 224, 686, 783, 787, 805
terminate, 710, 715, 717, 718, 720, 721, 735
terminate.h заголовочный файл, 721
this указатель, 474, 475, 476, 483, 420, 502
throw, 92, 713
throw(), 720
throw;, 719
tie функция-элемент, 664
time.h заголовочный файл, 190, 194, 986
tolower, 867, 869, 870, 951
toupper, 867, 869, 870, 952
try, 92
typedef, 92, 852, 853
U
и суффикс, 929
U суффикс, 929
ul суффикс, 929
UL суффикс, 929
unexpected, 720, 721, 735
unexpected.h заголовочный файл, 721
ungets, 968
union, 92, 863, 934-938
UNIX, 38, 40, 44, 46, 130, 642, 707, 745, 921, 930
unsigned, 92, 189, 192
— char, 189, 779, 883
— int, 189, 192, 361, 779, 877
— long int, 189, 206, 207, 361, 876, 779, 877
— short int, 189, 779
uppercase флаг, 658
V
va_arg, 922, 960
va_end, 922, 960
va_list, 922, 960
va_start, 922, 960
vfprintf, 966

virtual, 92, 936
void, 92
— * (указатель на void), 343, 638, 641, 716, 881
— — функция-операция элемент класса ios, 744
volatile, 92
vprintf, 966
vsprintf, 966
W
wcstombs, 978
wctomb, 977
while, 92, 99, 100, 104, 117, 118, 120, 146
width функция-элемент, 647
write функция-элемент, 635, 645, 646, 753-755
ws манипулятор потока, 650
Z
zc, 223

А
абсолютная величина, 182
абстрагирование, 66, 553
абстрактный базовый класс, 599-602, 614
— класс, 600, 601
— тип данных, 411, 485, 487, 488, 498
аварийное завершение программы, 709
автоматическая переменная, 203, 800
автоматический класс памяти, 199, 260, 274
— массив, 264, 267
— —, инициализация, 274
автоматическое преобразование, 689
АДА, 43
аддитивные операции, 64, 140
адрес, 323, 324, 328, 344
— (&) объединения, 935
— (&) структуры, 850
— битового поля, 866

- как аргумент, 328
- операнда, 324
- операция (&), 324, 327, 336, 500
- переменной структуры, 851
- функции, 354
- адресуемая единица памяти, 866
- аккумулятор, 386, 389
- алгебра, 57, 59, 60
- алгебраическое выражение, 57, 59, 65
- алгоритм, 89, 93, 94, 112, 122, 130, 134, 281
- амперсанд (&), 216
- анонимное объединение, 936, 937
- аппаратное обеспечение, 33, 35
- аргумент, 179, 181, 183, 686, 905
- аргументы командной строки, 924
 - по умолчанию, 220, 429
 - — — и перегруженные операции, 501
 - функции по умолчанию, 220, 429
- арифметика с плавающей запятой, 499
- арифметико-логическое устройство (АЛУ), 36
- арифметические операции, 57, 59
- арифметическое переполнение, 705
- ассоциативность, 59, 116, 140
 - и перегрузка, 501
 - слева направо, 62, 64, 116, 140
 - справа налево, 64, 107, 116, 129
- атрибуты, 66, 226, 369, 413, 552, 574
 - объектов, 227
- Б**
 - база данных, 741
 - базовая задача рекурсии, 205, 208, 211, 553
 - базовый класс, 553, 555, 556, 599, 600
 - — ios, 641
 - — защищенный, 567
 - — исключений, 709
 - — классов istream, ostream и iostream, 60
 - — открытый, 566
 - библиотека классов, 41, 420, 445, 487, 553, 573, 635
 - библиотечные функции, 708, 938
 - бинарная операция разрешения области действия (:), 416, 417, 480, 586, 696
 - бинарное дерево, 782, 807, 808, 825
 - — обход, 827

бинарные арифметические операции, 107
— операции, 55, 57, 140, 210, 501, 508
бит, 738, 848
битовое поле, 863—866
биты, операции, 739, 855, 866
блок, 98, 185, 199, 260, 434
— catch, 714—719
— try, 710, 712, 713, 714, 715, 716, 721, 735
— памяти, 479, 872
— центрального процессора (CPU), 36
блок—схема структуры do/while, 135
— структуры for, 123
блок—схемы, 91, 92, 94, 100, 123, 131
буква, 739, 866
буфер, 643, 660
буферизованный вывод, 635
быстрая разработка приложений (rapid application development—RAD), 445
БЭЙСИК, 783, 845
В
ввод данных из массива символов, 763
— — из файла, 741
— символов функцией—элементом getline, 643
ввод—вывод, 631, 961
— в память, 763
— определенных пользователем типов, 631, 661
— стандартных типов, 631
— файлов, 635
— форматированный, 126
—, операции, 386
—, функции, 360
вероятность, 191
вертикальная табуляция ('\v'), 870
взаимодействие объектов, 367, 368, 486

виртуальные функции, 573, 596, 597, 601,
— — базового класса, 599
— — чистые, 599, 601, 602, 607, 614, 618, 603, 607, 613, 614
вложенные блоки, 185, 201
— вызовы функций, 709, 710, 714, 735
— области действия в блоке try, 714
— управляющие структуры, 109, 130, 143, 933
внутреннее представление float, 935
возвращение, 119, 179, 186, 719
восьмеричная система счисления (с основанием 8), 635, 646, 656, 876, 992, 994, 997
— цифра, 656, 995
встраиваемые inline функции, 214, 215, 220, 258, 417, 907
встроенные типы, 188, 407, 488, 635, 686
— — данных, 486
выбор, 146, 933
вывод значений char *, 638
— значений с плавающей запятой, 635
— символов, 638
— строк, 635
— указателей, 635
— целых значений, 635
выделение блока памяти, 932
— памяти, 709
вызов деструктора автоматического объекта, 709
— конструктора базового класса, 571
— по значению, 216, 217, 218, 326, 327, 329, 331, 334, 336, 445, 515, 516, 528, 852
— по ссылке моделируемый, 277, 334
— по ссылке, 216, 217, 218, 322, 326, 328, 331, 334, 336, 409, 443, 445, 516, 852

— функции по константной ссылке, 445
вызов функции по ссылке, 326
вызываемая функция, 179, 326
выравнивание, 653, 654, 935
— вправо, 126, 653, 654
—, границы, 784
— по границам слова, 851, 866
выражение, 94
выход из глубоко вложенных структур, 933
— из структуры, 135
— из функции, 51 вычитание, 36, 58, 59, 486
— указателей, 342, 343
Г
генерация исключений, 705, 709, 710, 714, 716, 719, 720, 721
— случайных чисел, 148, 191, 194, 349
глобальная область действия, 433
— переменная, 200, 201, 202, 203, 221, 480, 927, 927, 936
— функция, 425, 696
граници выравнивания, 784
Д
дамп, 390, 391
данные—элементы, 67, 297, 407, 413, 429
— закрытые, 425, 436, 473
дата, 190, 400, 986
двоеточие (:), 412, 559, 864
двоичная система счисления (с основанием 2), 856, 992, 994, 995, 997, 999
— цифра, 739
двоичный поиск, 285, 288, 319
— в сортированном массиве, 288
двумерный массив, 290, 291, 293, 294, 296, 297, 349, 782
декремент, 114—116
— указателя, 342

деление на нуль, 46, 104, 391, 486, 705, 930
— — —, исключение, 712
— — —, ошибка, 712
дерево, 260, 322, 849
— двоичного поиска, 807, 809, 813, 814, 826
деструктор, 416, 433, 434, 480, 516, 521, 528, 563, 566, 570, 721, 745, 787, 936
— базового класса, 613
деструкторы, вызов для автоматического объекта, 709
десятичная система счисления (основание 10), 635, 646, 876, 992, 994
— точка, 53, 102, 105, 108, 126, 126
— цифра, 739, 995
динамически выделенная область памяти, 434, 443, 500, 515, 516, 517, 528
динамические структуры данных, 260, 322, 782
динамическое распределение памяти, 479, 485, 566, 795—797, 932, 933
директивы препроцессора, 44, 49, 52, 267, 421, 422, 685, 904, 906, 926
диск, 35, 36, 44, 631, 739, 751
—, свободное пространство, 744, 907
—, файлы, 763, 768
дно стека, 800
добавление одной строки к другой, 362
доступ к закрытым элементам класса, 425
— к защищенным элементам базового класса, 554
— к элементам, 412, 423, 472
— — — класса, 418
— — — структуры, 409, 850
— по умолчанию для класса, 423
дробная часть, 58

дружественные функции, 423, 437, 471, 472, 500, 507, 508, 696
друзья, 423, 429, 436, 437, 488, 502, 503, 521, 577, 586, 686, 695, 696
Ж
живучесть программы, 704
З
завершение программы, 705, 709, 927, 928
заголовочный файл, 108, 189, 190, 191, 324, 361, 420, 424, 445, 573, 633, 721, 905, 927
загрузка, 46, 386
задание начальных условий, см.
 инициализация
закрытое наследование, 556, 566, 567, 800
закрытые данные—элементы, 425, 436, 473, 480
запись, 333, 739, 741, 741, 748, 751, 754, 849
заполнение символом, 635, 647, 653, 655
зацикливание, 99, 107, 121, 134, 208, 212
защищенное наследование, 556, 566, 567
защищенный базовый класс, 567
звездочка (*), 57, 324
звук, 50
золотое сечение, 208
И
идентификатор, 53, 92, 685
иерархия, 555, 596, 597, 600, 601, 611, 613, 614, 741
— классов, 562, 600, 601, 613, 741
— потоков ввода—вывода, 634, 741
именованная константа, 266
именованный параметр обработчика
 catch, 714
имя, 56, 116, 223, 261
— массива, 261
— — как константный указатель, 344

индексы, 261, 272, 294
—, выход за допустимые пределы, 521, 714
инициализаторы, 264, 264, 267, 272, 292, 429, 480, 851, 936
— элементов, 464, 468, 559, 568
инициализация вновь создаваемого объекта, 479
— константных встроенных типов данных, 467
— массива, 264, 271, 292
— переменных, 103, 117
— ссылки, 218
инициализация статического элемента с областью действия файл, 480, 696
— структуры, 851
— указателя, 323, 335
— элементов базового класса, 559
инкапсуляция, 407, 417, 507, 553, 556, 562
инкремент, 114—116
— указателя, 342
интерпретатор, 40
интерфейс, 66, 407, 416, 417, 420, 441, 485, 601, 602, 614
— открытый, 420, 425, 485, 553, 602, 800, 805
инфiksная форма выражения, 821, 841, 845
— — —, преобразование в постфиксную форму, 821, 841, 845
исключение деления на нуль, 712
исключения, 708, 712, 719
— операций с плавающей запятой, 930
— родственные, 716
—, базовый класс, 709
—, классы, 709, 714
—, обработка, 484, 708, 709, 721
—, объекты, 713, 714

—, преобразование одного типа в другой, 716
исполняемый файл, 927
исходный файл (код), 420, 911
К
канал связи, 741
квадратные скобки ([]), 261
клавиатура, 35, 36, 55, 631, 634, 741, 921
класс, 67, 146, 368, 407, 411, 596, 739, 849, 863, 905, 936
— ios, 660
—, клиенты, 423
—, объявление, 419, 472
—, описание, 416, 417, 419, 420, 432, 445, 474, 508, 689
— памяти автоматический, 199, 260, 274
— — статический, 199, 200, 201
— с самоадресацией, 783, 784, 796
—, тело описания, 412
—, шаблоны, 689—694, 695, 696, 782, 787, 800, 802, 803, 805, 807, 845
классы как элементы других классов, 467
— конкретные, 599, 600, 601
— контейнеры, 426, 471, 489
— памяти, 198, 199, 926
—, библиотеки, 41, 420, 445, 487, 554, 573, 635
—, иерархия, 562, 600, 601, 613, 741
клиенты класса, 423
ключевая запись (поле), 740, 741, 751, 753, 777
ключевые слова, 92
— — C++, 92
КОБОЛ, 43
код символа, 365
командная строка, 921
— —, аргументы, 924
комментарий, 49
компилятор, 39, 40, 44, 782, 783, 800, 828—844, 927, 996

компиляция, 44, 420
—, ошибки, 51
композиция, 417, 446, 467, 489, 554, 575, 694, 782, 800, 803
компьютер, 35
— Apple, 38
конвойер (), 921
конец файла, 129, 130, 640, 641, 642, 660, 745, 748, 921, 924
конкатенация, см. сцепление
константная переменная, 464, 905
— ссылка, 445
— функция, 598
— функция—элемент, 461
константное выражение, 266
константный объект, 266, 464
— указатель, 331
— — на константные данные, 335
— — на неконстантные данные, 335
— элемент данных, 461, 466
— — класса, 466
конструктор, 426, 429, 434, 461, 467, 479, 480, 481, 489, 516, 559, 563, 567, 570, 721, 743
— базового класса, 568, 571
— — —, вызов, 571
— копий, 515, 516, 518, 526, 526, 528, 529, 721
— с аргументами по умолчанию, 429
конструкторы и обработка исключений, 721
— и объединения, 936
— преобразования, 520, 526, 526, 528, 529, 572
копирование объектов, 327, 443, 530
— строки, 361
— частей одной и той же строки, 882
косвенный базовый класс, 568
курсор, 50, 52
Л
левое поддерево, 807, 809, 813
линейные структуры данных, 796,

линейный поиск, 285, 287, 319
локальная переменная, 183, 184, 199,
200, 203, 334
локальный автоматический объект,
434

М

макрос, 190, 685, 904, 905, 906, 922
макрос, расширение, 906—908
манипулятор потока
 непараметризованный, 108, 650
маскирование, 859
массив, 260, 322, 434, 487, 489, 508,
738, 849, 849
— автоматический, 264, 267, 274
— локальный, 274
— —, инициализация, 274
— битов, 866
— как аргумент функции, 277
— как параметр в описании функции,
278
— многомерный, 290, 291, 292
— объектов, 721
— одномерный, 292, 297, 331, 782
массив операций, 486
—, присваивание, 487
— символов, 271, 359, 632, 642
— строк, 348, 924
— структур, 852
— указателей, 348
— базового класса, 620
— — на функции, 357, 358, 395, 396,
613
—, ввод—вывод, 270, 487
—, выход индекса за пределы, 734
—, границы, 270, 270
—, индексация, 334, 344, 345, 714
—, инициализация, 264, 274
—, объявление, 263
—, операция индексации ([]), 515
—, передаваемый вызовом по
значению, 277
—, передача автоматического
массива по ссылке, 336

—, размер, 276, 338
массивы и указатели, 278, 344
—, сравнение, 487, 508
масштабирование, 191, 195
машинно—зависимый, 39, 341, 784,
851, 856, 862, 866
машинный язык, 39, 44, 782, 783, 800
метка, 201
многомерный массив, 290, 291, 292
множественное наследование, 66,
554, 580, 584, 586, 634
мобильность, 47
моделируемый вызов по ссылке, 276,
334
модифицируемость программ, 410,
425
мультипликативные операции, 64,
108, 140

Н

набор символов, 132, 365, 651
наибольший общий делитель, 251,
255
наследование, 66, 368, 417, 553, 554,
555, 568, 573, 574, 575, 577, 599,
600, 601, 607, 612, 618, 634, 686,
721, 782, 805
— закрытое, 556, 566, 567, 800
— защищенное, 556, 567, 567
— множественное, 66, 554, 580, 584,
586, 634
— открытое, 556, 559, 566, 577, 567,
607
— прямое, 554, 580, 584, 586, 634
неисправимая ошибка, 46, 98, 104,
187, 324, 391
нелинейные структуры данных, 796
нелокальный переход, 958
ненормальное завершение
 программы, 930
непараметризованный манипулятор
потока, 108, 650
неполиморфное поведение, 600
неразрешенная ссылка, 832, 926

нестатическая функция—элемент, 483 506, 520, 531
нестатический элемент класса, 483
неявное преобразование, 107, 529, 562
— приведение типов, 556, 562
нулевой символ ('\0'), 272, 345, 359, 360, 362, 366, 642, 643, 646, 764
— указатель (0), 783, 784, 933

О

область действия, 199, 200, 201, 467, 705, 926, 927
— — блок, 201
— — класс, 417, 419, 468, 480
— — функция, 201, 419
обработка исключений, 484, 704—735
— — блок try, 713, 714, 716, 721
— потока строк, 763
— сигналов, 929—932, 959
обратный слэш (), 50, 784, 908
— —, управляющая последовательность ('\b'), 870
обход бинарного дерева по слоям, 814, 827
объединение, 863, 934—938
объект, 35, 41, 65, 67, 90, 147, 407
— cin, 55, 639
— автоматический, 434, 719, 721
— — локальный, 434
— как элемент другого объекта, 467
— потока ввода (cin), 54, 55
объектная ориентация, 65
объектно—ориентированное программирование (ООП), 33, 35, 36, 66, 132, 147, 208, 367, 369, 407, 553, 573, 597, 705
— проектирование, 41, 65, 147, 148, 176, 297, 367, 369, 407, 445, 471, 574
объектный код, 44, 46, 49
объекты базового класса, 553, 556, 573, 596, 719

—, взаимодействие, 367, 368, 486
объявления, 53, 54, 117, 185
ограничитель, 642, 643
округление, 58, 108, 182, 248
операции поразрядные, 855—863
операнд, 55, 386
оператор, 50, 133, 185
— break, 130, 131, 132, 135, 136, 717, 933
— continue, 135, 136
— goto, 933, 934
— switch, 596, 717
— арифметического присваивания, 113, 862
— присваивания, 55, 129, 441
— составной, 97, 98, 107, 185
операции ввода—вывода, 386
— мультиплексивные, 64, 108, 140
— отношения, 62, 64, 94, 123, 137, 140, 343, 508
— передачи управления, 386
— присваивания +=, -=, *=, /=, %=, 113, 140, 568
— проверки равенства, 61, 62, 64, 94, 140, 508, 508
— с битами, 739, 855—863, 866
— со строками символов, 866, 872
— унарные, 107, 140, 324, 339, 501, 506
—, правила следования, 58, 59, 62
—, старшинство, 58, 62, 64, 96, 108, 116, 121, 139, 140
операционная система, 37, 38, 40, 360, 601, 632, 745, 782, 921, 930, 996
операция (+), 500, 503
— (+=), 508, 529
операция (<), 529
— (<=), 529
— (>), 529
— (>=), 529, 530
— <<, 506, 515
— >>, 504, 514, 515

- `char *`, 520
- `new`, 446, 479, 480, 424, 516, 517, 528, 734, 784, 796, 932
- `sizeof`, 876
- `void *`, функция—элемент, 661
- взять из потока `>>`, 751
- вставить в поток `<<`, 751
- выбора элемента `(.)`, 599
- вычисления остатка `(%)`, 57, 58, 59, 85, 191, 195
- декремента `(--)`, 114—116
- доступа к элементу `(.)`, 408
- индексации `([])`, 515, 519, 530
- инкремента `(++)`, 114—116
- логического И `(&&)`, 137, 140, 858
- — ИЛИ `(||)`, 137, 138, 140, 860
- — НЕ (отрицание) `(!)`, 137, 138, 140
- приведения типа, 108, 188, 520, 641
 - — унарная, 107
- присваивания `(=)`, 55, 62, 64, 116, 443, 508, 516, 517, 935
- базового класса, 568
- проверки неравенства `(!=)`, 518
- равенства `(==)`, 62, 139, 518
- разрешения области действия, бинарная `(::)`, 416, 417, 480, 584, 696
- разыменования `(*)`, 323, 324, 327
- последования запятая `(,)`, 64, 121, 140, 211
- стрелка `(->)`, 408, 419, 935
- точка `(.)`, 408, 475, 935
- условная `(?:)`, 95, 116, 140, 640, 869
- описание структур, 408, 849, 850, 863, 866
- функции, 182, 183, 188, 201, 216, 273, 685, 693
- оптимизирующий компилятор, 127, 200, 612
- освобождение динамически выделенной памяти, 564
- области памяти, выделенной операцией `new`, 718
- основная память, 631
- открытое наследование, 556, 559, 566, 577, 567, 607
- открытые элементы базового класса, 556
- открытый базовый класс, 566
- интерфейс, 420, 425, 485, 553, 602, 800, 805
- отладка, 44, 47, 187, 192, 212, 484, 553, 562, 597, 909, 911
- отрицательное значение в `char`, 867
- отрицательные числа, 1004
- очередь, 260, 322, 489, 782, 805, 805, 849
- ошибка времени выполнения, 46, 713
- деления на нуль, 712
- доступа к памяти, 930
- занижения (или завышения) на единицу, 120, 262
- неисправимая, 46, 98, 104, 187, 324, 391
- синтаксическая, 51, 58, 62, 96, 98, 141
- ошибки компиляции, 51
- , обработка, 704, 705, 708, 710
- , состояния, 639, 660
- , флаги, 660
- П
- пакет, 37
- пакетная обработка, 37
- память, 35, 36, 56, 199, 263, 277, 291, 324, 408, 631, 850
- , адресация, 324, 339
- вторичная, 36, 44, 738
- , распределение, 190, 479
- параметр, 183, 199, 200
- параметризованный тип, 686, 690
- Паскаль, 33, 34, 43, 44
- перегруженная операция `++`, 531
 - `+=`, 502, 508, 529, 532
 - `--`, 531

- — [], 503, 515, 530
- — взять из потока >>, 504, 515, 639, 661
- — вставить в поток <<, 411, 504, 515, 520, 562, 620, 635, 637, 639, 661, 767
- — вызова функции (), 503, 530
- — приведения типов, 520, 641
- — присваивания (=), 502, 503, 515, 516, 517, 521, 528, 572, 935
- — проверки равенства (==), 515, 518, 529
- — перегруженные операции отношения, 529
- функции, 222, 223, 224, 685, 688
- перегрузка, 631, 635, 694
- бинарных операций, 508
- конструктора, 429
- операций, 56, 411, 500, 631, 635, 856
- унарных операций, 506
- функций, 222, 631
- — с параметрами по умолчанию, 224
- функций—элементов, 419, 461
- передача имени файла в программу, 924
- массива в функцию, 276, 338
- по ссылке, 216
- структуры в функцию, 852
- элемента в функцию, 852
- переменная, 53, 67, 91, 92, 103, 260, 407
- автоматическая локальная, 202
- локальная, 183, 184, 199, 200, 203, 334
- переменные только для чтения, 266
- переназначение ввода, 921
- —, символ <, 921
- вывода, 921
- —, символ >, 921
- переносимость, 47
- переопределение виртуальных функций, 599, 611
- функций производного класса, 618
- функций—элементов базового класса, 597, 618
- переполнение, 930
- аккумулятора, 391
- перехват исключений, 705, 709, 710, 714, 721
- персональный компьютер, 35, 38, 44, 339
- плавающая запятая, арифметика, 499
- —, вычисления с целыми значениями, 107
- плавающая запятая, исключения, 930
- —, числа, значения, пределы, 102, 105, 107, 108, 109, 118, 181, 190, 646
- платформа, 40, 926
- побитовое копирование, 443, 500
- побочные эффекты, 200, 210, 216, 507, 685, 908
- повторение, 99, 100, 146, 146, 933
- , управляемое меткой, 102—109
- повторное использование, 417, 445, 580, 612, 631, 685, 687, 800, 926
- подсчет числа объектов класса, 481
- поиск, 285, 287, 288, 489, 787, 814, 826
- блока памяти, 872, 881
- , функции библиотеки обработки строк, 876—880
- поле, 739, 741, 751
- полиморфизм, 132, 368, 552, 554, 573, 577, 596, 597, 600, 601, 612
- поля ширина, 126, 263, 635, 645, 647
- поразрядная операция И (&), 856, 858, 859
- — ИЛИ (), 650, 856, 858, 859, 860
- — исключающего ИЛИ (^), 856, 858, 860
- — НЕ (дополнение, отрицание) (~), 856, 861
- — присваивания, 862
- поразрядные операции сдвига, 861

последним вошел—первым вышел (LIFO), 485, 692, 799 последовательность вычисления выражений, 828 — операндов, 210 постфиксная форма выражения, 821, 841, 845 — — —, преобразование из инфиксной формы, 821, 841, 845 поток, 50, 660, 741, 743, 928 — ввода, 639 — управления, 65, 94, 123, 130, 135, 719 —, форматы состояния, 650 потоки ввода—вывода файлов, 763 — —, иерархия, 635, 741 правая фигурная скобка {}, 50, 53, 56, 423 правила следования операций, 58, 59, 62 — формирования структурированных программ, 143 правое поддерево, 807, 809, 809, 813 преобразование двоичных, восьмеричных и шестнадцатеричных чисел в десятичные, 1003 — из инфиксной формы в постфиксную, 821, 841, 845 преобразование неявное, 107, 529, 562 — одного типа исключения в другой, 717 — строк в числовые значения, 872 — указателя базового класса в указатель производного класса, 556 преобразования встроенных типов, 520 — типов, 519 препроцессор, 44, 45, 108, 484, 904 —, директивы, 44, 49, 52, 267, 421, 422, 685, 904, 906, 926

прерывание, 705, 930 приведение типа, 324, 343, 556, 562, 753 — выражения, 906 — — — указателя, возвращаемого malloc, 479 — указателя производного класса к типу указателя базового класса, 573 приглашение, 55, 745 приложение, 487, 270 принцип наименьших привилегий, 199, 278, 330, 331, 337, 346, 420, 425, 461, 748, 926, 927 присваивание массива массиву, 270 — объединений одного типа, 935 — побитовым копированием, 443 — структур одного типа, 850 пробел (' '), 64, 365, 870 программа со многими исходными файлами, 199, 201, 420, 924—927 программирование структурное, 33, 34, 36, 41, 44, 48, 65, 90, 91, 135, 407, 705, 933 — объектно—ориентированное, 33, 34, 36, 41, 66, 132, 146, 208, 367, 369, 407, 554, 573, 597, 705 программное обеспечение, 33, 36, 44, 47, 66, 91, 553, 573, 612 программы, завершение, 705, 710, 927, 928 —, легкость чтения, 64, 93, 97, 111 —, модифицируемость, 410, 425 производный класс, 554, 555, 556, 574, 597, 599, 600, 602, 603, 715 — — —, объекты 554, 556, 559, 571, 573, 597, 601, 611 простота чтения, 64, 93, 97, 111, 119 пространство на диске, 745, 907 прототип, 183, 328, 331, 416, 597, 905, 927, 932, 938 прототипы функций, 125, 183, 184, 188, 189, 190, 201, 214, 216, 328,

331, 338, 416, 432, 472, 597, 905, 927, 926, 938
процедура, 90
прямое наследование, 554, 580, 584, 586, 635
прямой базовый класс, 567
псевдослучайные числа, 192
псевдокод, 89, 90, 93, 95, 96, 97, 99, 101, 103, 105, 110, 111, 112
псевдоним, 218, 219, 441
P
разделение интерфейса и реализации, 416, 419
разделители, 645, 649, 910
разыменование указателя, 324, 328
— — this, 475
— — void*, 343
рандомизация, 192
расширение макроса, 906—908
расширяемость, 500, 507, 601, 631
— C++, 661, 704
расширяемые языки, 208, 270, 413, 487
реализация, 66, 407, 416, 417, 419, 429, 432, 485
редактор, 44, 45, 90
рекурсивная функция, 205, 207, 208, 787
рекурсивный вызов, 206, 207, 208, 800
— двоичный поиск, 212, 285, 319
рекурсия, 162, 206, 212, 516, 562, 566
решение, 33, 92, 93, 94, 123, 126, 130, 135, 137, 146
родительский узел, 807, 825
C
самоадресуемая структура, 408, 850
самоадресуемый класс, 783, 784, 796
самоприсваивание, 476, 517, 521, 528
сброс буфера, 56, 635, 636
— потока, 639, 645, 928

связный список, 322, 426, 485, 489, 515, 554, 556, 573, 786—799, 845, 849
связывание, 44, 46, 198, 199, 420, 783, 807, 926
— выходного потока с входным, 664
сервер, 804
сети, 631, 705, 804, 855, 996
сеть компьютеров, 38, 804
C, 33, 685, 938, 939
сигнатура, 223, 531, 562, 599, 600
символ, 127, 739, 848, 866
— добавления в вывод >>, 921
—, код, 365
— новая строка ('\n'), 50, 52, 64, 132, 636, 638, 645, 870
— ограничитель, 360
символическая константа, 905, 906, 911
символы разделители, 64, 93, 273, 642, 870, 905
— специальные, 53, 358
—, численное представление, 127
синонимы встроенных типов данных, 852, 853
синтаксическая ошибка, 51, 58, 62, 96, 98, 141
синхронизация istream и ostream, 664
система счисления восьмеричная (с основанием 8), 635, 646, 656, 876, 992, 994, 997
— управления базами данных (СУБД), 741
скаляр, 277, 337
скобки квадратные ([]), 261
— в структуре do/while, 133
— квадратные ([]), 290, 409
— фигурные ({ }), 97, 98, 264, 412, 935
скрытие глобальной переменной, 418
— реализаций, 179, 407, 417
словари, 403, 515, 900
слово, 40, 386, 850, 866

сложение указателя с целым, 342
случайное целое, 820, 823
— число, 189, 191, 808
— —, генерация, 148, 191, 194, 349
— —, рандомизация, 192
создание типа данных, 56, 486, 487
— файла произвольного доступа, 751
сортировка, 280, 281, 338, 489
— блочная, 317, 393
— пузырьковая, 280, 281, 308, 336,
 354, 393, 758
специальные символы, 53, 358
список аргументов переменной
 длины, 922—924
— параметров, 184, 223, 687
сравнение блоков памяти, 872, 881
— массивов, 270, 508
— объединений, 935
— строк, 364
— структур, 850
— указателей, 344
среднее значение, 59, 100, 102, 104,
 105, 282
ссылка, 218, 219, 322, 324, 327, 441,
 517, 600, 631, 798, 852
— на закрытые данные—элементы,
 441, 443
— на константные данные, 335
— на константу, 218
— на локальную переменную, 535
— на объект класса, 502, 508
— на объект, 408, 418, 441, 507, 639
— на объект, который уже не
 существует, 535
— на переменную, 800
— на тип, определенный
 пользователем, 664
стандартная библиотека C, 44, 179,
 181, 324, 358, 361, 530
стандартные блоки, 143, 145
— типы данных, 107, 339
стандартный поток ввода (cin), 46, 55,
 633

— — —, объект, 741
— — — вывода (cout), 46, 633
— — —, объект, 50, 633, 741
стандартный поток ошибок (cerr), 46
— — —, объект, 741
старшинство операций, 58, 62, 64, 96,
 108, 116, 121, 138, 140
статические данные—элементы, 480,
 480, 485, 519, 656, 696
— функции—элементы, 483, 485,
 519, 696
статический класс памяти, 199, 200,
 202
стиль C в динамическом
 распределении памяти, 932
— — — ввода—вывода, 631, 632, 633
строка, 50, 322, 367, 434, 515, 848
— с нулевым завершающим
 символом, 348, 639
строки, библиотека обработки, 361,
 362, 876
—, функции преобразования, 872,
 872
строковая константа, 271, 359, 359
структура выбора switch, 92, 127—
 133, 135, 201, 390
— единственного выбора if, 93, 97
— множественного выбора, 92, 131,
 132, 146
— повторения for, 92, 119—127, 135,
 136, 146, 263, 294
— повторения while, 99, 104, 117,
 118, 120, 146
— с самоадресацией, 408, 850
— следования, 91—92, 103
структурное программирование, 33,
 34, 36, 41, 43, 48, 65, 66, 89, 91,
 135, 407, 705, 933
структуры, 260, 322, 333, 848, 905
структуры выбора, 91, 92, 92
— данных, 260, 348, 408, 485, 602
— — динамические, 260, 322, 782
— — — нелинейные, 796

— повторения, 91, 92, 99, 133, 211
—, описание, 408, 849, 850, 863, 866
стек, 260, 334, 426, 485, 489, 689, 782,
799, 849
— вызовов функций, 334
—, дно, 800
суффиксы констант, 929
сцепление, 504
— двух объектов связных списков,
820
— операций вставить в поток, 56
сцепленные вызовы, 515
— — функций, 476
— операции взять из потока, 62
— присваивания, 517
счетчик, 100, 102, 104, 111
Т
таблица истинности, 137, 138, 139
табулированный формат, 263, 264,
263, 294
тело описания класса, 412
— функции, 50
— цикла, 107, 119, 123, 135
тип, 53, 56, 108
— данных, 53
— — char, 132
— — float, 107
— —, размер, 339
точка с запятой (,), 50, 64, 408, 412,
849
точность по умолчанию, 108, 647
тэг, 408
У
угловые скобки (< и >), 224, 686, 905
удаление, 446, 480, 489, 529, 785, 787,
932
— узлов бинарного дерева, 814
— элементов связного списка, 797
узел корневой, 807, 809
— родительский, 807, 825
узлы, 787, 804
указатели, вычитание, 342, 343
указатель, 263, 278, 322, 327, 787, 852

указатель NULL, 324, 852, 877, 880,
883, 909, 950, 963, 982, 986
— this, 474, 475, 476, 483, 484, 502
— базового класса, 556, 573, 597, 600,
601, 602, 611, 612, 613, 620
— на void (void *), 343
— на абстрактный базовый класс,
601
— на объект, 408, 418
— — — производного класса, 573
—, арифметика, 341, 342, 343, 344,
345, 499
—, индексация, 345
умножение, 36, 57, 58, 59, 62, 486
унарные операции, 107, 140, 324, 339,
501, 506
упаковка значений символов в целое
без знака, 899
управляющая переменная, 116, 117,
120, 121, 123, 134
— структура do/while, 92, 133, 134,
135, 146
— — — двойного выбора if, 67, 94—98,
127, 146
управляющее выражение, 130, 131
управляющие символы, 870
— структуры, 67, 90, 91, 93, 94, 99,
130, 143, 933
— — вложенные, 109, 130, 143, 933
условие, 61, 91, 94, 96, 99, 134
— составное, 137
условная компиляция, 904, 908—910
— операция (?,:), 95, 96, 116, 140, 640,
869
— передача управления, 388
условное выполнение директив
препроцессора, 904
— выражение, 95, 639, 720, 734
условные директивы препроцессора,
909
услуги класса, 423
утечка памяти, 785
— ресурсов, 710

утилиты, 190, 426

Ф

файл, 633, 738, 739, 741, 748, 849, 921, 936

— «сырых» данных, 738

— заголовочный, 108, 189, 190, 191, 324, 361, 420, 420, 424, 445, 633, 721, 905, 926

— последовательного доступа, 742—751

— произвольного доступа, 751—758, 778

—, признак конца, 129, 130, 639, 641, 642, 660, 745, 748, 921, 924

—, режим открывания, 743, 748, 756, 965

—, указатель позиции, 748

файлы на диске, 763, 768

—, ввод—вывод, 635

—, потоки ввода—вывода, 763

фиксированная точка, значение, 126

—, формат, 108

флаги формата, 651—660

формальный параметр, 686, 687

— шаблона функции, 225

формат с фиксированной точкой, 108

—, флаги, 651—660

форматирование, 126, 646

— в памяти, 631

форматированный ввод—вывод, 632, 635, 750

форматы состояния потока, 650

формулирование алгоритма, 100, 102, 109

ФОРТРАН, 43

функции, 41, 50, 66, 146, 179, 181, 407

— "set", 416, 425, 436, 438, 468

— библиотечные, 708, 938

— ввода—вывода, 360

— виртуальные, 574, 596, 597, 601, 603, 607, 613, 614

— встраиваемые inline, 214, 215, 220, 258, 417, 907

— перегруженные, 222, 223, 224, 685, 688

— преобразования строк, 872, 872

— утилиты, 190, 426

—, вложенные вызовы, 710, 712, 714, 734

—, операция вызова (), 530

—, описания, 182, 183, 188, 189, 201, 216, 273, 685, 693

—, перегрузка, 222, 631

функции, прототипы, 125, 183, 185, 188, 189, 190, 201, 214, 216, 328, 331, 338, 416, 432, 472, 597, 905, 927, 926, 938

—, стек вызовов, 334

—, шаблоны, 224, 258, 685—689

функции—элементы, 67, 297, 407, 413, 416, 445, 562

функционализация, 36, 182, 212

функция доступа, 425

Ц

целочисленная арифметика, 499

целочисленное деление, 58, 107

целочисленные данные, 127

цикл, 99, 100, 104, 117, 123, 123, 125, 127

—, управляемый меткой, 107, 134, 388, 831

—, управляемый счетчиком, 111, 134, 388, 831

цифра, 53, 866, 867, 996

Ч

число случайное, 189, 191, 808

чистые виртуальные функции, 599, 601, 602, 607, 614, 618

чтение данных, 52, 601

— файла последовательного доступа, 748

— — произвольного доступа, 756, 758

чувствительность к регистру, 54

III

- шаблон, 224, 684, 783, 787, 800, 804, 805, 807, 845
- класса, 222, 689—694, 695, 696
- функции, 222, 224, 225, 685—689
- шаблоны класса, 689—694, 695, 696, 783, 787, 800, 802, 803, 804, 805, 807, 845
- функций, 224, 258, 685—689
- шестнадцатеричные цифры, 867, 992
- шестнадцатеричное число (по основанию 16), 325, 635, 638, 646, 656, 656, 876, 992, 995
- ширина поля, 864
- ширина поля, 126, 263, 635, 646, 647

Э

экран, 35, 36, 46, 56, 741, 751, 763,
921

экспоненциальная сложность, 211

элемент, 411, 739

— базового класса, 562

элементы, доступ, 413, 425, 472

Я

язык ассемблера, 39, 996

—, чувствительный к регистру, 99

языки процедурного

программирования, 66, 407

—, расширяемость, 208, 270, 413, 487

Предисловие

Добро пожаловать в мир C++! Эта книга создана двумя авторами — один из них в возрасте, другой молодой. Старший (Харви М. Дейтел; Массачусетский Технологический институт, 1967) занимался программированием и преподавал его более 30 лет. Младший (Пол Дж. Дейтел; Массачусетский Технологический институт, 1991) занимался программированием около десяти лет и обнаружил немало дефектов в учебниках и методике преподавания. Старший компаньон программирует и преподает исходя из своего опыта; младший — из неистощимого источника энергии. Старший жаждет четкости; младший — эффективности. Старший ценит элегантность и красоту; младший — результаты. Мы объединились, чтобы создать книгу, которую, как мы надеемся, сочтут информативной, интересной и занимательной.

Обычно в большинстве учебных заведений C++ преподается программистам, уже имеющим опыт работы с С. Многие преподаватели считают, что сложность C++ и ряд других трудностей делают этот язык непригодным для начального обучения программированию — первой задачи этой книги. Так зачем же мы ее писали?

C++ начинает заменять С как один из языков разработки систем в промышленности и есть основания считать, что C++ станет доминирующим языком в середине — конце 90-ых годов. Харви Дейтел преподает вводные курсы программирования в университетах на протяжении двух десятилетий, подчеркивая необходимость разработки ясных и хорошо структурированных программ. Значительное внимание в этих курсах уделялось основным принципам программирования с упором на эффективное использование управляющих структур и функционализацию. Мы представили этот материал точно в таком виде, в котором Харви Дейтел делает это в своих университетских курсах. Есть, конечно, определенные камни преткновения, но там, где они встречаются, мы честно указываем их и приводим процедуры, эффективно их обходящие. Наш опыт говорит, что студенты воспринимают курс по C++ примерно так же, как и вводные курсы по Паскалю или С. Хотя есть одна заметная разница: у студентов очень высока мотивация к изучению передового языка, каким является C++, и передовых принципов программирования (объектно-ориентированного программирования), которые становятся им полезны, как только они покидают стены университета. Это увеличивает их энтузиазм при изучении материала, что очень помогает, учитывая сложность C++ по сравнению с Паскалем или С.

Наша цель была простой: создать учебник по программированию на C++ для вводных университетских курсов, читаемых студентам с малым опытом программирования или вовсе без такового, но тем не менее мы стремились обеспечить глубину и строгость изложения теории и прикладных вопросов, требуемые в традиционных курсах высокого уровня по C++. Чтобы достичь этих целей, мы создали книгу большего объема, чем другие труды по C++, так как наша книга учит принципам как процедурного, так и объектно-ориентированного программирования. Сотни студентов изучили этот материал в наших курсах. Десятки тысяч студентов во всем мире изучали С и знакомились с C++ по нашей близкой к данной теме книге «Как программировать на С», выходящей сейчас вторым изданием.

Введение в объектное ориентирование начинается с главы 1!

Мы столкнулись с серьезной проблемой при создании этой книги. Должна ли книга давать чисто объектно-ориентированный подход? Или же она должна давать смешанный подход, балансируя между процедурным и объектно-ориентированным программированием?

Большинство вузовских профессоров, которые будут учить по этой книге, преподавали процедурное программирование много лет (вероятно, на С или Паскале) и имеют, возможно, некоторый опыт преподавания объектно-ориентированного программирования. С++ сам по себе не является чисто объектно-ориентированным языком. Скорее он является гибридным языком, дающим возможность и процедурного, и объектно-ориентированного программирования.

Так что мы выбрали следующий подход. Первые пять глав книги знакомят с процедурным программированием на С++. Они описывают принципы программирования, управляющие структуры, функции, массивы, указатели и строки. Эти главы освещают компоненты С ANSI в С++ и усовершенствования С, сделанные в С++.

Мы ввели нечто, делающее эти пять глав действительно уникальными. В конце каждой из этих глав имеются специальные разделы, называемые «Размышления об объектах». Эти разделы знакомят с концепциями и терминологией объектной ориентации и призваны помочь студентам понять, что такое объекты и как они себя ведут.

В главе 1 раздел «Размышления об объектах» знакомит с концепциями и терминологией объектной ориентации. Соответствующие разделы глав 2–5 представляют набор требований для создания серьезного проекта объектно-ориентированной системы, а именно — программы моделирования лифта, и проводят студента через типичные этапы процесса объектно-ориентированного проектирования. В этих параграфах рассматривается, как идентифицировать объекты в задаче, как определить атрибуты и функции объекта и как определить взаимодействия объектов. К тому времени, когда студент завершил главу 5, он (или она) уже провел тщательное объектно-ориентированное проектирование модели лифта и готов, если не жаждет, начать программирование лифта на С++.

Главы 6, 7 и 8 освещают абстрагирование данных, классы и перегрузку операций в С++. Эти главы также содержат разделы «Размышления об объектах», которые облегчают студентам продвижение через различные стадии программирования их лифтов на С++.

Теперь приведем детальный обзор остальных частей книги.

Об этой книге

Книга «Как программировать на С++» содержит богатый набор примеров, упражнений и проектов, взятых из различных областей, чтобы дать студенту возможность решать действительно интересные и жизненные задачи. В книге делается упор на принципы хорошего стиля программирования, на ясность программ. Мы избегаем таинственной терминологии и синтаксических определений в пользу обучения на примерах.

Эта книга написана двумя преподавателями, которые большую часть своего времени заняты обучением по актуальным для практики темам в университетских курсах и на семинарах для профессионалов. Книга решает прежде всего задачи обучения. Например, фактически по каждой новой теме как C++, так и объектно-ориентированного программирования, приводится законченная рабочая программа на C++ и тут же показывается результат ее выполнения. Чтение этих программ весьма похоже на их ввод в компьютер с последующим прогоном.

Среди других педагогических приемов книга содержит:

- цели и план каждой главы;
- типичные ошибки программирования, советы по хорошему стилю программирования, советы по повышению эффективности, замечания по мобильности программного обеспечения, замечания по технике программирования — все это перечисляется в каждой главе и суммируется в ее конце;
- алфавитные перечни терминов в каждой главе;
- вопросы для самопроверки и ответы на них в каждой главе;
- наиболее богатое собрание упражнений среди всех учебников по C++.

Упражнения варьируются от простых вопросов на повторение пройденного до серьезных задач программирования и крупных проектов. Преподаватели, ищущие темы для курсовых проектов, найдут много подходящих задач в упражнениях глав 3–18. Мы вложили много труда в эти упражнения, чтобы повысить ценность курса для студентов. Имеется пособие для преподавателя на дискетах в форматах PC и Макинтош с программами, встречающимися в основном тексте, и ответами на почти все вопросы, помещенные в конце глав (к данному изданию эти дискеты не прилагаются — *прим. ред.*).

При написании этой книги мы использовали варианты компиляторов C++, работающие на рабочих станциях Sun SPARCstation, компьютерах Apple Macintosh (Symantech Cи++), IBM PC (Turbo C++, Borland C++, CSET++) фирмы IBM и Microsoft C/C++ версии 7 и Visual C++) и DEC VAX/VMS (DEC C++). Большая часть программ из текста книги будут работать на всех этих компиляторах при незначительной их модификации или вовсе без такой. Мы публикуем версии, разработанные нами на Borland C++.

Этот текст следует предварительному стандарту ANSI на Си++. Если вам нужны дополнительные подробности о языке, смотрите справочные пособия для вашей конкретной системы или достаньте копию «Рабочего документа по предлагаемому предварительному международному стандарту для информационных систем — языку программирования C++» в Американском Национальном Институте Стандартов (ANSI) по адресу: СВЕМА, 1250 Eye Street, NW, Suite 200, Washington DC 20005. Мы использовали различные материалы из этого документа по разрешению специально выданному нам ANSI/СВЕМА. Ниже следует ссылка на это разрешение.

Приведенный здесь материал взят из рабочего документа по предлагаемому предварительному Американскому Национальному Стандарту — языку программирования C++. Работы по его утверждению и техническому развитию ведутся Аккредитованным Комитетом Стандартов X3, Информационной Технологией и ее Техническим комитетом X3J16, Язык Программирования C++, соответственно. Для дальнейших подробностей свяжитесь с Секретариатом X3, 1250 Eye Street, NW, Washington, DC 20 005.

Эта книга имеет ряд особенностей, облегчающих обучение студента.

Цели

Каждая глава начинается с формулировок ее целей. Они говорят студенту, чего он должен ожидать, и позволяют ему после прочтения главы определить, достиг ли он этих целей. Это дает уверенность и является источником закрепления знаний.

План

План главы помогает студенту обозреть весь изучаемый материал. Это также помогает студенту знать, что будет далее, и выбрать удобный и эффективный темп обучения. [Мелочи: книга содержит 2132722 печатных символа и 342980 слов.]

Разделы

Каждая глава состоит из небольших разделов, посвященных ключевым понятиям. Мы предпочитаем строить главы из большого количества небольших разделов.

329 примеров программ (а также результатов выполнения программ и иллюстраций)

Возможности C++ иллюстрируются рядом законченных рабочих программ на C++. Тут же показываются результаты выполнения этих программ. Это позволяет студенту убедиться, что программы работают, как ожидалось. Сопоставление результатов с операторами программы, дающими эти результаты, является отличным путем изучения и укрепления знаний. Наши программы демонстрируют различные особенности C++. Внимательное чтение книги во многом похоже на набор и прогон этих программ на компьютере.

Иллюстрации

В книгу включено множество диаграмм и рисунков. Обсуждение управляющих структур сопровождается тщательно составленными блок-схемами. (Замечание: мы не учим использованию блок-схем как инструменту разработки программ, но мы используем сжатое представление в виде блок-схем, чтобы точно описать действие управляющих структур C++). Глава 15, «Структуры данных», использует рисунки для иллюстрации процессов создания и поддержания связных списков, очередей, стеков и бинарных деревьев.

Полезные советы по разработке программ

Чтобы помочь студентам сконцентрироваться на важных аспектах разработки программ, их тестирования и отладки, эффективности и мобильности, мы включили в книгу сотни советов, разбив их на категории: «Хороший стиль программирования», «Типичные ошибки программирования», «Советы по повышению эффективности», «Замечания по мобильности» и «Замечания по технике программирования». Эти советы представляют то лучшее, что мы смогли собрать за четыре десятилетия программирования и преподавания. Одна из наших студенток, профiliрующаяся по математике, сказала нам недавно, что этот подход напоминает ей то, как выделяются аксиомы, теоремы и следствия в трудах по математике; он обеспечивает базу для построения хорошего программного обеспечения.

111 советов по хорошему стилю программирования

В тексте выделены советы по хорошему стилю программирования. Они привлекают внимание студента к методикам, помогающим создавать хорошие программы. Когда мы читаем вводные курсы не программистам, то объявляем путеводной звездой курса «ясность программ» и говорим студентам, что будем выделять методики, позволяющие писать программы более прозрачные, понятные, легкие в отладке и сопровождении.

171 типичная ошибка программирования

Студенты, изучающие язык (особенно в своем первом курсе программирования) склонны часто делать совершенно определенные ошибки. Концентрируя внимание студента на этих типичных ошибках программирования книга помогает ему избежать их. Это также помогает уменьшить длинные очереди у кабинетов преподавателей в их приемные часы!

49 советов по повышению эффективности

По нашему опыту, научить студентов написанию ясных, понятных программ — это едва ли не наиболее важная задача первого курса программирования. Но студенты хотят писать программы, которые быстрее всех считают, используют меньше всех памяти, требуют наименьшего числа нажатий клавиш и блещут в других отношениях. Студенты серьезно озабочены эффективностью. Они хотят знать, что можно сделать, чтобы придать программам свойства «турбо». Так что мы включили в книгу советы по повышению эффективности, чтобы указать возможности улучшения программ.

28 замечаний по мобильности

Разработка программного обеспечения — сложное и дорогое занятие. Организации, разрабатывающие его, часто должны производить версии, при способленные к различным компьютерам и операционным системам. Так что сегодня делается большой упор на мобильность, то есть на способность программного обеспечения работать на разнообразных компьютерных системах при незначительном его изменении или даже вообще без изменений. Многие рекламируют C++ как язык, подходящий для разработки мобильного программного обеспечения, основываясь на том, что C++ тесно связан с C ANSI, и на том, что скоро появится стандартная версия C++ ANSI. Некоторые считают, что если они разработали прикладную программу на C++, то она автоматически будет мобильной. Это просто не соответствует действительности. Достижение мобильности требует аккуратного и осторожного проектирования. На этом пути есть много «подводных камней». Мы включили в книгу много замечаний по мобильности, чтобы помочь студентам писать мобильные программы.

106 замечаний по технике программирования

Объектно-ориентированное программирование требует полного переосмысления способов, которыми мы создаем системы программного обеспечения. C++ — это эффективный язык для разработки хорошего программного обеспечения. Замечания по технике программирования обращают внимание на методики, вопросы архитектуры и конструирования программного обеспечения, особенно больших систем. Многое из того, что студент здесь изучает, будет полезно в курсах более высокого уровня и в промышленности, когда студент начнет работать с большими, сложными реальными системами.

Резюме

Каждая глава заканчивается дополнительным педагогическим приемом — резюме. Мы представляем в виде списка основные итоги главы. Это помогает студентам просмотреть и закрепить ключевые вопросы данной главы.

Терминология

Мы включаем в каждую главу раздел «Терминология» с алфавитным списком важных терминов, определения которых даны в главе, для их дальнейшего закрепления.

Обзор советов, замечаний и ошибок

В конце каждой главы мы суммируем все приведенные в ней «Советы по хорошему стилю программирования», «Типичные ошибки программирования», «Советы по повышению эффективности», «Замечания по мобильности» и «Замечания по технике программирования».

525 заданий для самопроверки и ответов на них (при счете учтены отдельные части заданий)

Задания для самопроверки и ответы на них включены в книгу для целей самообучения. Они дают возможность студенту обрести уверенность в знании материала и подготовиться к основным упражнениям.

763 упражнения (решения в пособии преподавателя; при счете учтены отдельные части заданий)

Каждая глава завершается большим набором упражнений, включающих:

- простое напоминание важных терминов и принципов;
- написание отдельных операторов на С++;
- написание небольших функций и классов на С++;
- написание законченных функций, классов и программ на С++;
- написание крупных курсовых проектов.

Большое количество упражнений позволяет преподавателям приспосабливать свои курсы к потребностям конкретной аудитории и варьировать курсовые задания каждый семестр. Преподаватели могут использовать эти упражнения для составления домашних заданий, кратких опросов и проведения экзаменов.

1353 слова в предметном указателе (многие со ссылками на несколько страниц, каждое)

Мы включили в конце книги пространный предметный указатель. Это помогает студенту найти любой термин или понятие по ключевому слову. Указатель полезен читающим эту книгу впервые, но особенно — программистам-практикам, использующим ее в качестве справочника. Мы сделали так, чтобы каждый термин из разделов «Терминология» присутствовал в указателе (наряду со многими другими словами из глав). Таким образом, студент может использовать указатель совместно с разделами «Терминология», чтобы убедиться в том, что он охватил ключевой материал каждой главы.

Обзор книги

Книга разделена на несколько крупных частей. Первая часть — главы с 1 по 5, представляет собой детальное изложение процедурного программирования на C++, включая типы данных, ввод-вывод, управляющие структуры, функции, массивы, указатели и строки.

Вторая часть — главы с 6 по 8, обстоятельно рассматривает абстрагирование данных, классы, объекты и перегрузку операций. Этую часть можно по праву назвать «Программирование с объектами».

Третья часть — главы 9 и 10, излагает наследование, виртуальные функции и полиморфизм, т.е. основы истинного объектно-ориентированного программирования.

Следующая часть — главы 11 и 14, описывает ввод-вывод, ориентированный на потоки в стиле C++, включая потоки ввода-вывода клавиатуры, экрана, файлов и массивов символов; обсуждается обработка файлов как последовательного, так и произвольного доступа.

Следующая часть — главы 12 и 13, рассматривает два недавних крупных добавления в C++, а именно, шаблоны и обработку исключений. Шаблоны, называемые также параметризованными типами, способствуют повторному использованию программного обеспечения. Исключения позволяют программистам разрабатывать более надежные и устойчивые к ошибкам системы.

Следующая часть — глава 15, подробно излагает динамические структуры данных, такие, как связные списки, очереди, стеки и деревья.

Последняя часть основного текста — главы с 16 по 18, обсуждает различные темы, включая манипуляции с битами, символами и строками, пре-процессор, а также разнообразные «Другие темы».

Последнюю часть книги составляют справочные материалы, подкрепляющие основной текст, включая библиотеку стандартных функций, старшинство операций, набор символов ASCII, системы счисления (двоичную, десятичную, восьмеричную, шестнадцатеричную). Текст завершается подробным предметным указателем, помогающим читателю найти в тексте любой термин по ключевому слову.

Теперь давайте рассмотрим каждую главу в отдельности.

Глава 1, «Введение в C++», объясняет, что такое компьютер, как он работает и программируется. Глава знакомит с понятиями структурного программирования и объясняет, почему этот набор методик произвел революцию в разработке программ. В главе дается краткая история развития языков программирования от машинных до языков ассемблера и языков высокого уровня. Рассматривается происхождение языка C++. Глава включает знакомство с типичной средой программирования на C++ и дает сжатое введение в технику написания программ на C++. Приводится подробное рассмотрение принятия решений и арифметических операций, представленных в C++. После изучения этой главы студент станет понимать, как писать простые, но законченные программы на C++.

Глава 2, «Управляющие структуры», знакомит с понятием алгоритма решения задачи. Объясняется важность эффективного использования управляющих структур в создании программ, которые понятны, легко отлаживаются, поддерживаются и с большой вероятностью работают с первой попытки. Глава знакомит со структурами следования, выбора (*if*, *if/else* и *switch*) и

повторения (`while`, `do/while` и `for`). В ней подробно исследуется повторение и сравниваются варианты циклов, управляемых счетчиком и меткой. Глава объясняет методику исходящей пошаговой детализации, которая является ключевой для создания хорошо структурированных программ, и представляет популярное средство построения программ — псевдокод. Методы и подходы, используемые в главе 2, способствуют эффективному применению управляющих структур в любом языке программирования, а не только в C++. Эта глава помогает студенту выработать навыки качественного программирования в преддверии более серьезных задач, с которыми он встретится далее. Глава завершается рассмотрением логических операций `&&` (И), `||` (ИЛИ) и `!` (НЕ).

Глава 3, «Функции», посвящена проектированию и построению программных модулей. Возможности C++, относящиеся к функциям, включают функции стандартной библиотеки, функции, определяемые программистом, рекурсию, вызовы по значению и по ссылке. Методики, представленные в главе 3, существенны для построения правильно структурированных программ, особенно тех больших программ, которые будущие системные и прикладные программисты будут по все вероятности разрабатывать в реальной жизни. Рассматривается стратегия «разделяй и властвуй» как эффективное средство решения сложных проблем путем разделения их на более простые взаимодействующие компоненты. Студенты с удовольствием изучают случайные числа и моделирование, они ценят рассмотрение азартных игр, элегантно использующее управляющие структуры. Глава предлагает основательное введение в рекурсию и содержит таблицу, в которой перечислены десятки примеров и упражнений по рекурсии, содержащихся в остальной части книги. Некоторые учебники рассматривают рекурсию лишь в последней главе; мы же считаем, что эту тему лучше раскрывать постепенно на протяжении всего курса. Обширный набор из 60 упражнений в конце главы включает несколько классических рекурсивных задач, как, например, задача о Башнях Ханоя. Глава рассматривает так называемые «расширения C++ языка С», включая встраиваемые функции, ссылочные параметры, аргументы по умолчанию, унарную операцию разрешения области действия, перегрузку функций и шаблоны функций.

Глава 4, «Массивы», описывает структурирование данных в массивы или группы связанных элементов одного типа. В главе приводятся многочисленные примеры одномерных и двумерных массивов. Общепризнанно, что правильное структурирование данных столь же важно в разработке хорошо структурированных программ, как и эффективное использование управляющих структур. Примеры, приведенные в главе, посвящены распространенным операциям с массивами, печати гистограмм, сортировке данных, передаче массивов функциям и введению в область анализа данных обследований (с простой статистикой). Особенностью этой главы является рассмотрение элементарных методик сортировки и поиска, представление двоичного поиска как радикальной альтернативы линейного поиска. 38 упражнений в конце главы включают набор интересных и перспективных задач, таких, как улучшенные методики сортировки, проектирование системы бронирования билетов на авиалинии, введение в основы построения траектории черепахи (пропавшей в языке LOGO), задачи Путешествие коня и Восемь ферзей, которые знакомят с понятиями эвристического программирования, широко используемого в области искусственного интеллекта. Упражнения завершаются восемью рекурсивными задачами, включая сортировку отбором, палиндромы, линейный поиск, двоичный поиск, задачу Восьми ферзей, распечатку

массива, обратную распечатку строки и нахождение минимального значения в массиве.

Глава 5, «Указатели и строки», описывает одно из наиболее мощных средств языка C++. Глава дает детальное объяснение операций с указателями, вызова по ссылке, выражений с указателями, арифметики указателей, связи указателей с массивами, массивов указателей и указателей функций. В C++ существует тесная связь между указателями, массивами и строками; поэтому мы знакомим с основами работы со строками и обсуждаем наиболее часто применяемые при этом функции, а именно, `getlin` (ввод строки текста), `strcpuy` и `strncpy` (копирование строки), `strcat` и `strncat` (сплление двух строк), `strcmp` и `strncmp` (сравнение двух строк), `strtok` (разбиение строки на лексемы) и `strlen` (вычисление длины строки). 49 упражнений этой главы включают классическую гонку черепахи и зайца, алгоритмы тасования и раздачи карт, рекурсивную быструю сортировку и рекурсивное блуждание по лабиринту. Включен также специальный раздел, озаглавленный «Построение вашего собственного компьютера». Этот раздел объясняет принципы программирования на машинном языке и предлагает спроектировать и реализовать моделирующую программу, позволяющую читателю писать и выполнять программы на машинном языке. Это уникальное задание будет особенно полезно читателю, который хочет понять, как в действительности работают компьютеры. Нашим студентам этот проект очень нравится и они часто предлагают существенные усовершенствования его; многие из этих усовершенствований мы переадресовали читателю в упражнениях. В главе 15 другой аналогичный специальный раздел помогает читателю построить свой компилятор; программа на машинном языке, построенная этим компилятором, может затем выполняться с помощью моделирующей программы, созданной в главе 5. Информация от компилятора к моделирующей программе передается через файл последовательного доступа (см. главу 14). Второй специальный раздел главы 5 включает упражнения на операции со строками, связанные с анализом текста, обработкой слов, печатью дат в различных форматах, защите чеков, написанию словесного эквивалента суммы чека, азбуке Морзе и переводу метрических мер в английскую систему мер.

Глава 6, «Классы и абстрагирование данных», начинает серьезное рассмотрение объектов. Глава предоставляет прекрасную возможность обучения «правильному» абстрагированию данных с помощью языка (C++), специально предназначенному для реализации абстрактных типов данных (АТД). В последние годы абстрагирование данных стало важной темой вводных компьютерных курсов. Главы 6, 7 и 8 содержат цельное изложение абстрагирования данных. В главе 6 рассматривается реализация АТД в виде структур `struct`, в виде классов в стиле C++, доступ к элементам классов, разделение интерфейса и реализации, использование функций доступа и функций-утилит, инициализация объектов конструкторами, уничтожение объектов деструкторами, присваивание по умолчанию побитовым копированием и повторное использование программного обеспечения. Упражнения главы предлагают студентам разработку классов комплексных чисел, рациональных чисел, времени, дат, прямоугольников, больших целых чисел и для игры в тик-так-тоу. Студентам обычно нравятся игровые программы.

Глава 7, «Классы, часть II», продолжает изучение классов и абстрагирования данных. В главе рассматриваются объявление и использование константных объектов, константные функции-элементы, композиция — процесс создания классов, содержащих в качестве элементов объекты других классов,

дружественные функции и дружественные классы, которые имеют особое право доступа к закрытым и защищенным элементам класса, указатель `this`, позволяющий объекту знать собственный адрес, динамическое распределение памяти, статические элементы класса для хранения и операций с данными, имеющими одну копию для всех объектов класса, примеры распространенных абстрактных типов данных (массивы, строки, очереди), классы контейнеры и итераторы. Упражнения главы предлагают студенту разработать класс счетов, связанных с хранением вкладов, и класс множества целых чисел.

Глава 8, «Перегрузка операций», является одной из самых интересных в нашем курсе по C++. Студентам этот материал действительно нравится. Они находят его прекрасно соглашающимся с изложением абстрактных типов данных в главах 6 и 7. Перегрузка операций позволяет программисту указать компилятору, как применять существующие операции к объектам новых типов. C++ сам по себе знает, как применять операции только к объектам таких встроенных типов, как целые числа, числа с плавающей запятой и символы. Но, предположим, мы создали новый класс — строки. Что означает применительно к строкам знак плюс? Многие программисты используют плюс применительно к строкам для обозначения их сплеления. Из главы 8 программист узнает, как перегрузить знак плюс таким образом, чтобы при его написании в выражении между двумя строковыми объектами компилятор генерировал бы вызов функции-операции, которая сплела бы эти две строки. В главе рассматриваются основы перегрузки операций, ограничения при перегрузке операций, различия в перегрузке функций-элементов класса и функций, не являющихся элементами, перегрузка унарных и бинарных операций, преобразование типов. Особенностью главы является обстоятельное изучение ряда важных частных случаев, включая класс массивов, класс строк, класс дат, класс больших целых чисел и класс комплексных чисел (для двух последних классов в упражнениях приводится полный исходный текст). Математически подготовленный студент получит удовольствие, создавая в ходе упражнения класс полиномов.

Глава 9, «Наследование», посвящена одной из основополагающих особенностей объектно-ориентированных языков программирования. Наследование — это форма повторного использования кода, которая позволяет быстро и легко разрабатывать новые классы путем использования возможностей уже существующих классов и добавления к ним каких-то новых возможностей. В главе вводятся понятия базовых классов и производных классов, защищенных элементов, открытого, защищенного и закрытого наследования, прямых и косвенных базовых классов, рассматривается применение конструкторов и деструкторов в базовых и производных классах, методика программирования с использованием наследования. В главе сравниваются наследование (отношение «является ...») с композицией (отношение «содержит ...») и вводятся отношения типа «использует ...» и «знает ...». Особенностью главы является изучение нескольких важных частных случаев. В частности, развернутый учебный пример реализует иерархию классов точка, круг, цилиндр. Глава завершается учебным примером множественного наследования — особенности развитого C++, позволяющей производному классу наследовать атрибуты и функции нескольких базовых классов. Упражнения предлагают студенту сравнить создание новых классов наследованием и композицией; расширить иерархии наследования, рассмотренные в главе; создать иерархию наследования четырехугольник, трапеция, параллелограмм, прямоугольник, квадрат; создать более общую иерархию двумерных и трехмерных форм.

Глава 10, «Виртуальные функции и полиморфизм», рассматривает другую фундаментальную особенность объектно-ориентированного программирования — полиморфное поведение. Когда много классов связаны наследованием с одним базовым классом, каждый объект производного класса может рассматриваться как объект базового класса. Это позволяет писать программы в общем виде независимо от специфики типов объектов производных классов. Новые типы объектов тоже могут обрабатываться той же самой программой, что способствует расширяемости систем. Полиморфизм позволяет заменить в программах сложную логику переключений структуры `switch` более простой «линейной» логикой. Например, экранный администратор в видео-игре может просто посылать сообщение «нарисуй» каждому объекту связного списка, который должен быть нарисован. А каждый объект знает, как себя нарисовать на экране. К программе без ее модификации может быть добавлен новый объект, если он тоже знает, как себя нарисовать. Такой стиль программирования типичен для реализации популярных сегодня графических интерфейсов пользователя. В главе рассматривается механизм достижения полиморфного поведения путем использования виртуальных функций. Глава проводит различие между абстрактными классами, объекты которых не могут быть созданы, и конкретными классами, которые используются для создания объектов. Абстрактные классы используются для предоставления интерфейса, который наследуется всеми классами иерархии. Особенностью главы являются два больших учебных примера полиморфизма: система расчета зарплаты и еще одна версия рассмотренной в главе 9 иерархии форм точка, круг, цилиндр. Упражнения главы предлагают студенту обсудить ряд концептуальных проблем и подходов, добавить абстрактные классы в иерархию форм, разработать эскиз графического пакета, изменить приведенный в главе класс служащих; причем все эти проекты выполняются с помощью виртуальных функций и полиморфного программирования.

Глава 11, «Потоки ввода-вывода в C++», содержит всестороннее рассмотрение нового объектно-ориентированного стиля ввода-вывода в C++. Глава обсуждает различные возможности ввода-вывода C++, включая вывод операцией поместить в поток, ввод операцией взять из потока, сохранение типов при вводе-выводе (хорошее усовершенствование по сравнению с C), форматированный ввод-вывод, неформатированный ввод-вывод (более эффективный), манипуляторы потока для управления основанием счисления потока (десятичное, восьмеричное или шестнадцатеричное), числами с плавающей запятой, управление шириной поля, манипуляторы определяемые пользователем, состояния формата потока, состояния ошибки потока, ввод-вывод объектов определяемого пользователем типа и связь выходных потоков со входными (чтобы гарантировать, что приглашение появится на экране перед пользователем действительно прежде, чем он должен будет вводить ответы). Обширный набор упражнений предлагает студенту написать различные программы, которые затрагивают фактически все возможности ввода-вывода, обсужденные в тексте.

Глава 12, «Шаблоны», обсуждает одно из наиболее современных добавлений к развивающемуся языку C++. Шаблоны функций рассматривались ранее в главе 3. Глава 12 дает дополнительный пример шаблона функции. Шаблоны класса дают возможность программисту заимствовать наиболее существенное из абстрактных типов данных (например, стека, массива или очереди) и затем создавать с минимальным дополнительным программированием варианты этих АТД для более конкретных типов (например, очереди

целых чисел `int`, очереди чисел с плавающей запятой `float`, очереди строк и т.д.). По этой причине шаблоны классов часто называют параметризованными типами. Глава обсуждает использование типизированных и нетипизированных параметров, а также взаимодействие между шаблонами и такими понятиями C++, как наследование, друзья и статические элементы. Упражнения предлагают студенту написать ряд шаблонов функций и классов и применить их в полноценных, завершенных программах.

Глава 13, «Обработка исключений», обсуждает последнее существенное добавление к языку C++. Обработка исключений дает возможность программисту писать программы более живучие, более отказоустойчивые и более приспособленные для выполнения ответственных задач. Глава обсуждает, когда обработка исключений уместна, а когда нет; дает основы обработки исключений блоками `try`, операторами `throw` и блоками `catch`; показывает, как и когда целесообразно генерировать исключение повторно; объясняет, как писать спецификации исключений и обрабатывать непредусмотренные исключения; обсуждает важные связи между исключениями, конструкторами, деструкторами и наследованием. Особенности главы — 43 упражнения, которые проводят студента через реализацию программ, иллюстрирующих многообразие и мощность возможностей C++ по обработке исключений.

Глава 14, «Обработка файлов и ввод-вывод потоков строк», обсуждает методы, используемые при обработке текстовых файлов с последовательным и произвольным доступом. Глава начинается с введения в иерархию данных: от битов к байтам, полям, записям и файлам. Затем представлен обзор файлов и потоков в C++. Файлы последовательного доступа обсуждаются на примере трех программ, которые показывают, как открывать и закрывать файлы, как последовательно сохранять данные в файле, и как последовательно читать данные из файла. Файлы произвольного доступа обсуждаются на примере четырех программ, которые показывают, как создать файл произвольного доступа, как осуществлять произвольное чтение и запись такого файла и как последовательно читать данные из файла произвольного доступа. Четвертая программа, объединяет методы произвольного и последовательного доступа при реализации законченного приложения диалоговой обработки запросов. Слушатели наших семинаров для промышленности говорили нам, что после изучения материала по обработке файлов они были способны составить серьезные программы обработки файлов, которые были немедленно использованы в их организациях. Глава обсуждает также возможности C++ по вводу данных из символьных массивов в памяти и выводу данных в символьные массивы в памяти; эти возможности часто называются форматированием в памяти или обработкой потоков строк. Упражнения предлагают студентам реализовать ряд программ, которые формируют и обрабатывают как файлы последовательного доступа, так и файлы произвольного доступа.

Глава 15, «Структуры данных», обсуждает методы, используемые для создания и управления динамическими структурами данных. Глава начинается с обсуждения классов с самоадресацией и динамического распределения памяти. Далее в главе обсуждается, как создавать и поддерживать различные динамические структуры данных, включая связные списки, очереди, стеки и деревья. Для каждого типа структур данных мы даем полные рабочие программы и показываем типичные выводы результатов их работы. Глава реально помогает студентам овладеть указателями. Она включает множество примеров, использующих разименование и двойное разименование — понятия, особо трудные для усвоения. Одна из проблем при работе с указателями

состоит в том, что студентам трудно визуально представить себе структуры данных и связь их узлов. Поэтому мы включили в главу иллюстрации, которые показывают связи и последовательность, в которой они создаются. Пример двоичного дерева — превосходная основа для изучения указателей и динамических структур данных. В этом примере создается двоичное дерево, убираются дубликаты и рекурсивно осуществляются три вида обхода дерева: последовательный, обход в ширину и обратный. Студенты испытывают подлинное чувство завершенности, когда они изучают и реализуют этот пример. Они особенно рады увидеть, как последовательный обход печатает значения в узлах в упорядоченной последовательности. Глава включает множество упражнений. Исключительной особенностью главы является специальный раздел «Создание вашего собственного компилятора». Упражнения проводят студента через разработку программы преобразования инфиксной формы записи выражений в постфиксную форму и программы вычисления выражения в постфиксной форме. Затем мы изменяем алгоритм постфиксного вычисления так, чтобы он генерировал коды машинного языка. Транслятор помещает этот код в файл, используя методы, рассмотренные в главе 14. Затем студенты передают файл программы на машинном языке, созданный их транслятором, в программу моделирования компьютера, построенную ими в упражнениях главы 5. 35 упражнений данной главы включают моделирование очереди в супермаркете, рекурсивный поиск в списке, рекурсивную печать списка в обратной последовательности, удаление узлов двоичного дерева, послойный обход бинарного дерева, печать деревьев, написание фрагментов оптимизирующего компилятора, написание интерпретатора, вставка и удаление произвольного узла связного списка, реализацию списков и очередей без указателей на их конец, анализ эффективности поиска и сортировки на двоичном дереве и построение класса индексированных списков.

Глава 16, «Биты, символы, строки и структуры», представляет ряд важных особенностей C++. Мощные средства манипулирования битами в C++ позволяют программистам писать программы, которые эффективно используют аппаратные возможности низшего уровня. Эти полезные программы обрабатывают поля битов, устанавливают и сбрасывают отдельные биты и хранят информацию более компактно. Такие возможности, обычно присущие только языкам ассемблера низкого уровня, оценены программистами, пишущими системное программное обеспечение типа операционных систем и программного обеспечения работы с сетями. Как вы помните, мы ознакомили студента с манипулированием строками в главе 5 и представили там наиболее популярные функции обработки строк. В главе 16 мы продолжаем наше рассмотрение символов и строк. Мы рассказываем о различных возможностях обработки символов, предоставляемых библиотекой *ctype*; они включают возможность проверить, является ли символ цифрой, буквой, алфавитно-цифровым символом, шестнадцатеричной цифрой, символом в нижнем регистре, в верхнем регистре и т.д. Мы представляем функции работы со строками различных библиотек. Как всегда, для каждой функции приводится использующая ее полная рабочая программа на C++. Структуры в C++ подобны записям в Паскале и других языках программирования; они группируют элементы данных различных типов. Структуры используются в главе 14 при формировании файлов, состоящих из записей информации. Структуры вместе с указателями и динамическим распределением памяти используются в главе 15 при формировании динамических структур данных типа связных списков, очередей, стеков, и деревьев. Особенностью данной главы 16 явля-

ется пересмотренное, более эффективное, чем рассматривалось ранее, моделирование тасования и раздачи карт. Это превосходная возможность для преподавателя подчеркнуть важность стремления к созданию высококачественных алгоритмов. 35 упражнений вдохновляют студента испытать большинство функциональных возможностей, обсужденных в данной главе. Одно из характерных упражнений стимулирует студента к разработке программы проверки орфографии.

Глава 17, «Препроцессор», предлагает детальное обсуждение директив препроцессора. Глава включает более полную информацию о директиве `#include`, которая перед компиляцией копирует указанный файл на место этой директивы, и о директиве `#define`, которая создает символические константы и макросы. Глава объясняет условную трансляцию, предоставляющую программисту возможность управлять выполнением директив препроцессора и компиляции кода программы. Обсуждаются также операция `#`, которая преобразовывает операнд, являющийся макросом, в строку с кавычками, и операция `##`, который склеивает два макроса. Представлены пять предопределенных символьических констант (`_LINE_`, `_FILE_`, `_DATE_`, `_TIME_` и `_STDC_`). В заключение обсуждается макрос `assert` заголовочного файла `assert.h`; макрос `assert` полезен при тестировании и отладке программ.

Глава 18, «Другие темы», рассматривает некоторые дополнительные вопросы, включая несколько сложных тем, обычно не рассматриваемых во вводных курсах. Мы показываем, как переназначить ввод в программу на ввод из файла, переназначить вывод из программы на вывод в файл, переназначить вывод из одной программы на ввод в другую программу (конвейерная пересылка), как добавлять вывод программы в конец существующего файла, как разработать функции, которые используют списки параметра переменной длины, как можно передать в функцию `main` и использовать в ней параметры командной строки, как компилировать программу, состоящую из множества файлов, как регистрировать функции, которые будут выполняться при завершении программы, с помощью `atexit` и `exit`, как использовать спецификаторы типа `const` и `volatile`, как определять тип числовой константы, используя суффиксы целых чисел и чисел с плавающей запятой, как использовать библиотеку обработки сигналов для перехвата непредвиденных событий, как создавать и использовать динамические массивы с помощью `calloc` и `realloc`, как использовать объединения для экономии памяти и как использовать спецификации редактирования при связывании программ на C++ с кодами, унаследованными от С.

Несколько приложений дают читателю ценный справочный материал. В приложении А мы даем обзор с пояснениями всех стандартных библиотечных функций; в приложении Б даем таблицу приоритетов и ассоциативности операций; приложение В содержит набор кодов ASCII для символов; в приложении Г обсуждаются двоичные, восьмеричные, десятичные и шестнадцатеричные системы счисления. Книга содержит обширный предметный указатель, позволяющий читателю найти в тексте любой термин или понятие по ключевым словам.

Приложение А и различные рисунки были отобраны из стандартного документа ANSI/ISO по письменному разрешению Американского Национального Института Стандартов; это приложение — детализированный и ценный источник информации для практикующего программиста. Мы искренне благодарим за сотрудничество и за помощь нам в получении необходимого разрешения на эту публикацию Роберта Хагера, исполняющего

обязанности директора ANSI по публикациям. Разрешение на этот материал приведено ниже:

Различные рисунки и приложение А: Стандартная библиотека были сжаты и адаптированы с разрешения из Американского Национального Стандарта для Информационных систем — язык программирования C, ANSI/ISO 9899: 1990. Копии этого стандарта могут быть приобретены в Американском Национальном Институте Стандартов по адресу: West 42nd Street, New York, NY 10036.

Благодарности

Одно из наибольших удовольствий при написании учебника состоит в возможности принести благодарность за усилия многих людей, чьи имена не могут появиться на обложке, но без чьей упорной работы, сотрудничества, дружбы и понимания написание этого текста было бы невозможно.

Х. М. Дейтел хочет поблагодарить своих коллег из Nova University Энда Леблейна, Энда Симко, Кловис Тондо, Фила Адамса, Рэйса Сабо, Рауля Салазара, Брайана Уэллетта и Барбару Эдж.

Мы хотели бы поблагодарить наших друзей из Digital Equipment Corporation (Стефанию Стосур Шварц, Сью-Лайн Гарретт, Бетти Миллс, Джени Конноли, Барбари Кутюре и Пола Сандора, из Sun Microsystems (Гарри Морин и Карин Шо), из Informative Stages (Дона Холла и Дэвида Литвака), из Semaphore Training (Клива Ли), из Cambridge Technology Partners (Карта Дэвиса, Пола Шермана и Вильберто Мартинеса) и многих других наших общих коллег, которые сделали преподавание этого материала в производственной обстановке такой радостью.

Мы были счастливы возможностью работать над этим проектом с талантливой и просвещенной группой издательских профессионалов в Prentice Hall. Эта книга появилась благодаря поддержке, энтузиазму и упорству нашего издателя Алана Апта и главного редактора Марсия Хортонса.

Сондра Чавес координировала усилия рецензентов этой рукописи и была всегда невероятно полезна, когда мы нуждались в помощи; ее кипучая энергия и поддержка искренне оценены нами. Джой Скордато провел замечательную работу как организатор проекта. Ширли Макгуайр была превосходным помощником редактора.

Мы ценим усилия наших рецензентов:

Тима Борна (AT&T, Bell Labs)

Тони Л. Хансена (AT&T Bell Labs; Хансен — автор книги «*The C++ Answer Book*», Addison-Wesley 1990, которая содержит ответы на упражнения и широко используется вместе с «*The C++ Programming Language*» — классической книгой по C++ Бъярне Строуструпа)

Кловиса Тондо (T & T TechWork, Inc. и Nova Southeastern University; Тондо — соавтор книги «*The C Answer Book*», Prentice-Hall, 1989, которая содержит ответы на упражнения и широко используется вместе с «*The C Programming Language*» — классическим трудом Брайана Кернигана и Дениса Ритчи)

Р. Рейда (Michigan State University)

Джина Спаффорда (Purdue University)

Х.Е. Дунсмора (Purdue University)

Дэвида Фальконе (California State University at Fullerton)

Дэвида Финкеля (Worcester Polytechnic)

Кеннета А. Рика (Rochester Institute of Technology)

Кена Арнольда (Wildfire Communications)

Дона Костуча (You Can C Clearly Now)

Роберта Г. Плантса (Sonoma State University)

Все они тщательно исследовали весь текст и сделали бесчисленное множество ценных предложений для улучшения точности и законченности изложения.

Авторы хотели бы выразить специальную благодарность Эдду Либлейну, одному из всемирно известных авторитетов по методологии программного обеспечения за его конструктивные комментарии и критику при расширении материала по C++ и ООП.

Тэм Нието пожертвовал долгие часы упорного труда, помогая нам сформировать специальный раздел «Постройте ваш собственный компилятор» в конце главы 15. Нието также работает с нами над «Руководством для преподавателя» для этой книги.

И последнее, но, конечно, не по значимости: мы хотели бы поблагодарить Барбару и Абби Дейтел за их любовь и понимание и за их огромную помощь в подготовке рукописи. Они затратили бесконечные часы, проверяя каждую программу в тексте, помогая в каждой фазе подготовки рукописи и вычитывая каждый вариант текста вплоть до публикации. Их острое зрение предотвратило неисчислимые ошибки в рукописи.

Мы будем высоко ценить любые комментарии наших читателей, критику, исправления и предложения по улучшению текста. Пожалуйста, шлите нам ваши предложения по улучшению и расширению наших списков *Типичные ошибки программирования*, *Хороший стиль программирования*, *Советы по повышению эффективности*, *Замечания по мобильности*, *Замечания по технике программирования*. Мы выражаем признательность всем корреспондентам в следующем издании нашей книги. Пожалуйста адресуйте всю почту на наш адрес e-mail:

deitel@world.std.com

или пишите нам:

Harvey M. Deitel (автор)

Paul J. Deitel (автор)

c/o Computer Science Editor

College Book Editorial

Prentice Hall

Englewood Cliffs, New Jersey 07632

Мы ответим немедленно.

Харви М. Дейтел

Пол Дж. Дейтел

1

Введение в компьютеры и программирование на C++



Ц е л и

- Понять основные концепции вычислительной техники.
- Близко познакомиться с различными типами языков программирования.
- Понять среду разработки программ C++.
- Научиться писать простые программы на C++.
- Научиться использовать простые операторы ввода и вывода.
- Освоить основные типы данных.
- Научиться использовать арифметические операции.
- Понять приоритеты арифметических операций.
- Научиться писать простые операторы управления.

П л а н

- 1.1. Введение
- 1.2. Что такое компьютер?
- 1.3. Организация компьютера
- 1.4. Эволюция операционных систем
- 1.5. Персональные вычисления, распределенные вычисления и вычисления на платформе клиент/сервер
- 1.6. Машинные языки, языки ассемблера и языки высокого уровня
- 1.7. История C++
- 1.8. Библиотеки классов C++ и стандартная библиотека C
- 1.9. Параллельный C++
- 1.10. Другие языки высокого уровня
- 1.11. Структурное программирование
- 1.12. Общее описание типичной среды программирования на C++
- 1.13. Общие замечания о C++ и этой книге
- 1.14. Введение в программирование на C++
- 1.15. Простая программа: печать строки текста
- 1.16. Другая простая программа: сложение двух целых чисел
- 1.17. Концепции памяти
- 1.18. Арифметика
- 1.19. Принятие решений: операции проверки на равенство и отношения
- 1.20. Размышления об объектах

Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

1.1. Введение

Добро пожаловать в C++! Мы немало поработали над созданием для вас книги, которая, как мы надеемся, достаточно информативна, занимательна и поучительна. C++ — трудный язык, который может быть глубоко изучен только опытными программистами, так что эта книга уникальна среди учебников по C++:

- Она предназначена для специалистов технической ориентации с небольшим опытом программирования или вообще без такового.

- Она предназначена для опытных программистов, которые хотят проработать язык более глубоко.

Можно ли в одной книге апеллировать к обеим этим группам? Ответ заключается в том, что изложение материала в книге направлено на достижение ясности программы посредством испытанной техники *структурного и объектно-ориентированного программирования*. При этом те, кто не знаком с программированием, будут изучать программирование, как положено, с самого начала. Мы попытались писать ясно и просто. Книга богата иллюстрирована. Возможно, самое важное то, что книга содержит огромное число программ на C++ и показывает результаты выполнения этих программ на компьютере.

Первые пять глав знакомят с основами компьютеров, программированием и языком программирования C++. Начинающие, прошедшие этот курс, говорят нам, что материал этих глав представляет солидную основу для глубокой проработки C++ в главах с 6 по 16. Опытные программисты обычно быстро прочитывают первые пять глав и затем убеждаются в том, что проработка C++ в главах с 6 по 16 отличается и строгостью, и поучительностью.

Многие опытные программисты говорили нам, что они высоко оценили нашу трактовку структурного программирования. Они часто программировали на структурированных языках, подобных С или Паскалю, но, поскольку они никогда формально не применяли структурное программирование, они не писали на этих языках оптимальных программ. Когда же они просмотрели материал по структурному программированию в начальных главах этой книги, они оказались в состоянии значительно улучшить свой стиль программирования на С и Паскале. Так что независимо от того, новичок вы или опытный программист, для вас найдется здесь много информативного, занимательного и поучительного.

Большинству людей хорошо известно многое из того, что делают современные компьютеры. Используя этот учебник, вы научитесь управлять компьютером, заставлять его делать то, что вам нужно. Компьютером (который часто называют *техническим обеспечением*), управляет *программное обеспечение* (т.е. инструкции, которые вы пишете, чтобы заставить компьютер выполнять какие-то действия и принимать решения), и C++ на сегодня является одним из наиболее популярных языков создания программного обеспечения. В этой книге дается введение в программирование на той версии языка C++, которая стандартизована в Соединенных Штатах Американским Национальным Институтом Стандартов (ANSI). Перспективы развития принятой во всем мире версия C++ связаны прежде всего с усилиями Международной Организации Стандартов (ISO).

Применение компьютеров возрастает почти во всех областях. В эпоху неуклонно растущих цен стоимость вычислений на глазах снижается вследствие быстрого развития как аппаратных средств, так и технологий программирования. Компьютеры, которые 25 лет назад занимали большие помещения и стоили миллионы долларов, ныне могут быть вписаны в поверхность кремниевых микросхем, размером меньших ногтя на пальце и стоящих, возможно, всего несколько долларов каждая. Ирония судьбы заключается в том, что кремний является одним из наиболее распространенных на земле материалов — он входит в состав обычного песка. Технология кремниевых микросхем сделала вычисления настолько экономичными, что во всем мире используется около 200 миллионов компьютеров общего назначения, помогающих людям в бизнесе, производстве, управлении и личной жизни. За несколько лет это число может легко удвоиться.

Эта книга будет занимательной для вас по нескольким причинам. Ваши сверстники за последние несколько лет, возможно, изучили С или Паскаль в качестве своих первых языков программирования. Вы же изучите и С, и С++! Как? Да просто потому, что С++ включает в себя стандарт ANSI для С и добавляет к нему много нового.

Ваши сверстники, возможно, изучали методологию программирования, называемую *структурным программированием*. Вы изучите как структурное программирование, так и современную новейшую методологию — *объектно-ориентированное программирование*. Зачем надо изучать обе эти методологии? Мы определенно предвидим, что объектно-ориентированный подход будет ключевой методологией программирования во второй половине 90-х годов. В этом курсе вы построите многие *объекты* и будете с ними работать. При этом вы обнаружите, что внутреннюю структуру этих объектов часто лучше создавать с помощью техники структурного программирования. И логика оперирования объектами в ряде случаев выражается лучше с помощью структурного программирования.

Другая причина, по которой мы представляем здесь обе методологии, состоит в том, что в предстоящее десятилетие будет происходить массовый переход от систем, основанных на С, к системам, основанным на С++. На местах существует огромное так называемое «наследие программ на С». С использовался почти четверть столетия и в последние годы его применение стремительно росло. Но те, кто изучил С++, обнаруживают, что его возможности гораздо более мощные, чем у С, и они часто предпочитают переходить на С++. Они начинают переводить свои системы на С++ и этот процесс достаточно ярко выражен. Затем они начинают использовать различные свойства С++, известные как «усовершенствования С++ по сравнению с С», чтобы улучшить свой стиль написания своих С-подобных программ. Наконец, они начинают пользоваться возможностями объектно-ориентированного программирования на С++, чтобы в полном объеме реализовать преимущества этого языка.

Интересный феномен, наблюдаемый на рынке языков программирования, состоит в том, что многие из основных продавцов продают теперь комбинированный С/С++ продукт охотнее, чем предлагаемые отдельные продукты. Это предоставляет пользователям возможность продолжать программировать на С, если они того пожелают, а затем постепенно продвигаться в сторону С++.

С++ имеет шансы стать основным языком разработки программ 90-х годов. Но может ли он быть объектом изучения в начальном курсе программирования — именно для такого курса предназначена эта книга? Мы думаем, что может. Два года назад мы приняли подобный вызов, когда Паскаль являлся основным языком в начальных курсах по вычислительной технике. Мы написали книгу «*Как программировать на С*» — родную сестру данной книги. Сотни университетов во всем мире используют теперь второе издание «*Как программировать на С*». Курсы, основанные на этой книге, доказали, что они столь же эффективны, как их предшественники, основанные на языке Паскаль. Между ними не наблюдалось никаких существенных различий, за исключением того, что студенты были более заинтересованы в изучении С, поскольку они знали, что предпочтительнее использовать С, чем Паскаль, и в последующих курсах, и для их будущей карьеры. Студенты, изучающие С, знали также, что они будут лучше подготовлены к изучению С++.

В первых пяти главах этой книги вы будете изучать структурное программирование на C++, С «как часть» С++ и С++ как расширение С. Чтобы сбалансировать эту книгу, вы изучите объектно-ориентированное программирование на С++. Мы не хотим заставлять вас ждать шестой главы, чтобы вы ощущали суть объектной ориентации. Поэтому каждая из первых пяти глав завершается разделом, называемым «Размышления об объектах». Эти разделы вводят основные концепции и терминологию объектной ориентации. Когда мы дойдем до шестой главы, вы уже будете хорошо подготовлены к тому, чтобы начать использовать С++ для построения объектов и написания объектно-ориентированных программ.

Данная первая глава содержит три части. Первая часть знакомит с основами компьютеров и программирования. Вторая часть заставит вас сразу начать писать некоторые простые программы на С++. Третья часть даст вам возможность «подумать об объектах».

Итак, за работу! Вы находитесь в самом начале увлекательного и благодарного путешествия. По мере вашего продвижения, если вы не считете за труд общаться с нами, посыпайте нам электронную почту через [Интернет deitel@world.std.com](mailto:deitel@world.std.com). Мы сделаем все возможное, чтобы быстро ответить вам. Желаем удачи!

1.2. Что такое компьютер?

Компьютер — это прибор, способный производить вычисления и принимать логические решения в миллионы или даже миллиарды раз быстрее человека. Например, многие из современных персональных компьютеров могут выполнять десятки миллионов операций сложения в секунду. Человеку, работающему с настольным калькулятором, потребовалось бы десятилетия для того, чтобы завершить тот же самый объем вычислений, который мощный персональный компьютер выполняет за одну секунду. (Информация к размышлению: Как вы могли бы узнать, правильно ли человек сложил числа? Как вы могли бы узнать, правильно ли компьютер сложил числа?) Сегодняшние самые быстрые *суперкомпьютеры* могут выполнять сотни миллиардов операций сложения в секунду — это примерно столько же, сколько сотни тысяч людей могут выполнить за год. А в исследовательских лабораториях уже функционируют компьютеры с быстродействием в триллионы операций в секунду.

Компьютеры обрабатывают *данные* под управлением наборов команд, называемых *компьютерными программами*. Эти компьютерные программы направляют действия компьютера посредством упорядоченных наборов действий, описанных людьми, называемыми *компьютерными программистами*.

Разнообразные устройства (такие как клавиатура, экран, диски, память и процессорные блоки), входящие в состав компьютерной системы, называются *аппаратными средствами*. Компьютерные программы, исполняемые компьютером, называются *программным обеспечением*. Стоимость аппаратных средств в последние годы существенно снизилась и достигла уровня, когда персональные компьютеры превратились в предмет массового потребления. К сожалению, стоимость разработки программного обеспечения неуклонно росла, так как программисты создавали все более мощные и сложные прикладные программы, не имея средств улучшить технологию их разработки. В этой книге вы изучите апробированные методы создания программного

обеспечения, которые могут снизить его стоимость — структурное программирование, нисходящую пошаговую детализацию, функционализацию и объектно-ориентированное программирование.

1.3. Организация компьютера

Независимо от различий в способах физической реализации каждый компьютер фактически можно разделить на шесть логических блоков или частей:

1. **Входной блок.** Это «воспринимающая» часть компьютера. Она получает информацию (данные и компьютерные программы) от различных *устройств ввода* и размещает ее в других устройствах для последующей обработки. Большая часть информации поступает сегодня в компьютер через клавиатуру, подобную пишущей машинке, и устройство, называемое «мышью». В будущем, возможно, большая часть информации будет вводиться в компьютер с голоса.
2. **Выходной блок.** Эта часть компьютера выполняет роль «перевозчика». Она забирает информацию, которая была обработана компьютером, и размещает ее в различных *выходных устройствах*, чтобы сделать пригодной для использования вне компьютера. Большая часть *выходной информации* компьютера отображается сегодня на экране, печатается на бумаге или используется для управления другими устройствами.
3. **Блок памяти.** Это быстродоступная и относительно малоемкая часть компьютера, выполняющая роль «склада». Она хранит информацию, которая была введена через входной блок, и эта информация может стать доступной для обработки, как только это потребуется. Блок памяти хранит также информацию, которая уже обработана, до тех пор пока она не окажется размещенной в других устройствах выходным блоком. Блок памяти часто называют либо *памятью*, либо *первичной памятью*.
4. **Арифметико-логическое устройство (АЛУ).** Это «обрабатывающая» часть компьютера. Она отвечает за выполнение вычислений, таких, как сложение, вычитание, умножение и деление. Она содержит решающие механизмы, которые позволяют компьютеру, например, сравнивать два элемента из блока памяти, чтобы определить, равны они или нет.
5. **Центральное процессорное устройство (ЦПУ).** Это «административная» часть компьютера. Она координирует работу компьютера и осуществляет надзор за работой всех других частей. ЦПУ указывает входному блоку, когда информация должна быть считана в блок памяти, указывает АЛУ, когда информация из памяти должна быть использована в вычислениях, и указывает выходному блоку, когда послать информацию из блока памяти на определенное выходное устройство.
6. **Блок вспомогательных запоминающих устройств.** Эта часть является «складом» высокой емкости для долгосрочного хранения информации. Программы или данные, не используемые активно другими блоками, обычно размещаются во вспомогательных запоминающих устройствах

(таких, как диски) до тех пор, пока они снова не потребуются, возможно, спустя дни, месяцы или даже годы. Доступ к этой информации гораздо более медленный, чем к информации в первичной памяти. В то же время стоимость единицы памяти во вспомогательных запоминающих устройствах много меньше, чем в первичной памяти.

1.4. Эволюция операционных систем

Ранние компьютеры могли выполнять одновременно только одно *задание* или *задачу*. Такая форма компьютерной работы часто называется однопользовательской *пакетной обработкой*. Компьютер выполняет в каждый отрезок времени единственную программу, а обрабатываемые данные находятся в группах или пакетах. В этих ранних системах пользователи обычно представляли свои задачи в вычислительный центр в виде пакетов перфокарт. Пользователи часто были вынуждены ждать часами или даже днями, чтобы получить распечатки своих задач.

Системы программного обеспечения, называемые *операционными системами*, были разработаны для того, чтобы сделать использование компьютеров более удобным. Ранние операционные системы управляли плавным переходом от одной задачи к другой. Это минимизировало время, затрачиваемое операторами компьютеров на переключение между заданиями, и, следовательно, возрастало количество работ, которое компьютер мог обработать.

По мере возрастания мощности компьютеров становилось очевидным, что однопользовательский режим пакетной обработки редко позволял эффективно использовать компьютерные ресурсы. Было задумано сделать так, чтобы множество задач или задач имели возможность *совместно использовать* ресурсы компьютера для повышения эффективности их использования. Это так называемые *мультипрограммные* системы. Мультипрограммная система подразумевает «одновременное» выполнение на компьютере многих задач — компьютер разделяет свои ресурсы между задачами, претендующими на его внимание. В ранних мультипрограммных операционных системах пользователи все еще представляли свои работы в виде пакетов перфокарт и вынуждены были ждать результатов часы или дни.

В 60-х годах несколько групп в промышленности и университетах проложили путь операционным системам с *разделением времени*. Разделение времени — это специальный случай мультипрограммной системы, когда пользователи имеют доступ к компьютеру через *терминалы* — обычно устройства с клавиатурами и экранами. В типовых компьютерных системах с разделением времени десятки или даже сотни пользователей работают на компьютере одновременно. На самом деле компьютер не работает со всеми пользователями одновременно. В действительности он выполняет лишь маленькую порцию работы для одного пользователя, а затем переходит к обслуживанию следующего пользователя. Компьютер делает это так быстро, что он может обеспечить обслуживание каждого пользователя несколько раз в секунду. Тем самым создается *видимость* одновременного выполнения программ пользователей. Преимущество разделения времени состоит в том, что пользователь получает почти немедленные ответы на запросы по сравнению с длительными периодами ожидания в предыдущих системах организации вычислений.

1.5. Персональные вычисления, распределенные вычисления и вычисления на платформе клиент/сервер

В 1977 году фирма Apple Computer объявила о возможностях *персональных вычислений*. Первоначально это была мечта любителей. Компьютеры стали экономически достаточно доступными для приобретения их людьми в личных или деловых целях. В 1981 году компания IBM — самый крупный в мире продавец компьютеров, представила персональный компьютер IBM PC. Буквально на следующий день вычисления на персональных компьютерах получили прописку в бизнесе, промышленности и организации управления.

Но эти компьютеры были «изолированными» — люди выполняли свои работы на своих собственных вычислительных машинах, а потом переносили дискеты туда и обратно для обмена информации. Хотя ранние персональные компьютеры еще не были достаточно мощными для режима разделения времени между несколькими пользователями, тем не менее они могли быть связаны вместе в компьютерную сеть, иногда посредством телефонных линий, а иногда в локальные сети (LAN — local area network) внутри организации. Это привело к феномену *распределенных вычислений*, когда необходимые для данной организации вычисления выполняются не строго по месту установки некоторого центрального компьютера, а распределены путем подключения сети к местам, где выполняется реальная работа. Персональные компьютеры были достаточно мощными, чтобы электронным способом управлять как вычислительными запросами отдельных пользователей, так и основными коммуникационными задачами передачи информации.

Сегодняшние наиболее мощные персональные компьютеры имеют такие же мощности как и вычислительные машины, стоявшие десятилетие назад миллионы долларов. Наиболее мощные настольные вычислительные машины, называемые *рабочими станциями*, обеспечивают индивидуальных пользователей немыслимыми возможностями. Информация легко распределяется посредством компьютерных сетей: в них одни компьютеры, называемые *файл-серверами*, обеспечивают хранение программ и данных, которые могут быть использованы другими компьютерами типа *клиент*, распределенными по сети — отсюда термин *вычисления на платформе клиент/сервер*. С и C++ стали языками программирования, допускающими написание программ и для операционных систем, и для компьютерных сетей, и для распределенных приложений по технологии клиент/сервер. Сегодняшние популярные операционные системы, такие, как UNIX, OS/2 и Windows NT, обеспечивают все типы возможностей, рассмотренные в данном разделе.

1.6. Машинные языки, языки ассемблера и языки высокого уровня

Программисты пишут свои программы на разных языках программирования, некоторые из них непосредственно понятны компьютеру, другие требуют промежуточных шагов *трансляции*. На сегодня существуют сотни языков программирования. Их можно разделить на три основных типа:

1. Машинные языки

2. Языки ассемблера

3. Языки высокого уровня

Любой компьютер может непосредственно понимать лишь свой собственный *машинный язык*. Машинный язык — это «природный язык» определенного компьютера. Он определяется при проектировании аппаратных средств этого компьютера. Машинные языки в общем случае содержат строки чисел (в конечном счете сокращенные до единиц и нулей), которые являются командами компьютеру на выполнении большинства элементарных операций в тот или иной момент времени. Машинные языки *машинно-зависимы*, т.е. каждый машинный язык может быть использован только на компьютере одного определенного типа. Машинные языки тяжелы для человеческого восприятия, как это можно видеть из следующего примера программы на машинном языке, которая складывает сверхурочную зарплату с основной и запоминает результат как общую зарплату:

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

По мере повышения популярности компьютеров стало очевидно, что программирование на машинных языках просто слишком медленно и утомительно для большинства программистов. Вместо использования строк чисел, которые компьютер мог бы понимать непосредственно, программисты начали использовать похожие на английский язык аббревиатуры для представления элементарных компьютерных операций. Эти аббревиатуры, напоминающие английский язык, сформировали основу *языков ассемблера*. Для преобразования программ на языке ассемблера в машинный язык со скоростью компьютера были разработаны *программы трансляции*, называемые *ассемблерами*. Следующий фрагмент программы на языке ассемблера также складывает сверхурочную зарплату (OVERPAY) с основной (BASEPAY) и запоминает результат как общую зарплату (GROSSPAY), но он более понятен по сравнению со своим машинным аналогом:

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

Хотя такой код более понятен людям, он непонятен компьютеру до тех пор, пока не будет преобразован в компьютерный код.

Использование компьютеров резко возросло с появлением языков ассемблера, но эти языки все еще требовали много команд для полного описания даже простых задач. Для ускорения процесса программирования были разработаны *языки высокого уровня*, в которых иногда достаточно написать всего один оператор для решения реальной задачи. Программы трансляции, которые преобразуют программы на языках высокого уровня в машинные коды, называются *компиляторами*. Языки высокого уровня позволяют программисту писать программы, которые выглядят почти так же, как повседневный английский, и используют общепринятую математическую нотацию. Программа расчета зарплаты, написанная на языке высокого уровня, могла бы содержать такой оператор как:

```
grossPay = basePay + overtimePay
```

Очевидно, что языки высокого уровня гораздо удобнее с точки зрения программистов по сравнению с языками ассемблера и с машинными кодами. С и С++ относятся к числу наиболее мощных и наиболее распространенных языков высокого уровня.

Процесс компиляции программы с языка высокого уровня в машинный язык может занимать значительное время. Для непосредственного выполнения программ на языке высокого уровня без необходимости их компиляции в машинный язык были разработаны программы *интерпретаторы*. Хотя скомпилированные программы выполняются быстрее чем интерпретируемые, интерпретаторы популярны в таких условиях, когда программы часто перекомпилируются для добавления в них новых возможностей и исправления ошибок. Но когда программа разработана, ее скомпилированная версия будет выполняться более эффективно.

1.7. История С++

Язык С++ развился из С, который в свою очередь был создан на основе двух предшествующих языков — BCPL и В. Язык BCPL был создан в 1967 году Мартином Ричардом как язык для написания компиляторов и программного обеспечения операционных систем. Кен Томпсон предусмотрел много возможностей в своем языке В — дубликате BCPL и использовал В для создания ранних версий операционной системы UNIX в Bell Laboratories в 1970 году на компьютере DEC PDP-7. И BCPL, и В были «нетипичными» языками — каждый элемент данных занимал одно «слово» в памяти и бремя обработки элемента данных, например, как целого или действительного числа падало на плечи программиста.

Язык С был развит из В Деннисом Ритчи в Bell Laboratories и первоначально реализован на компьютере DEC PDP-11 в 1972 году. С использует многие важные концепции BCPL и В, а также добавляет типы данных и другие свойства. Первоначально С приобрел широкую известность как язык разработки операционной системы UNIX. Сегодня фактически все новые операционные системы написаны на С или на С++. В течение двух последних десятилетий С стал доступным для большинства компьютеров. С независим от аппаратных средств. При тщательной разработке на С можно написать *мобильные программы, переносимые на большинство компьютеров*.

В конце 70-х годов С развился в то, что теперь относят к «традиционному С», «классическому С» или «С Кернигана и Ритчи». Публикация издательством Prentice-Hall книги Кернигана и Ритчи «Язык программирования С» привлекла широкое внимание к этому языку. Эта публикация стала одной из наиболее удачных книг по вычислительной технике за все время.

Широкое распространение С на различных типах компьютеров (иногда называемых *аппаратными платформами*) привело, к сожалению, ко многим вариациям языка. Они были похожи, но несовместимы друг с другом. Это было серьезной проблемой для разработчиков программ, нуждавшихся в написании совместимых программ, которые можно было бы выполнять на нескольких plataформах. Стало ясно, что необходима стандартная версия С. В 1983 году при Американском Национальном Комитете Стандартов в области вычислительной техники и обработки информации был создан технический комитет X3J11, чтобы «обеспечить недвусмысленное и машинно-независимое определение языка». В 1989 году стандарт был утвержден. ANSI скоопери-

ровался с Международной Организацией Стандартов (International Standards Organizations — ISO), чтобы стандартизировать С в мировом масштабе; совместный стандарт был опубликован в 1990 году и назван ANSI/ISO 9899: 1990. Копии этого документа можно заказать в ANSI. Второе издание книги Кернигана и Ритчи, вышедшее в 1988 году, отражает эту версию, называемую ANSI C; эта версия языка используется теперь повсеместно.

Замечание по мобильности 1.1

Поскольку С — стандартизованный, аппаратно-независимый, широко доступный язык, приложения, написанные на С, часто могут выполняться с минимальными модификациями или даже без них на самых различных компьютерных системах.

C++ — расширение С — был разработан Бъерном Строустропом в начале 80-х годов в Bell Laboratories. C++ обеспечивает ряд свойств, которые «приводят в порядок» язык С, но, что более важно, он обеспечивает возможность *объектно-ориентированного программирования*. Это явилось революционной идеей в мире программного обеспечения. Быстро, корректное и экономное создание программного обеспечения остается иллюзорной мечтой и это в то время, когда требуется проектирование нового и все более мощного программного обеспечения. *Объекты* — это эффективные повторно используемые *компоненты* программного обеспечения, моделирующие элементы реального мира. Разработчики программного обеспечения обнаруживают, что использование достижений модульного, объектно-ориентированного проектирования может значительно повысить продуктивность групп разработки по сравнению с предшествующей популярной технологией программирования, такой, как структурное программирование. Объектно-ориентированные программы легче понимать, корректировать и модифицировать.

Было разработано много других объектно-ориентированных языков, включая наиболее заметный из них Smalltalk, разработанный в Центре Исследований Palo Alto компании Xerox (Palo Alto Research Centre — PARC). Smalltalk — чистый объектно-ориентированный язык — в нем буквально все является объектом. C++ — это гибридный язык, он предоставляет возможность программировать и в стиле С, и в объектно-ориентированном стиле, и в обоих стилях сразу. Существует глубокая уверенность в том, что с середины 90-х годов C++ станет доминирующим системно-образующим языком.

1.8. Библиотеки классов C++ и стандартная библиотека С

Программы на C++ содержат части, называемые *классами и функциями*. Вы можете программировать каждую часть, если вам необходимо сформировать C++ программу. Но большая часть программистов на C++ пользуется преимуществами богатого собрания уже существующих классов и функций из библиотек классов C++ и библиотеки стандартных функций ANSI C (которую C++ заимствовал из языка С версии ANSI). Таким образом, реально существуют две области изучения «мира» C++. Первая — это изучение C++ как такового и вторая — изучение того, как использовать классы из различных библиотек классов C++ и функции из стандартной библиотеки

ANSI C. На протяжении всей книги мы обсуждаем многие из этих классов и функций. Среди тех программистов, которые нуждаются в глубоком понимании функций библиотеки ANSI C, способов их реализации и использования для написания мобильных кодов, наиболее читаемой является книга Плаугера. Библиотеками классов обеспечивают преимущественно продавцы компиляторов, но многими библиотеками классов снабжают независимые продавцы программного обеспечения.

Замечание по технике программирования 1.1

Используйте для создания программ методологию стандартных блоков. Избегайте заново изобретать колесо. Используйте существующие кусочки программ — это называется повторным использованием кодов и служит основой объектно-ориентированного программирования.

Замечание по технике программирования 1.2

При программировании на C++ вы обычно будете использовать следующие стандартные блоки: классы из библиотек классов и функции из стандартной библиотеки ANSI C, классы и функции, созданные вами самими, и классы и функции, созданные другими людьми, но доступные вам.

Преимуществом создания собственных классов и функций является то, что вы точно знаете, как они работают. Вы имеете возможность исследовать код C++. Недостаток состоит в затратах времени и сил на проектирование и развитие новых функций и классов.

Совет по повышению эффективности 1.1

Использование функций стандартной библиотеки ANSI вместо написания собственных версий тех же функций может повысить эффективность программ, поскольку эти функции написаны специально с учетом эффективности их выполнения.

Замечание по мобильности 1.2

Использование функций стандартной библиотеки ANSI вместо написания собственных версий тех же функций может повысить мобильность программ, поскольку эти функции включены практически во все реализации C++.

1.9. Параллельный C++

В результате длительных исследований в Bell Laboratories были разработаны другие версии C и C++. Джехани разработал Параллельный C (Concurrent C) и Параллельный C++ — расширения C, включающие возможности спецификации параллельно выполняемых коллективных операций. Языки, подобные этим, станут популярными в следующем десятилетии, так как возрастет использование *мультипроцессоров*, т.е. компьютеров с более чем один ЦПУ. На момент написания этой книги языки Параллельный C и Параллельный C++ находятся пока еще в стадии первоначального исследования и не получили коммерческого применения. Обычно учебники по операцион-

ным системам включают значительные проработки параллельного программирования.

1.10. Другие языки высокого уровня

Несмотря на то, что разработаны сотни языков высокого уровня, лишь немногие из них получили широкое применение. Между 1954 и 1957 годами компанией IBM был разработан язык FORTRAN (FORmula TRANslator — транслятор формул), предназначенный для научных и инженерных применений, требующих сложных математических вычислений. FORTRAN до сих пор широко используется, особенно в инженерных приложениях.

В 1959 году группой производителей компьютеров и пользователей управляющими и промышленными компьютерами был разработан язык COBOL (COmmon Business Oriented Language — язык, ориентированный на задачи бизнеса). COBOL прежде всего использовался для решения коммерческих задач, требующих точной и эффективной обработки больших объемов данных. До сих пор более половины программного обеспечения для коммерческих задач еще пишется на COBOLe. Активно создают программы на COBOLe около миллиона людей.

Примерно в то же время, что и С, был разработан Pascal. Он был создан профессором Никлаусом Виртом и предназначался для учебных целей. Мы еще поговорим об этом языке в следующих разделах.

1.11. Структурное программирование

На протяжении 60-х годов попытки создания многих больших программных систем наталкивались на ряд трудностей. Графики создания программного обеспечения обычно не выполнялись, цены значительно превосходили бюджетные ассигнования, а конечные продукты отличались ненадежностью. Люди начали понимать, что создание программного обеспечения — гораздо более сложная задача, чем они себе представляли. Исследовательские работы 60-х годов привели к развитию *структурного программирования* — дисциплинированного подхода к написанию программ, отличающихся от неструктурированных программ ясностью, простотой тестирования и отладки и легкостью модификации. Принципы структурного программирования обсуждаются в главе 2. В главах с 3 по 5 разрабатывается ряд структурированных программ.

Одним из наиболее ощутимых результатов этих исследований была разработка в 1971 году Никлаусом Виртом языка программирования Паскаль (Pascal). Pascal, названный в честь математика и философа семнадцатого столетия Блеза Паскаля, был разработан для изучения структурного программирования в академической среде и вскоре стал наиболее предпочтительным языком программирования во многих университетах. К сожалению, в языке отсутствовали многие свойства, необходимые для его применения в коммерции, промышленности и управлении, так что в этих областях он не получил широкого распространения.

В течение 70 и начале 80-х годов под патронажем Министерства Обороны США был разработан язык программирования Ада. Для создания программных систем управления и руководства Министерства обороны США исполь-

зовались сотни отдельных языков. Но Министерство обороны хотело иметь один язык, полностью удовлетворяющий всем его запросам. В качестве базового языка был выбран Паскаль, но в конце концов язык Ада оказался совсем не похожим на Паскаль. Язык был назван по имени Ады Лавлейс, дочери поэта лорда Байрона. Леди Лавлейс приписывают честь написания первой в мире компьютерной программы (для спроектированного Чарльзом Беббиджем механического вычислительного устройства, названного Анализической Машиной). Наиболее важное свойство Ады называется *многозадачностью*; оно позволяет программистам определять много действий для из параллельного выполнения. Другие широко распространенные языки высокого уровня, которые мы уже обсуждали, включая С и С++, вообще говоря, позволяют писать программы, в которых одновременно можно выполнять только одно действие.

1.12. Общее описание типичной среды программирования на С++

Обычно системы программирования на С++ состоят из нескольких частей: среда программирования, язык, стандартная библиотека С и различные библиотеки классов. Рассмотрим типичную среду разработки программ на С++, показанную на рис. 1.1.

Как правило, для того, чтобы выполнить программу на С++, надо пройти шесть этапов (рис. 1.1): *редактирование, предварительную (препроцессорную) обработку, компиляцию, компоновку, загрузку и выполнение*. Мы остановимся на системе С++, ориентированной на UNIX. Если вы не пользуетесь системой UNIX, обратитесь к справочникам для вашей системы или спросите вашего инструктора, как приспособить эти задачи к вашей операционной среде.

Первый этап представляет собой редактирование файла. Он выполняется с помощью *редактора программ*. Программист набирает с помощью этого редактора свою программу на С++ и, если это необходимо, вносит в нее исправления. Программа запоминается на вспомогательном запоминающем устройстве, например, на диске. Имена файлов программ на С++ часто оканчиваются расширением .С (заметим, что С — прописная буква; некоторые операционные среды С++ требуют других расширений, таких как .срр или .схх; для получения более детальной информациисмотрите документацию по вашей операционной среде С++). В системе UNIX широко используются два редактора — vi и emacs. Некоторые пакеты С++, такие как Borland C++ и Microsoft C/C++, имеют встроенные редакторы, которые органично объединены с операционной средой программирования. Мы предполагаем, что читатель знает, как редактировать программу.

На следующем этапе программист дает команду *компилировать* программу. Компилятор переводит программу в машинный код (называемый также *объектным кодом*). В системе С++ перед началом этапа трансляции выполняется программа предварительной обработки. Эта программа в С++ подчиняется специальным командам, называемым *директивами препроцессора*, которые указывают, что в программе перед ее компиляцией нужно выполнить определенные преобразования. Обычно эти преобразования состоят во включении других текстовых файлов в файл, подлежащий компиляции, и выполнении различных текстовых замен. Наиболее общие директивы препро-

цессора обсуждаются в начальных главах; детальное обсуждение всех особенностей препроцессорной обработки дано в приложениях. Препроцессорная обработка инициируется компилятором перед тем, как программа преобразовывается в машинный код.

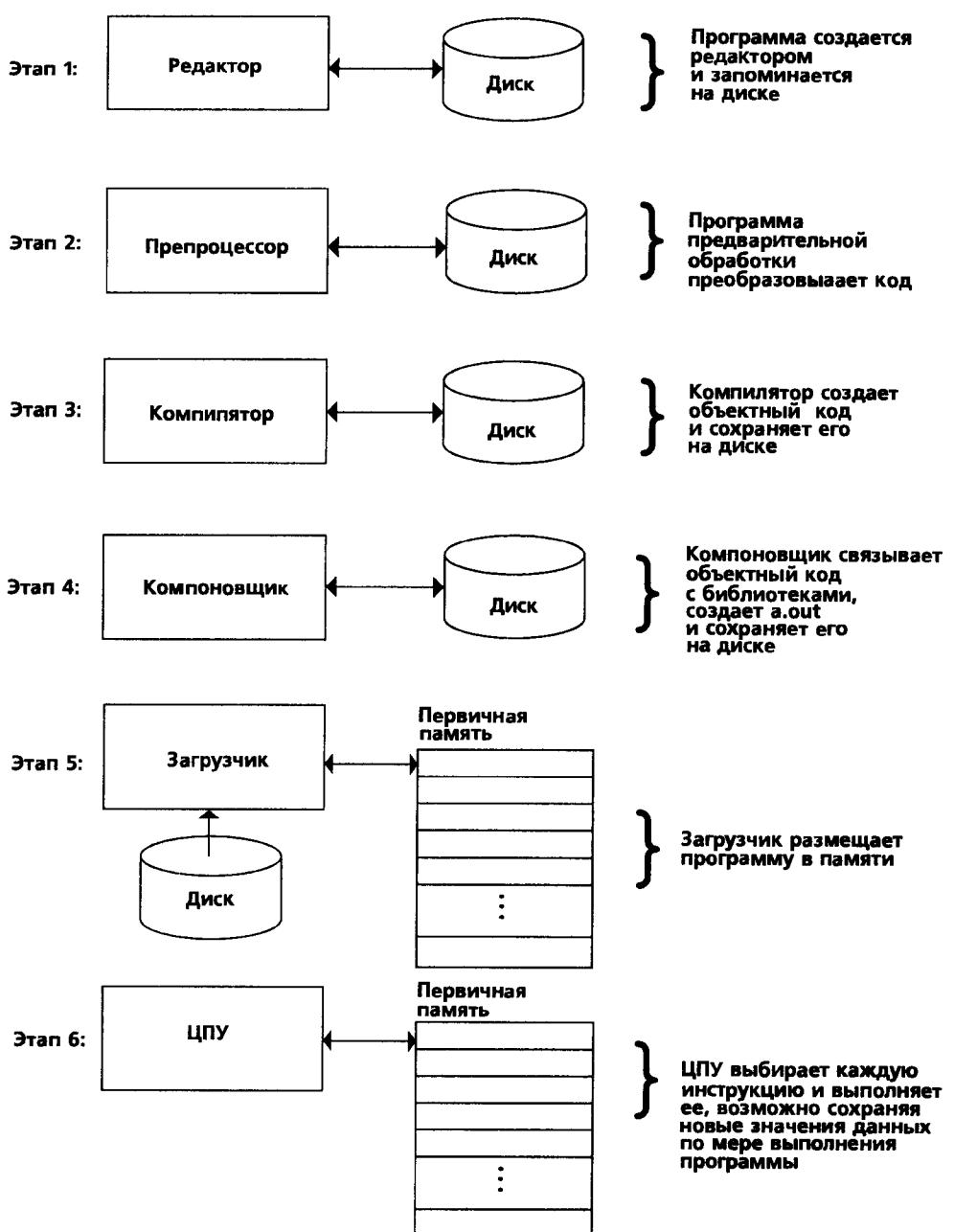


Рис. 1.1. Типичная среда программирования C++

Следующий этап называется *компоновка*. Программы на C++ обычно содержат ссылки на функции, определенные где-либо вне самой программы, например, в стандартных библиотеках или в личных библиотеках групп программистов, работающих над данным проектом. Объектный код, созданный компилятором, обычно содержит «дыры» из-за этих отсутствующих частей. *Компоновщик* связывает объектный код с кодами отсутствующих функций, чтобы создать *исполняемый загрузочный модуль* (без пропущенных частей). В типичной, основанной на UNIX, системе команда компиляции и компоновки программы на C++ обозначается как **CC**. Чтобы скомпилировать и скомпоновать программу по имени **welcom.C**, наберите по приглашению UNIX

```
CC welcom.C
```

и нажмите клавишу **return** (возврат каретки). Если программа скомпилировалась и скомпоновалась правильно, будет создан файл **a.out**. Это и есть исполняемый загрузочный модуль нашей программы **welcome.C**.

Следующий этап называется *загрузка*. Перед выполнением программы должна быть размещена в памяти. Это делается с помощью загрузчика, который забирает исполняемый загрузочный модуль с диска и перемещает его в память.

Затем компьютер под управлением своего ЦПУ выполняет поочередно в каждый момент времени по одной команде программы. Чтобы загрузить и выполнить программу в системе UNIX, мы набираем по приглашению UNIX команду **a.out** и нажимаем клавишу **return**.

Программа редко работает с первой попытки. Каждый из описанных этапов может окончиться неудачей из-за разного типа ошибок, которые мы обсудим в этой книге. Например, исполняемая программа может пытаться разделить на нуль (операция, не разрешенная в компьютерах так же, как и в арифметике). Это приведет к печати компьютером сообщения об ошибке. Программист должен вернуться к этапу редактирования, внести необходимые исправления и снова провести программу через остальные этапы, чтобы убедиться в правильности сделанных исправлений.

Типичная ошибка программирования 1.1

Ошибки, подобные делению на нуль, возникают во время выполнения программы, поэтому эти ошибки называются ошибками прогона или ошибками выполнения. Ошибка деления на нуль обычно является неисправимой ошибкой, т.е. ошибкой, вызывающей немедленное прекращение работы программы и исключающей ее нормальное выполнение. Исправимые ошибки позволяют программе продолжать счет до его завершения, часто приводя к неправильным результатам.

Большинство программ на C++ вводят и выводят данные. Определенные функции C++ выполняют ввод из **cin** (*the standard input stream — стандартный поток ввода*; произносится как «си-ин»), т.е. обычно с клавиатуры, но **cin** может быть связан и с другим устройством. Вывод данных производится в **cout** (*the standard output stream — стандартный поток вывода*; произносится как «си-аут»), т.е. обычно на экран, но **cout** может быть связан и с другим устройством. Когда мы говорим, что программа печатает результат, мы обычно подразумеваем, что результат отображается на экране. Данные могут быть выведены и на другие устройства, например на диски или на принтер в виде твердой копии. Существует также *стандартный поток ошибок* (*the standard error stream*), который обозначается **cerr**. Поток **cerr** (обычно связанный с экраном) используется для отображения сообщений об

ошибках. Как правило, пользователи направляют обычные выходные данные, т.е. `cout`, на отличные от экрана устройства, сохраняя для `cerr` экран, чтобы иметь возможность немедленно получать информацию об ошибках.

1.13. Общие замечания о C++ и об этой книге

C++ — трудный язык. Иногда программисты, экспериментирующие с C++, испытывают чувство гордости за то, что они оказались способны создать на нем хоть что-то, пусть и с грубым, искаженным и искривленным применением языка. Это — скверный стиль программирования. Программы получаются трудными для чтения, увеличивается вероятность их странного поведения, затрудняется их тестирование, отладка и адаптация к изменяющимся требованиям. Данная книга предназначена для программистов-новичков, поэтому мы делаем упор на ясность. Ниже приведен наш первый совет по «хорошему стилю программирования».

Хороший стиль программирования 1.1

Пишите программы на C++ в простом и четком стиле. Об этом иногда говорят как о KIS («keep it simple» — придерживайтесь простоты). Не «насилуйте» язык попытками его причудливого использования.

Мы будем давать много таких советов на протяжении всей книги, чтобы обратить ваше внимание на практические приемы, помогающие писать программы более ясные, более понятные, обеспечивающие удобство сопровождения, простые для проверки и отладки. Эти практические советы являются лишь путеводными вехами; вы, без сомнения, выработаете свой собственный стиль программирования. Мы будем также обращать ваше внимание на типичные ошибки программирования (с целью предостеречь вас от повторения этих ошибок в своих программах), давать советы по повышению эффективности (приемы, которые помогут вам писать программы, работающие быстрее и использующие меньше памяти), советы по мобильности (приемы, которые помогут вам писать программы, выполняемые на разных компьютерах с минимальными доработками или вообще без них) и советы по технике программирования (мысли и идеи, затрагивающие и улучшающие общую архитектуру систем программного обеспечения и, в частности, больших программных систем).

Вы только что услышали, что С и C++ — это мобильные языки и что написанные на них программы могут выполняться на многих разных компьютерах. *Мобильность (переносимость)* — это несбыточная мечта. Стандарт ANSI С содержит длинный перечень дискуссионных вопросов мобильности, были написаны целые книги, в которых обсуждается мобильность языка С.

Замечание по 1.3

Написать мобильную (переносимую) программу можно, но для разных компиляторов С и C++ и для разных компьютеров возникнет множество проблем, которые могут сделать мобильность труднодостижимой. Само по себе написание программ на С и C++ еще не обеспечивает мобильность. Программист будет вынужден часто считаться с особенностями разновидностей компиляторов и компьютеров.

Мы тщательно проследили эволюцию стандарта ANSI C++ и проверили наши аргументы против него на полноту и точность. Однако, C++ — богатый язык и в нем существуют некоторые тонкости и некоторые прогрессивные моменты, которых мы еще не касались. Если вам необходимы дополнительные технические подробности по C++, мы предлагаем вам прочитать наиболее современное краткое изложение стандарта ANSI C++. Другой доступный источник (хотя немного устаревший) — *Аннотированное Справочное Руководство по C++* Маргарет Эллис и Бьерна Струуструпа (Addison Wesley Publishing Company, 1991).

Многие возможности современных версий C++ не совместимы с более ранними реализациями C++, поэтому может оказаться, что некоторые из программ в этой книге не будут работать со старыми компиляторами C++.

Хороший стиль программирования 1.2

Читайте руководства для той версии C++, которой вы пользуетесь. Чаще обращайтесь к этим руководствам, чтобы быть уверенным в знании богатого набора возможностей C++ и в том, что вы правильно пользуетесь этими возможностями.

Хороший стиль программирования 1.3

Ваши компьютер и компилятор — хорошие учителя. Если после тщательного чтения руководства по той версии C++, которой вы пользуетесь, у вас нет уверенности в правильном понимании того, как реализуется та или иная особенность C++, экспериментируйте исмотрите, что происходит. Установите опции вашего компилятора в режим «предупреждений о максимальном числе ситуаций». Изучайте каждое сообщение, поступающее к вам при компиляции ваших программ и исправляйте программы до исчезновения этих сообщений.

1.14. Введение в программирование на C++

Язык C++ облегчает структурированный и упорядоченный подход к проектированию компьютерных программ. Теперь мы познакомимся с программированием на C++ и представим несколько примеров, иллюстрирующих многие важные свойства C++. Каждый пример посвящен одному оператору. В главе 2 мы детально рассмотрим *структурное программирование* на C++. Затем мы используем структурный подход при изложении материала главы 5. Начиная с шестой главы, мы изучим объектно-ориентированное программирование на C++. Из-за особой важности объектно-ориентированного программирования в этой книге каждая из первых пяти глав заканчивается разделом, озаглавленным «Размышления об объектах». Эти специальные разделы знакомят с основными идеями объектного подхода.

1.15. Простая программа: печать строки текста

В C++ используются формы записи, которые непрограммисту могут показаться странными. Мы начинаем с рассмотрения простой программы, печатающей строку текста. Программа и результаты ее работы на экране показаны на рис. 1.2.

```
// Первая программа на C++
#include <iostream.h>

main()
{
    cout << "Добро пожаловать в C++!\n";
    return 0; // показывает, что программа успешно окончена
}

Добро пожаловать в C++!
```

Рис. 1.2. Программа печати текста

Эта программа иллюстрирует несколько важных свойств языка C++. Рассмотрим детально каждую строку программы.

Строка

```
// Первая программа на C++
```

начинается с символа `//`, показывающего, что остальная часть строки — это **комментарий**. Программисты вставляют комментарии, чтобы *документировать* программу и облегчить ее чтение. Комментарии помогают другим людям читать и понимать вашу программу. Комментарии не вызывают никаких действий компьютера при выполнении программы. Они игнорируются компилятором C++ и не вызывают генерации каких-либо объектных кодов на машинном языке. Комментарий **первая программа на C++** просто описывает цель программы. Комментарий, который начинается с `//`, называется **однострочным комментарием**, потому что комментарий заканчивается в конце текущей строки. Позже мы обсудим альтернативную запись комментария, которая облегчает написание встроенных и многострочных комментариев.

Хороший стиль программирования 1.4

Каждая программа должна начинаться с комментария, описывающего цель программы.

Строка

```
#include <iostream.h>
```

является **директивой препроцессора**, т.е. сообщением препроцессору C++. Строки, начинающиеся с `#`, обрабатываются препроцессором перед компиляцией программы. Данная строка дает указание препроцессору включить в программу содержание **головного файла ввода/вывода iostream.h**. Этот файл должен быть включен для всех программ, которые выводят данные на экран или вводят данные с клавиатуры, используя принятый в C++ стиль, основанный на понятии потока ввода-вывода. Как вы вскоре увидите, программа на рис. 1.2 выводит данные на экран. Содержание **iostream.h** будет детально объяснено в главе 3.

Типичная ошибка программирования 1.2

Забывают включить файл **iostream.h** в программу, которая вводит данные с клавиатуры или выводит данные на экран.

Строка

```
main()
```

является частью каждой программы на C++. Круглые скобки после `main` показывают, что `main` — это программный блок, называемый *функцией*. Программа на C++ содержит одну или более функций, одна из которых должна быть `main`. Программа на рис. 1.2 содержит только одну функцию. Обычно программа на C++ начинается выполнением функции `main`, даже если `main` — не первая функция программы.

Левая фигурная скобка `{` должна начинать *тело* каждой функции. Соответствующая *правая фигурная скобка* должна заканчивать каждую функцию. Страна

```
cout << "Добро пожаловать в C++!\n";
```

является командой компьютеру напечатать на экране строку символов, заключенную в кавычки. Полная строка, включающая `cout`, операцию `<<`, строку "Добро пожаловать в C++!\n" и точку с запятой `(;)`, называется *оператором*. Каждый оператор должен заканчиваться точкой с запятой (известной также как *признак конца оператора*). Все вводы и выводы в C++ выполняются над потоками символов. Таким образом, когда выполняется предыдущий оператор, он посыпает поток символов Добро пожаловать в C++! объекту *стандартный поток вывода cout*, который обычно «связан» с экраном. Мы обсудим `cout` более подробно в главе 11, «Поток ввода-вывода».

Операция `<<` называется *операцией поместить в поток*. При выполнении этой программы значение справа от оператора, правый *операнд*, помещается в поток вывода. Символы правого операнда обычно выводятся в точности так, как они выглядят между двойными кавычками. Заметим, однако, что символы `\n` не выводятся на экране. Обратный слэш `(\)` называется *знаком перехода* или *escape-символом* (эскейп). Он свидетельствует о том, что должен выводиться «специальный» символ. Когда обратный слэш встречается в цепочке символов, следующий символ комбинируется с обратным слэшем и формирует *управляющую последовательность* (*escape-последовательность*). Управляющая последовательность `\n` означает *новую строку*. Она вызывает перемещение курсора (т.е. индикатора текущей позиции на экране) к началу следующей строки на экране. Некоторые другие управляющие последовательности приведены на рис. 1.3.

Управляющая последовательность	Описание
<code>\n</code>	Новая строка. Позиционирование курсора к началу следующей строки.
<code>\t</code>	Символ горизонтальной табуляции. Перемещение курсора к следующей позиции табуляции.
<code>\r</code>	Возврат каретки. Позиционирование курсора к началу текущей строки; запрет перехода к следующей строке.
<code>\a</code>	Сигнал тревоги. Звук системного звонка.
<code>\\\</code>	Обратный слэш. Используется для печати символа обратного слэша.
<code>\"</code>	Двойные кавычки. Используются для печати символа двойных кавычек.

Рис. 1.3. Некоторые общие управляющие последовательности

Типичная ошибка программирования 1.3

Пренебрежение точкой с запятой в конце оператора является синтаксической ошибкой. Синтаксическая ошибка возникает тогда, когда компилятор не может распознать оператор. Обычно компилятор выдает сообщение об ошибке, чтобы помочь программисту локализовать и исправить неверный оператор. Синтаксические ошибки — это нарушения правил языка. Синтаксические ошибки называются также ошибками компиляции или ошибками во время компиляции, потому что они обнаруживаются на этапе компиляции.

Строка

```
return 0; //показывает, что программа успешно окончена
```

включается в конце каждой функции `main`. Ключевое слово C++ `return` — один из нескольких способов *выхода из функции*. Когда оператор `return` используется в конце `main`, как в этой программе, значение 0 свидетельствует о том, что программа завершена успешно. В главе 3, где детально обсуждаются функции, станут яснее причины включения этого оператора. Ну а пока будем просто включать этот оператор в каждую программу, иначе в некоторых системах компилятор может выдавать предупреждение.

Правая фигурная скобка } означает окончание `main`.

Хороший стиль программирования 1.5

Последний символ, печатаемый любой функцией печати, должен быть `\n` (новая строка). Это дает уверенность в том, что функция переместит курсор в начало новой строки. Соглашения подобного рода способствуют возможности повторного использования программного обеспечения — ключевой задаче любой среды разработки программного обеспечения.

Хороший стиль программирования 1.6

Делайте одинаковые отступы для всего тела каждой функции внутри фигурных скобок, определяющих тело функции. При этом функциональная структура программы получается более понятной и легкой для чтения.

Хороший стиль программирования 1.7

Установите соглашение о величине желательного отступа и затем везде придерживайтесь этого соглашения. Для создания отступа можно использовать клавишу табуляции, а позиции табуляции можно изменять. Мы рекомендуем для формирования величины отступа примерно 0,5 см или (предпочтительнее) три пробела.

Добро пожаловать в C++! можно напечатать несколькими способами. Например, программа на рис. 1.4 многократно использует операторы вывода в поток, при этом получается такой же результат, как в программе на рис. 1.2. Допустимость такого способа объясняется тем, что каждый оператор вывода в поток возобновляет печать с того места, в котором предыдущий оператор ее прекратил. Первый оператор печатает пробел после **Добро пожаловать**, а второй оператор начинает печатать на той же самой строке с позиции, следующей за пробелом. Вообще C++ позволяет программисту представлять операторы множеством способов.

```
// Печать строки с помощью группы операторов
#include <iostream.h>

main()
{
    cout << "Добро пожаловать ";
    cout << "в C++!\n";

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

Добро пожаловать в C++!
```

Рис. 1.4. Печать на одной строке с помощью отдельных операторов **cout**.

Один оператор может напечатать группу строк, как показано на рис. 1.5. Каждый раз, когда в потоке вывода встречается управляющая последовательность вида `\n` (новая строка), курсор перемещается к началу следующей строки. Чтобы на выходе получить пустую строку, просто поместите подряд два символа новой строки.

```
// Печать группы строк с помощью одного оператора
#include <iostream.h>

main()
{
    cout << "Добро пожаловать\nв\nC++!\n";
    return 0; //показывает, что программа успешно завершена
}

Добро пожаловать
в
C++!
```

Рис. 1.5. Печать группы строк с помощью одного оператора **cout**

1.16. Другая простая программа: сложение двух целых чисел

Наша следующая программа использует объект `cin` входного потока и операцию *взять из потока* `<<`, чтобы получить два целых числа, введенных пользователем на клавиатуре, посчитать сумму этих чисел и вывести этот результат с помощью `cout`. Программа и результат ее работы показаны на рис. 1.6.

Комментарий

```
// Программа сложения
```

указывает цель программы. Директива препроцессора C++

```
#include <iostream.h>
```

включает в программу содержание заголовочного файла `iostream.h`.

```
// Программа сложения
#include <iostream.h>

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

    cout << "Введите первое целое число\n"; //приглашение
    cin >> integer1; //чтение целого
    cout << "Введите второе целое число\n"; //приглашение
    cin >> integer2; //чтение целого
    sum = integer1 + integer2; //присваивание значения сумме
    cout << "Сумма равна " << sum << endl; //печатать суммы
    return 0; //показывает, что программа успешно завершена
}

Введите первое целое число
45
Введите второе целое число
72
Сумма равна 117
```

Рис. 1.6. Программа сложения

Как указывалось ранее, выполнение каждой программы начинается с использования функции `main`. Левая фигурная скобка отмечает начало тела `main`, а соответствующая правая фигурная скобка отмечает его конец. Стока

```
int integer1, integer2, sum; //объявление
```

называется *объявлением*. Слова `integer1`, `integer2` и `sum` являются именами *переменных*. Переменная — это область в памяти компьютера, где может храниться некоторое значение для использования его в программе. Данное объявление определяет, что переменные `integer1`, `integer2` и `sum` имеют тип данных `int`; это значит, что эти переменные всегда будут содержать *целые значения*, т.е. целые числа, такие как 7, -11, 0, 31914. Все переменные должны объявляться с указанием имени и типа данных прежде, чем они могут быть использованы в программе. Несколько переменных одного типа могут быть объявлены в одном или в нескольких объявлениях. Мы могли бы написать три объявления, по одному для каждой переменной, но предыдущее объявление более компактно.

Вскоре мы обсудим типы данных `float` (для спецификации действительных чисел, т.е. чисел с десятичными запятыми типа 3.4, 0.0, -11.19) и `char` (для спецификации символьных данных; переменная `char` может хранить только одну строчную букву или одну прописную букву, одну цифру, один специальный символ типа *, \$ и т.д.).

Хороший стиль программирования 1.8

Ставьте пробел после каждой запятой (,), чтобы программу было легче читать.

Имя переменной — это любой допустимый *идентификатор*. Идентификатором называется последовательность символов, содержащая буквы, цифры и символы подчеркивания (_), которая не начинается с цифры. В C++ допускаются идентификаторы любой длины, но ваша система или среда C++

могут налагать некоторые ограничения на длину идентификаторов. Язык C++ чувствителен к регистру — прописные и строчные буквы различаются, так что a1 и A1 — это разные идентификаторы.

Хороший стиль программирования 1.9

Выбор осмысленных имен переменных помогает programme быть «самодокументируемой», т.е. такую programme легче понимать при чтении, даже не обращаясь к справочным пособиям или обширным комментариям.

Хороший стиль программирования 1.10

Избегайте идентификаторов, которые начинаются с подчеркиваний, потому что компилятор C++ может использовать похожие на них имена для своих собственных внутренних целей. Это предотвратит путаницу в именах, выбираемых вами и компилятором.

Объявления могут размещаться в функциях почти всюду. Однако, объявления переменных должны предшествовать их использованию в programme. Например, в programme на рис. 1.6 вместо использования одного объявления для всех трех переменных можно было использовать три отдельных объявления. Объявление

```
int integer1;
```

можно было поместить непосредственно перед строкой

```
cin >> integer1;
```

объявление

```
int integer1;
```

можно было поместить перед строкой

```
cin >> integer2;
```

и объявление

```
int sum;
```

можно было поместить перед строкой

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

Хороший стиль программирования 1.11

Всегда помещайте пустую строку перед объявлением, которое находится между выполняемыми операторами. Это делает объявление заметными в programme и способствует ее четкости.

Хороший стиль программирования 1.12

Если вы предпочитаете размещать объявления в начале функции, отделяйте эти объявления от выполняемых операторов в этой функции пустой строкой, чтобы выделить конец объявлений и начало выполняемых операторов.

Оператор

```
cout << "Введите первое целое число\n"      // приглашение
```

печатает на экране буквенное сообщение **Введите первое целое число** и позиционирует курсор на начало следующей строки. Это сообщение называется *приглашением*, потому что оно предлагает пользователю выполнить некоторое действие. О предыдущем операторе можно сказать так: «*cout получает символьную строку "Введите первое целое число\n"*».

Оператор

```
cin >> integer1;                      // чтение целого
```

использует объект входного потока **cin** и операцию **взять из потока >>**, чтобы получить от пользователя значение. Объект **cin** забирает вводимую информацию из стандартного потока ввода, которым обычно является клавиатура. О предыдущем операторе можно сказать так: «*cin дает значение первого целого числа*».

Когда компьютер выполняет предыдущий оператор, он ждет от пользователя ввода значения переменной **integer1**. В ответ пользователь набирает на клавиатуре целое число и затем нажимает *клавишу возврата — return* (называемую иногда *клавишей ввода — enter*), чтобы послать это число в компьютер. Компьютер затем присваивает это число, или *значение*, переменной **integer1**. Любое последующее обращение в программе к **integer1** будет использовать это значение.

Объекты потоков **cout** и **cin** вызывают взаимодействие между пользователем и компьютером. Поскольку это взаимодействие напоминает диалог, часто говорят о *диалоговом расчете или интерактивном расчете*.

Оператор

```
cout << "Введите второе целое число\n"; // приглашение
```

печатает на экране сообщение **Введите второе целое число** и затем позиционирует курсор на начало следующей строки. Этот оператор приглашает пользователя выполнить действие. Оператор

```
cin >> integer2;                      // чтение целого
```

получает от пользователя значение переменной **integer2**.

Оператор присваивания

```
sum = integer1 + integer2;           // присваивание значения сумме
```

рассчитывает сумму переменных **integer1** и **integer2** и присваивает результат переменной **sum**, используя *операцию присваивания* **=**. Оператор читается так: «*sum получает значение, равное integer1 + integer2*». Оператор присваивания используется в большинстве расчетов. Операция **=** и операция **+** называются *бинарными операциями*, потому что каждая из них имеет по два *операндов*. В случае операции **+** этими operandами являются **integer1** и **integer2**. В случае операции **=** двумя operandами являются **sum** и значение выражения **integer1 + integer2**.

Хороший стиль программирования 1.13

Размещайте пробелы с обеих сторон бинарной операции. Это выделит операцию и улучшит читаемость программы.

Оператор

```
cout << "Сумма равна " << sum << endl; //печатать суммы
```

печатает символьную строку **Сумма равна**, затем численное значение переменной **sum**, за которым следует **endl** (аббревиатура словосочетания «end line» — конец строки) — так называемый *манипулятор потока*. Манипулятор **endl** выводит символ новой строки и затем «очищает буфер вывода». Это просто означает, что в некоторых системах, где выводы накапливаются в вычислительной машине до тех пор, пока их не станет достаточно, чтобы «имело смысл печатать на экране», **endl** вызывает немедленную печать на экране всего накопленного.

Заметим, что предыдущий оператор выводит множество значений разных типов. Операция поместить в поток «знает», как выводить каждую единицу данных. Многократное использование операции поместить в поток (**<<**) в одном операторе называется *цепленной операцией поместить в поток*. Таким образом, не обязательно иметь множество операций вывода для вывода множества фрагментов данных.

В операторах вывода можно также выполнять вычисления. Мы могли бы объединить два предыдущих оператора в один

```
cout << "Сумма равна" << integer1 + integer2 << endl;
```

Правая фигурная скобка **}** информирует компьютер о том, что функция **main** окончена.

Мощным свойством C++ является предоставление пользователям возможности создавать свои собственные типы данных (мы исследуем этот вопрос в главе 6). Пользователи затем могут «научить» C++ вводить и выводить значения этих новых типов данных, используя операции **>>** и **<<** (это называют *перегрузкой операций* — мы исследуем этот вопрос в главе 8).

1.17. Концепции памяти

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

В программе сложения на рис. 1.6 при выполнении оператора

```
cin >> integer1;
```

значение, введенное пользователем, помещается в область памяти, которой компилятор присвоил имя **integer1**. Допустим, пользователь вводит число 45 как значение **integer1**. Компилятор разместит 45 в области памяти **integer1**, как показано на рис. 1.7.

Всякий раз, когда значение помещается в область памяти, оно замещает предыдущее значение, хранившееся в этом месте. Предыдущее значение при этом пропадает.

Возвращаясь к нашей программе сложения допустим, что при выполнении оператора

```
cin >> integer2;
```

пользователь вводит значение 72. Это значение помещается в область памяти **integer2** и теперь распределение памяти выглядит так, как показано на рис. 1.8. Заметим, что соседство этих областей в памяти необязательно.

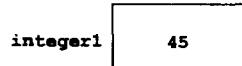


Рис. 1.7. Распределение памяти с указанием имени и значения переменной

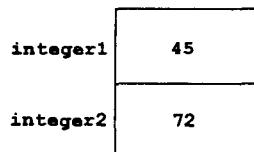


Рис. 1.8. Распределение памяти после ввода значений двух переменных

Когда программа получила значения `integer1` и `integer2`, она складывает их и помещает сумму в переменную `sum`. Оператор

```
sum = integer1 + integer2
```

выполняет сложение и подразумевает уничтожение предыдущего значения `sum`. Это происходит, когда вычисленное значение суммы `integer1` и `integer2` помещается в область памяти `sum` (независимо от того, каково было прежнее значение `sum`). После вычисления `sum` распределение памяти выглядит так, как показано на рис. 1.9. Заметим, что значения `integer1` и `integer2` выглядят точно так же, как и до их использования при вычислении `sum`. Эти значения использовались, но не пропали во время выполняемых компьютером вычислений. Таким образом, считывание значений из памяти — процесс неразрушающий.

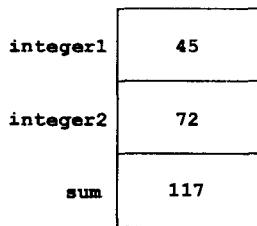


Рис. 1.9. Распределение памяти после вычислений

1.18. Арифметика

Большинство программ выполняет арифметические вычисления. Множество *арифметических операций* показано на рис. 1.10. Отметим использование в них разнообразных специальных символов, не используемых в алгебре. Звездочка (*) обозначает умножение, а знак процента (%) — это операция *вычисления остатка*, которая вкратце будет еще обсуждаться. Арифметические операции на рис. 1.10 являются бинарными операциями. Например, выражение `integer1 + integer2` содержит бинарную операцию + и два операнда `integer1` и `integer2`.

Операция C++	Арифметическая операция	Алгебраическое выражение	Выражение на C++
Сложение	+	$f + 7$	$f + 7$
Вычитание	-	$p - c$	$p - c$
Умножение	*	$b m$	$b * m$
Деление	/	x/y или $\frac{x}{y}$ или $x \div y$	x / y
Вычисление остатка	%	$r \bmod s$	$r \% s$

Рис. 1.10. Арифметические операции

Целочисленное деление дает целый результат; например, выражение $7 / 4$ равно 1, а выражение $17 / 5$ равно 3. Заметим, что любая десятичная часть при целочисленном делении просто отбрасывается (т.е. усекается) — округление не производится. В C++ имеется операция вычисления остатка %, которая дает в качестве результата остаток от целочисленного деления. Выражение $x \% y$ дает остаток от деления x на y . Таким образом, $7 \% 4$ равно 3, $17 \% 5$ равно 2. В последующих главах мы обсудим много интересных применений операции вычисления остатка, таких, как определение, является ли одно число кратным другому.

Типичная ошибка программирования 1.4

Попытка использования операции вычисления остатка % с нецелочисленными operandами является синтаксической ошибкой.

Арифметические выражения на C++ должны быть записаны в *одну строку*, чтобы программу можно было ввести в компьютер. Таким образом, такое выражение как «*a разделить на b*» должно быть записано в виде a / b так, чтобы все константы, переменные и операции размещались в одной строке. Алгебраическая запись

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

в общем случае не пригодна для компиляции, хотя существуют некоторые специальные пакеты программ, которые поддерживают более естественную форму записи сложных математических выражений.

Круглые скобки используются в C++ в основном точно так же, как и в алгебраических выражениях. Например, чтобы перемножить величины a и $b+c$, мы пишем

$$a * (b + c)$$

C++ применяет операции в арифметических выражениях в порядке старшинства, определенного *правилами старшинства операций*, которые, вообще говоря, совпадают с теми, которые используются в алгебре:

1. Операции в выражениях, заключенных внутри круглых скобок, выполняются в первую очередь. Таким образом, *круглые скобки можно использовать для формирования любой желательной для программиста последовательности вычислений*. О скобках говорят, что они имеют «высший уровень приоритета». В случае *вложенных*, или *встроенных* круглых скобок операции внутри самой внутренней пары круглых скобок выполняются первыми.

2. Следующими выполняются операции умножения, деления и вычисления остатка. Если выражение содержит несколько операций умножения, деления и вычисления остатка, то операции выполняются слева направо. Говорят, что операции умножения, деления и вычисления остатка имеют одинаковый уровень приоритета.
3. Операции сложения и вычитания выполняются последними. Если выражение содержит несколько операций сложения и вычитания, то операции выполняются слева направо. Операции сложения и вычитания тоже имеют одинаковый уровень приоритета.

Правила старшинства операций дают C++ возможность выполнять операции в правильной последовательности. Когда мы говорим о выполнении операций слева направо, то мы имеем в виду *ассоциативность* операций. Мы увидим, что у некоторых операций *ассоциативность* справа налево. Рис. 1.11 обобщает правила старшинства операций. Эта таблица будет расширяться по мере знакомства с дополнительными операциями C++. Полная таблица старшинства операций приведена в приложении Б.

Символ операции	Имя операции	Последовательность выполнения (старшинство)
()	Круглые скобки	Выполняются первыми. Если круглые скобки – вложенные, выражение внутри самой внутренней пары вычисляется первым. Если имеется несколько пар круглых скобок «одинакового уровня» (т.е. не вложенных), они выполняются слева направо.
* / или %	Умножение Деление Вычисление остатка	Выполняются вторыми. Если их несколько, они выполняются слева направо.
+ или -	Сложение Вычитание	Выполняются последними. Если их несколько, они выполняются слева направо.

Рис. 1.11. Последовательность выполнения арифметических операций

Давайте теперь рассмотрим несколько выражений в свете правил, определяющих последовательность выполнения операций. Каждый пример представляется алгебраическим выражением и его эквивалентом на C++.

Ниже приводится пример среднего арифметического значения пяти членов:

Алгебра: $m = \frac{a + b + c + d + e}{5}$

C++: $m = (a + b + c + d + e) / 5;$

Круглые скобки необходимы, так как деление имеет более высокий уровень приоритета, чем сложение. Полная величина $(a + b + c + d + e)$ делится на 5. Если ошибочно проигнорировать круглые скобки, получим $a+b+c+d+e/5$, что эквивалентно

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

Следующий пример — уравнение прямой линии:

Алгебра: $y = mx + b$

C++: $y = m * x + b;$

Ни каких круглых скобок не требуется. Умножение выполняется первым, потому что оно имеет более высокий приоритет, чем сложение.

Следующий пример содержит операции вычисления остатка (%), умножения, деления, сложения и вычитания:

Алгебра: $z = pr \% q + w / x - y$

C++: $z = p * r \% q + w / x - y;$



Цифры в кружках под операциями обозначают последовательность, в которой C++ выполняет операции. Умножение, вычисление остатка и деление выполняются первыми в последовательности слева направо (т.е. их ассоциативность слева направо), так как они имеют более высокий приоритет, чем сложение и вычитание. Сложение и вычитание выполняются следующими в той же последовательности слева направо.

Не все выражения с несколькими парами круглых скобок содержат вложенные круглые скобки. Например, выражение

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

не содержит вложенных круглых скобок. В подобном случае принято говорить, что круглые скобки находятся на одном и том же уровне.

Для углубления понимания правил старшинства операций, рассмотрим вычисление полинома второй степени.

$y = a * x * x + b * x + c;$



Цифры в кружках под операциями обозначают последовательность, в которой C++ выполняет операции. В C++ нет арифметической операции возведения в степень, поэтому мы представляем x^2 как $x * x$. Скоро мы рассмотрим библиотечную функцию pow («power»), которая выполняет возведение в степень. Поскольку для понимания функции pow надо учитывать некоторые тонкие вопросы, относящиеся к типам данных, мы отложим детальное объяснение pow до третьей главы.

Допустим, что a , b , c и x имеют следующие начальные значения: $a = 2$, $b = 3$, $c = 7$ и $x = 5$. Рисунок 1.12 иллюстрирует последовательность, в которой выполняются операции в предыдущем примере вычисления полинома второй степени.

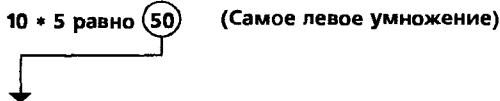
Как и в алгебре, для того, чтобы сделать выражение яснее, в нем можно расставлять дополнительные, не являющиеся необходимыми круглые скобки. Эти не являющиеся необходимыми круглые скобки называются *избыточными круглыми скобками*. Например, в предыдущем операторе скобки могут быть расставлены следующим образом:

$y = (a * x * x) + (b * x) + c;$

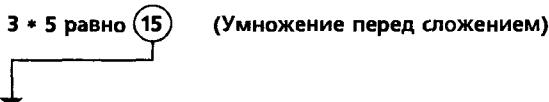
Шаг 1. $y = 2 * 5 * 5 + 3 * 5 + 7;$



Шаг 2. $y = 10 * 5 + 3 * 5 + 7;$



Шаг 3. $y = 50 + 3 * 5 + 7$



Шаг 4. $y = 50 + 15 + 7$



Шаг 5. $y = 65 + 7$



Рис. 1.12. Последовательность вычисления полинома второй степени

1.19. Принятие решений: операции проверки на равенство и отношения

Этот раздел познакомит вас с простой версией структуры *if* в C++, которая позволяет программе принимать решение, основываясь на истинности или ложности некоторого условия.

Если условие удовлетворено, т.е. условие есть *true* (истина), то оператор тела структуры *if* выполняется. Если условие не удовлетворяется, т.е. условие есть *false* (ложь), то оператор в теле не выполняется. Мы покажем вскоре на примере.

Условия в структурах *if* могут быть сформированы с использованием операций проверки на равенство и отношения, сводка которых приведена на рис. 1.13. Все операции отношения имеют одинаковый приоритет и ассоциативность слева направо. Операции проверки на равенство также имеют одинаковый уровень приоритета, который, однако, ниже приоритета операций отношения. Операции проверки на равенство также имеют ассоциативность слева направо.

кот
ти

в т
лов
это

опе
на
соц
оди
ций
нос

Обычная алгебраическая операция проверки на равенство или отношения	Операция C++ проверки на равенство или отношения	Пример условия на C++	Значение условия C++
Операции проверки на равенство			
=	==	x == y	x равен y
≠	!=	x != y	x не равен y
Операции отношения			
>	>	x > y	x больше у
<	<	x < y	x меньше у
≥	≥	x ≥ y	x больше или равен у
≤	≤	x ≤ y	x меньше или равен у

Рис. 1.13. Операции проверки на равенство и отношения

Типичная ошибка программирования 1.5

Если операции ==, !=, >= и <= содержат между своими символами пробелы, это — синтаксическая ошибка.

Типичная ошибка программирования 1.6

Перестановка символов в обозначении операций: вместо !=, >= и <= запись =!, => и =< соответственно вызывает синтаксическую ошибку.

Типичная ошибка программирования 1.7

Смешивание операции проверки на равенство == с операцией присваивания =. Операция проверки на равенство должна читаться как «равно», а операция присваивания должна читаться как «присваивает». Некоторые предпочитают читать операцию проверки на равенство как «двойное равенство». Как мы вскоре увидим, смешивание этих операций может вызывать нелегко распознаваемую синтаксическую ошибку, а может вызвать и чрезвычайно запутанные логические ошибки.

В приведенном ниже примере используется шесть операторов if для сравнения двух вводимых пользователем чисел. Если в каком-либо из этих операторов if условие удовлетворяется, выполняется связанный с данным if оператор вывода. Программа и три примера вывода показаны на рис. 1.14.

Заметим, что в программе на рис. 1.14 используются сцепленные операции взять из потока для ввода двух целых чисел. Сначала читается значение переменной num1, а затем — переменной num2. Отступы в операторах if облегчают чтение программы. Кроме того заметим, что каждый из операторов if на рис. 1.14 имеет в своем теле только один оператор. В главе 2 мы расскажем, как записать операторы if с телом, содержащем несколько операторов.

```
// Использование операторов if, операций отношения
// и операций проверки на равенство
#include <iostream.h>

main()
{
    int num1, num2;

    cout    << "Введите два целых числа и я скажу вам\n"
          << "соотношения, в которых они находятся: ";
    cin    >> num1 >> num2;      // прочитать два целых числа

    if (num1 == num2)
        cout << num1 << " равно " << num2 << endl;

    if (num1 != num2)
        cout << num1 << " не равно " << num2 << endl;

    if (num1 < num2)
        cout << num1 << " меньше " << num2 << endl;

    if (num1 > num2)
        cout << num1 << " больше " << num2 << endl;

    if (num1 <= num2)
        cout << num1 << " меньше или равно " << num2 << endl;

    if (num1 >= num2)
        cout << num1 << " больше или равно " << num2 << endl;

    return 0; // указывает на успешное завершение программы
}
```

```
Введите два целых числа и я скажу вам
соотношения, в которых они находятся: 3 7
3 не равно 7
3 меньше 7
3 меньше или равно 7
```

```
Введите два целых числа и я скажу вам
соотношения, в которых они находятся: 22 12
22 не равно 12
22 больше 12
22 больше или равно 12
```

```
Введите два целых числа и я скажу вам
соотношения, в которых они находятся: 7 7
7 равно 7
7 меньше или равно 7
7 больше или равно 7
```

Рис. 1.14. Использование операций проверки на равенство и отношения

Хороший стиль программирования 1.14

Отступы в теле структуры `if` выделяют тело структуры и упрощают чтение программы.

Хороший стиль программирования 1.15

В каждой строке программы должно быть не более одного оператора.

Типичная ошибка программирования 1.8

Постановка точки с запятой после правой круглой скобки, завершающей условие в структуре `if`. Такая ошибочная точка с запятой приводит к тому, что тело структуры становится пустым, так что сама структура `if` не выполняет никаких действий, независимо от того, истинно условие или нет. Более того, подлинный оператор тела структуры `if` превращается в оператор, следующий за структурой `if`, и выполняется всегда.

Отметим использование пробелов на рис. 1.14. В операторах C++ *символы-разделители*, такие, как символы табуляции, новой строки и пробела, обычно игнорируются компилятором. Так, например, операторы могут быть разбиты на несколько строк и могут разделяться промежутками в соответствии со вкусами программиста. Но нельзя разбивать идентификаторы.

Хороший стиль программирования 1.16

Оператор может занимать несколько строк. Если один оператор должен быть разбит на несколько строк, выбирайте характерные точки разбиения, такие, как позиция после запятой в списке с запятыми в качестве разделителя или позиция после операции в длинном выражении.

В таблице на рис. 1.15 показано старшинство операций, введенных в этой главе. Операции показаны сверху вниз по мере уменьшения приоритета. Заметим, что все эти операции, за исключением операции присваивания `=`, имеют ассоциативность слева направо. Сложение ассоциативно слева направо, так что выражение, подобное `x + y + z`, вычисляется так, как если бы оно было записано в виде `(x + y) + z`. Операции присваивания `=` ассоциативны справа налево, так что выражение, подобное `x = y = 0`, вычисляется так, как если бы оно было написано в виде `x = (y = 0)`, где сначала 0 присваивается переменной `y`, а затем результат этого присваивания — 0 — переменной `x`.

Операции	Ассоциативность	Тип операций
<code>()</code>	слева направо	круглые скобки
<code>*</code> <code>/</code> <code>%</code>	слева направо	мультипликативные
<code>+</code> <code>-</code>	слева направо	аддитивные
<code><</code> <code><=</code> <code>></code> <code>>=</code>	слева направо	отношение
<code>==</code> <code>!=</code>	слева направо	проверка на равенство
<code><<</code> <code>>></code>	слева направо	поместить в/взять из
<code>=</code>	справа налево	присваивание

Рис. 1.15. Приоритеты и ассоциативность рассмотренных операций

Хороший стиль программирования 1.17

Обращайтесь к таблице старшинства операций при написании выражений, содержащих много операций. Убедитесь, что операции в выражениях выполняются в ожидаемой вами последовательности. Если вы не уверены в последовательности выполнения в сложных выражениях, используйте для задания последовательности круглые скобки так же, как вы это делаете в алгебраических выражениях. Убедитесь путем наблюдений, что некоторые операции, такие как присваивание (`=`), имеют ассоциативность не слева направо, а справа налево.

Мы познакомились со многими важными особенностями C++, включая вывод данных на экран, ввод данных с клавиатуры, выполнение вычислений и принятие решений. В главе 2, построенной на этих приемах, мы познакомимся со *структурным программированием*. Вам станет более близка техника отступа. Мы изучим, как описывать и изменять порядок выполнения операторов — этот порядок называется *программным управлением*.

1.20. Размышления об объектах

Приступим теперь к начальному знакомству с объектной ориентацией. Мы увидим, что объектная ориентация — естественный способ размышлений о мире и написания компьютерных программ.

Но почему тогда мы не начали объектную ориентацию с первой страницы? Почему мы откладываем объектно-ориентированное программирование на C++ до шестой главы? Ответ заключается в том, что объекты, которые мы будем строить, будут состоять, в частности, из частей структурированных программ, так что нам необходимо сначала освоить основы структурного программирования.

В каждой из первых пяти глав мы обращаем внимание на «стандартную» методологию структурного программирования. Затем мы завершаем каждую главу пространным введением в объектную ориентацию. Какова будет наша стратегия в этих разделах по объектной ориентации? В этой первой главе мы познакомимся с основными концепциями (т.е. с «объектным мышлением») и терминологией (т.е. «объектным языком»). В главах со второй по пятую мы рассмотрим наиболее важные вопросы, а затем начнем активное изучение животрепещущих проблем техники объектно-ориентированных проектирования (OOD — *object-oriented design*).

Мы проанализируем формулировки типичных проблем, которые возникают при построении системы, определим, какие объекты нужно встраивать в систему, выясним, какие атрибуты должны иметь эти объекты, определим, какое поведение этих объектов должно быть описано, и определим, как объекты должны взаимодействовать друг с другом, чтобы соответствовать общесистемным задачам. Мы сделаем это прежде всего для проблем реального мира, а не для искусственных учебных примеров, и мы сделаем это до того, как вы научитесь писать объектно-ориентированные программы на C++. Когда мы доберемся до шестой главы, вы уже будете готовы начать разработку объектно-ориентированных систем на C++.

Мы начинаем со знакомства с некоторыми ключевыми терминами объектной ориентации. Посмотрите вокруг себя на реальный мир. Всюду, куда бы вы ни бросили взгляд, — *объекты!* Люди, животные, растения, автомобили, самолеты, здания, сенокосилки, компьютеры и тому подобное. Чело-

вечество думает в терминах объектов. Мы обладаем удивительной способностью *абстрагирования*, которая позволяет нам наблюдать картинки на экране (людей, самолеты, деревья и горы) именно в виде объектов, а не отдельных цветовых точек. Мы можем, если пожелаем, думать в терминах пляжа, а не песка, леса, а не деревьев, домов, а не кирпичей.

Мы могли бы склониться к тому, чтобы разделить все объекты на две категории — оживленные объекты и неоживленные. Оживленные объекты — «живые» в некотором смысле. Они передвигаются и выполняют действия. Неоживленные объекты, подобно бревнам, кажутся полными бездельниками. Они — своего рода «сидячее окружение». Однако все эти объекты имеют нечто общее. Все они имеют *атрибуты*, такие как размер, форма, цвет, вес и тому подобное. Все они проявляют какое-то *поведение*, например, мяч катится, подпрыгивает, надувается и спускается; ребенок кричит, спит, ползает, шагает и мигает глазами; автомобиль — разгоняется, поворачивает, тормозит и т.д.

Человечество узнает об объектах путем изучения их атрибутов и наблюдения их поведения. Разные объекты могут иметь много одинаковых атрибутов и представлять похожие черты поведения. Например, можно провести сравнение между детьми и взрослыми, людьми и шимпанзе. Легковые автомобили, грузовики, маленькие красные вагоны и роликовые коньки имеют много общего.

Объектно-ориентированное программирование (ООП) моделирует объекты реального мира с помощью программных аналогов. Это приводит к появлению отношений *классов*, когда объекты определенного класса — такого, как класс средств передвижения — имеют одинаковые характеристики. Это выдвигает отношения *наследования* и даже отношения *множественного наследования*, когда вновь создаваемые классы приобретают наследуемые характеристики существующих классов, а также содержат свои собственные уникальные характеристики. Объекты класса автомобилей с откидным верхом определенно имеют характеристики класса автомобилей, но крыша у них откладывается и закрывается.

Объектно-ориентированное программирование дает нам наиболее естественный и интуитивный способ рассмотрения процесса программирования как *моделирования* реально существующих объектов, их атрибутов и поведения. ООП моделирует также связи между объектами. Подобно тому, как люди посылают друг другу *сообщения* (например, сержант, командующий группе стоять смирно), объекты тоже связываются друг с другом посредством сообщений.

ООП инкапсулирует данные (атрибуты) и функции (способы поведения) в пакеты, называемые *объектами*; данные и функции объектов тесно взаимосвязаны. Объекты имеют свойство *скрытия информации*. Это означает, что хотя объекты могут знать, как связаться друг с другом посредством хорошо определенных интерфейсов, они обычно не могут узнать, как реализованы другие объекты — детали реализации спрятаны внутри самих объектов. Несомненно, можно эффективно ездить на автомобиле, не зная деталей того, как работают внутри него мотор, трансмиссия и система выхлопа. Мы увидим, почему скрытие информации так важно для разработки хорошего программного обеспечения.

В С и других языках *процедурного программирования* программирование имеет тенденцию быть *ориентированным на действия*, тогда как в С++ программирование стремится быть *ориентированным на объекты*. В С единицей программирования является *функция*. В С++ единицей программирования

является *класс*, на основе которого в конечном счете создаются *экземпляры* объектов.

Программисты, использующие С, сосредоточены на написании функций. Группы действий, выполняющих некоторую общую задачу, формируются в виде функций, а функции группируются так, чтобы сформировать программу. Данные конечно важны в С, но существует мнение, что данные предназначены в первую очередь для поддержки выполняемых функциями действий. Глаголы в объявлении системы помогают программисту на С при разработке системы определить набор функций, которые, работая совместно, и обеспечивают функционирование системы.

Программисты на С++ сосредоточены на создании своих собственных *определеняемых пользователем типов*, называемых *классами*. Каждый класс содержит данные и набор функций, которые манипулируют этими данными. Компоненты данных класса называются *данными-элементами*. Компоненты функции класса называются *функциями-элементами*. Точно так же, как экземпляр встроенного типа, такого как `int`, называется *переменной*, экземпляр определенного пользователем типа (т.е. класса) называется *объектом*. Программист использует встроенные типы как блоки для конструирования определенных пользователем типов. В С++ внимание фокусируется скорее на объектах, чем на функциях. Имена существительные в описании системы помогают программисту на С++ при создании системы определить набор классов, из которых будут созданы объекты, которые, работая совместно, и обеспечивают функционирование системы.

В этой главе мы познакомились с одной из управляющих структур С++, а именно, со структурой `if`. Во второй главе мы обсудим шесть оставшихся управляющих структур и объясним, как формировать структурированные программы путем объединения управляющих структур с различными операторами действий. Мы обнаружим, что для формирования любой мыслимой программы на С++ необходимы только три разных типа управляющих структур и только два различных способа их комбинирования. В конце второй главы мы продолжим наше первоначальное знакомство с объектной ориентацией, определяя объекты, которые будут необходимы для разработки систем реального мира.

Резюме

- Компьютер — это прибор, способный производить вычисления и принимать логические решения в миллионы или даже миллиарды раз быстрее человека.
- Компьютеры обрабатывают данные под управлением компьютерных программ.
- Различные устройства (такие как экран, диски, клавиатура, память и процессор) которые входят в состав компьютера, называются аппаратными средствами.
- Программы, которые выполняет компьютер, называются программным обеспечением.
- Входной блок является «воспринимающей» частью компьютера. Основная информация вводится сегодня в компьютеры с помощью клавиатур, подобных пишущей машинке.

- Выходной блок является частью компьютера, «транспортирующей» информацию на устройства вывода. Большая часть информации выводится из компьютеров путем отображения на экране или печати на бумаге.
- Блок памяти является «складирующей» частью компьютера и его часто называют либо памятью, либо первичной памятью.
- Арифметико-логическое устройство (АЛУ) выполняет вычисления и принимает решения.
- Не используемые активно другими устройствами программы и данные обычно помещаются во вспомогательных запоминающих устройствах (таких как диски) до тех пор, пока они снова не будут нужны.
- В режиме однопользовательской пакетной обработки компьютер выполняет одновременно единственную программу, а обрабатываемые данные находятся в группах или пакетах.
- Операционные системы — это системы программного обеспечения, которые делают использование компьютеров более удобным и обеспечивают более высокую производительность компьютеров.
- Мультипрограммные операционные системы предоставляют возможность «одновременного» выполнения многих заданий на компьютере — компьютер разделяет свои ресурсы между этими заданиями.
- Разделение времени — это специальный случай мультипрограммной системы, когда пользователи имеют доступ к компьютеру через терминалы. Создается видимость одновременного выполнения программы пользователя.
- При распределенных вычислениях расчеты организации распределены путем подключения сети к местам, где выполняется реальная работа организации.
- Файл-серверы хранят программы и данные, которые могут быть использованы распределенными по сети компьютерами-клиентами, отсюда термин — вычисления на платформе клиент/сервер.
- Любой компьютер может непосредственно понимать только свой собственный машинный язык.
- Машинные языки в общем случае содержат ряды чисел (в конечном счете сведенных к единицам и нулям), которые являются командами компьютеру на выполнение большинства элементарных операций в тот или иной момент времени. Машинные языки машинно-зависимы.
- Англо-подобные аббревиатуры образуют основу языков ассемблера. Ассемблеры транслируют языки ассемблера в машинные коды.
- Компиляторы транслируют языки высокого уровня в машинные коды. Языки высокого уровня содержат английские слова и общепринятую математическую нотацию.
- Интерпретирующие программы непосредственно выполняют программы на языках высокого уровня и не требуют их трансляции в машинный код.
- Хотя скомпилированные программы выполняются быстрее, чем интерпретируемые программы, интерпретаторы популярны в таких услови-

ях, когда программы часто перекомпилируются для добавления в них новых возможностей и исправления ошибок. Но когда программа разработана, ее скомпилированная версия будет выполняться более эффективно.

- С известен как язык разработки операционной системы UNIX.
- На С и С++ можно писать программы, которые переносимы на большинство компьютеров.
- FORTRAN (FORmula TRANslator — транслятор формул) используется для инженерных приложений.
- COBOL (COmmon Business Oriented Language — язык, ориентированный на задачи бизнеса) прежде всего использовался для решения коммерческих задач, требующих точной и эффективной обработки больших объемов данных.
- Структурное программирование — дисциплинированный подход к написанию программ, отличающихся от неструктурированных программ ясностью, простотой тестирования и отладки и легкостью модификации.
- Язык Паскаль был предназначен для изучения структурного программирования в академической среде.
- Язык Ада был разработан под патронажем Министерства Обороны США на основе языка Паскаль.
- Многозадачность позволяет программисту описывать параллельные действия.
- Все системы С++ состоят из трех частей: среды программирования, языка и стандартных библиотек. Библиотечные функции не входят в состав языка С++; эти функции выполняют такие операции как ввод-вывод и математические вычисления.
- Как правило, программа на С++ должна пройти шесть этапов для своего выполнения: редактирование, предварительную (препроцессорную) обработку, компиляцию, компоновку, загрузку и выполнение.
- Программист набирает программу на С++ с помощью редактора и вносит исправления, если это необходимо. Имена файлов С++ в типичной основанной на UNIX системе оканчиваются расширением .С.
- Компилятор компилирует программу на С++ в машинный код (или объектный код).
- Препроцессор выполняет директивы препроцессора, которые обычно указывают, что в компилируемый файл должны быть включены другие файлы, а также обрабатывает некоторые специальные символы, размещенные в тексте программы.
- Компоновщик связывает объектный код с кодами отсутствующих функций, чтобы создать исполняемый загрузочный модуль (без пропущенных частей). В типовой основанной на UNIX системе С++ команда компиляции и компоновки программы обозначается как СС. Если программа скомпилировалась и скомпоновалась правильно, будет создан файл, называемый a.out. Это и есть исполняемый загрузочный модуль программы.

- Загрузчик считывает выполняемый модуль с диска и передает его в память.
- Компьютер под управлением своего ЦПУ выполняет программу по одной команде в каждый момент времени.
- Ошибки, подобные делению на нуль, возникают во время выполнения программы, поэтому эти ошибки называются ошибками прогона или ошибками выполнения.
- Ошибка деления на нуль обычно является неисправимой ошибкой, т.е. ошибкой, вызывающей немедленное прекращение работы программы и исключающей ее нормальное выполнение. Исправимые ошибки позволяют программе продолжать счет до его завершения, часто приводя к неправильным результатам.
- Определенные функции C++ выполняют ввод из `cin` (the standard input stream — стандартный поток ввода), т.е. обычно с клавиатуры, но `cin` может быть связан и с другим устройством. Вывод данных производится в `cout` (the standard output stream — стандартный поток вывода), т.е. обычно на экран, но `cout` может быть связан и с другим устройством.
- Стандартный поток ошибок (the standard error stream) обозначается как `cerr`. Поток `cerr` (обычно связанный с экраном) используется для отображения сообщений об ошибках.
- Существует множество проблем, связанных с различными версиями C++ и разными компьютерами, которые могут сделать мобильность сомнительной.
- C++ обеспечивает возможность объектно-ориентированного программирования.
- Объекты — это фактически повторно используемые компоненты программного обеспечения, моделирующие элементы реального мира.
- Однострочный комментарий начинается с символа `//`. Программисты вставляют комментарии, чтобы документировать программу и улучшить ее читаемость. Комментарии не вызывают никаких действий компьютера при выполнении программы.
- Стока `#include <iostream.h>` дает указание препроцессору C++ включить в программу содержание головного файла потоков ввода-вывода. Этот файл содержит информацию, необходимую, чтобы компилировать программу, которая использует `cin` и `cout`.
- Обычно программа на C++ начинает выполнение с функции `main`.
- Объект потока вывода `cout`, обычно подключенный к экрану, используется для вывода данных. Множество элементов данных может быть выведено с помощью сцепленной операции поместить в поток (`<<`).
- Объект потока ввода `cin`, обычно подключенный к клавиатуре, используется для ввода данных. Множество элементов данных может быть введено с помощью сцепленной операции взять из потока (`>>`).
- Все переменные в программе на C++ должны быть объявлены перед тем, как они могут быть использованы.

- Имя переменной в C++ — это любой допустимый идентификатор. Идентификатором называется последовательность символов, содержащая буквы, цифры и символы подчеркивания (_). Идентификатор не может начинаться с цифры. В C++ допускаются идентификаторы любой длины, но некоторые системы и среды C++ могут налагать некоторые ограничения на длину идентификаторов.
- C++ чувствителен к регистру.
- Большинство вычислений выполняется в операторах присваивания.
- Каждая определенная переменная, хранящаяся в памяти компьютера, имеет имя, значение, тип и размер.
- Каждый раз, когда новое значение помещается в область памяти, оно замещает предыдущее значение в этой области. Предыдущее значение уничтожается.
- Процесс считывания значения из памяти — не деструктивный, т.е. считывается копия значения, а оригинал значения остается в памяти нетронутым.
- C++ вычисляет арифметические выражения в четкой последовательности, определяемой правилами приоритета и ассоциативности операций.
- Оператор if позволяет программе принимать решения при выполнении определенного условия. Оператор if имеет следующий формат

```
if (условие)
    оператор;
```

Если условие истинно, оператор в теле if выполняется. Если условие не удовлетворяется, т.е. ложно, тело оператора пропускается.

- Условия в операторах if обычно формируются с помощью операций проверки на равенство и отношения. Результат использования этих операций всегда или «истина — true» или «ложь — false».
- Объектная ориентация — это естественный способ размышлений о мире и написания компьютерных программ.
- Все объекты имеют атрибуты, такие как размер, форма, цвет, вес и тому подобное. И все они имеют различные способы поведения.
- Человечество узнает об объектах, изучая их атрибуты и наблюдая их поведение.
- Различные объекты могут иметь много одинаковых атрибутов и похожих черт поведения.
- Объектно-ориентированное программирование (ООП) моделирует объекты реального мира с помощью программных аналогов. Это приводит к появлению отношений классов, когда объекты определенного класса имеют одинаковые характеристики. В этом польза отношения наследования и даже отношения множественного наследования, когда вновь создаваемые классы приобретают наследственные характеристики существующих классов, а также имеют и свои собственные уникальные характеристики.
- Объектно-ориентированное программирование обеспечивает интуитивный способ рассмотрения процесса программирования как моделирования реально существующих объектов, их атрибутов и поведения.

- ООП моделирует также связи между объектами посредством сообщений.
- ООП инкапсулирует в объекте данные (атрибуты) и функции (способы поведения).
- Объекты имеют свойство скрытия информации. Это означает, что хотя объекты могут знать, как связаться друг с другом посредством хорошо определенных интерфейсов, они обычно не могут узнать, как реализованы другие объекты.
- Скрытие информации важно для разработки хорошего программного обеспечения.
- В С и других процедурных языках программирования программирование имеет тенденцию быть ориентированным на действия. Данные конечно важны в С, но существует мнение, что данные предназначены в первую очередь для поддержки действий, выполняемых функциями.
- Программисты на С++ сосредоточены на создании своих собственных определяемых пользователем типов, называемых классами. Каждый класс содержит данные и набор функций, которые манипулируют этими данными. Компоненты данных класса называются данными-элементами. Компоненты функций класса называются функциями-элементами.

Терминология

ANSI C

main

абстрагирование

арифметико-логическое

устройство (АЛУ)

арифметическая операция

ассоциативность операций

ассоциативность слева направо

ассоциативность справа налево

атрибуты объекта

библиотека классов С++

бинарная операция

ввод-вывод (I/O)

вложенные скобки

данные

данные-элемент

действие

загрузка

зарезервированные слова

знак перехода (\)

значение переменной

идентификатор

имя переменной

инкапсуляция

интерпретатор

интерфейс

исправимая ошибка

класс

комментарий (//)

компилятор

компоновка

компьютер

компьютерная программа

круглые скобки ()

логическая ошибка

машинно-зависимый

машинный язык

область в памяти

многозадачность

множественное наследование

моделирование

мультипрограммная система

мультипроцессор

наследование

неисправимая ошибка

объект

объектно-ориентированное

программирование (ООП)

объектно-ориентированное

проектирование (OOD)

объектное мышление

объектный язык

объявление

операнд

оператор	C
операции отношения	C++
> больше	символ новой строки (\n)
< меньше	символы пустого пространства
>= больше или равно	синтаксическая ошибка
<= меньше или равно	скрытие информации
операции проверки на равенство	сообщение
== равно	стандартная библиотека Си
!= не равно	стандартный поток ввода (cin)
операция	стандартный поток вывода (cout)
операция вычисления остатка (%)	стандартный поток ошибок (cerr)
операция присваивания (=)	строка
операция умножения (*)	структурная if
определеняемый пользователем тип	структурное программирование
ошибка во время выполнения	тело функции
ошибка во время компиляции	терминал
ошибка времени выполнения	техническое обеспечение
ошибка компиляции	точка с запятой (;) как
память	признак конца оператора
первичная память	управляющая последовательность
переменная	управляющая логика
платформа клиент/сервер	условие
поведение объекта	устройство ввода
повторное использование программного обеспечения	устройство вывода
правила старшинства операций	файл-сервер
препроцессор	функция
приглашение	функция-элемент
признак конца оператора (;)	целое (int)
приоритет	целочисленное деление
программа транслятор	центральное процессорное устройство (ЦПУ)
программное обеспечение	чувствительность к регистру
процедурное программирование	язык ассемблера
распределенные вычисления	язык высокого уровня
редактор	язык программирования
решение	

Типичные ошибки программирования

- 1.1. Ошибки, подобные делению на нуль, возникают во время выполнения программы, поэтому эти ошибки называются ошибками прогона или ошибками выполнения. Ошибка деления на нуль обычно является неисправимой ошибкой, т.е. ошибкой, вызывающей немедленное прекращение работы программы и исключающей ее нормальное выполнение. Исправимые ошибки позволяют программе продолжать счет до его завершения, часто приводя к неправильным результатам.
- 1.2. Забывают включить файл `iostream.h` в программу, которая вводит данные с клавиатуры или выводит данные на экран.
- 1.3. Пренебрежение точкой с запятой в конце оператора является синтаксической ошибкой. Синтаксическая ошибка возникает тогда, когда компилятор не может распознать оператор. Обычно компилятор выдает сообщение об ошибке, чтобы помочь программисту

локализовать и исправить неверный оператор. Синтаксические ошибки — это нарушения правил языка. Синтаксические ошибки называются также ошибками компиляции или ошибками во время компиляции, потому что они обнаруживаются на этапе компиляции.

- 1.4. Попытка использования операции вычисления остатка `%` с нецелочисленными operandами является синтаксической ошибкой.
- 1.5. Если операции `==`, `!=`, `>=` и `<=` содержат между своими символами пробелы, это — синтаксическая ошибка.
- 1.6. Перестановка символов в обозначении операций: вместо `!=`, `>=` и `<=` запись `=!`, `=>` и `=<` соответственно. Это вызывает синтаксическую ошибку.
- 1.7. Смешивание операции проверки на равенство `==` с операцией присваивания `=`. Операция проверки на равенство должна читаться как «равно», а операция присваивания должна читаться как «присваивает». Некоторые предпочитают читать операцию проверки на равенство как «двойное равенство». Как мы вскоре увидим, смешивание этих операций может вызывать нелегко распознаваемую синтаксическую ошибку, а может вызвать и чрезвычайно запутанные логические ошибки.
- 1.8. Постановка точки с запятой после правой круглой скобки, завершающей условие в структуре `if`. Такая ошибочная точка с запятой приводит к тому, что тело структуры `if` становится пустым, так что сама структура `if` не выполняет никаких действий, независимо от того, истинно условие или нет. Более того, подлинный оператор тела структуры `if` превращается в оператор, следующий за структурой `if`, и выполняется всегда.

Хороший стиль программирования

- 1.1. Пишите программы на C++ в простом и четком стиле. Об этом иногда говорят как о KIS («keep it simple» — придерживайтесь простоты). Не «насильуйте» язык попытками его причудливого использования.
- 1.2. Читайте руководства для той версии C++, которой вы пользуетесь. Чаще обращайтесь к этим руководствам, чтобы быть уверенными в знании богатого набора возможностей C++ и в том, что вы правильно пользуетесь этими возможностями.
- 1.3. Ваши компьютер и компилятор — хорошие учителя. Если после тщательного чтения руководства по той версии C++, которой вы пользуетесь, у вас нет уверенности в правильном понимании того, как реализуется та или иная особенность C++, экспериментируйте и смотрите, что происходит. Установите опции вашего компилятора в режим «предупреждений о максимальном числе ситуаций». Изучайте каждое сообщение, поступающее к вам при компиляции ваших программ и исправляйте программы до исчезновения этих сообщений.
- 1.4. Каждая программа должна начинаться с комментария, описывающего цель программы.

- 1.5. Последний символ, печатаемый любой функцией печати, должен быть \n (новая строка). Это дает уверенность в том, что функция переместит курсор в начало новой строки. Соглашения подобного рода способствуют возможности повторного использования программного обеспечения — ключевой задаче любой среды разработки программного обеспечения.
- 1.6. Делайте одинаковые отступы для всего тела каждой функции внутри фигурных скобок, определяющих тело функции. При этом функциональная структура программы получается более понятной и легкой для чтения.
- 1.7. Установите соглашение о величине желательного отступа и затем везде придерживайтесь этого соглашения. Для создания отступа можно использовать клавишу табуляции, а позиции табуляции можно изменять. Мы рекомендуем для формирования величины отступа примерно 0,5 см или (предпочтительнее) три пробела.
- 1.8. Ставьте пробел после каждой запятой (,), чтобы программу было легче читать.
- 1.9. Выбор осмысленных имен переменных помогает программе быть «самодокументируемой», т.е. такую программу легче понимать при чтении, даже не обращаясь к справочным пособиям или обширным комментариям.
- 1.10. Избегайте идентификаторов, которые начинаются с подчеркиваний, потому что компилятор C++ может использовать похожие на них имена для своих собственных внутренних целей. Это предотвратит путаницу в именах, выбираемых вами и компилятором.
- 1.11. Всегда помещайте пустую строку перед объявлением, которое находится между выполняемыми операторами. Это делает объявления заметными в программе и способствует ее четкости.
- 1.12. Если вы предпочитаете размещать объявления в начале функции, отделяйте эти объявления от выполняемых операторов в этой функции пустой строкой, чтобы выделить конец объявлений и начало выполняемых операторов.
- 1.13. Размещайте пробелы с обеих сторон бинарной операции. Это выделит операцию и улучшит читаемость программы.
- 1.14. Отступы в теле структуры if выделяют тело структуры и упрощают чтение программы.
- 1.15. В каждой строке программы должно быть не более одного оператора.
- 1.16. Оператор может занимать несколько строк. Если один оператор должен быть разбит на несколько строк, выбирайте характерные точки разбиения, такие, как позиция после запятой в списке с запятыми в качестве разделителя или позиция после операции в длинном выражении.
- 1.17. Обращайтесь к таблице старшинства операций при написании выражений, содержащих много операций. Убедитесь, что операции в выражениях выполняются в ожидаемой вами последовательности.

Если вы не уверены в последовательности выполнения в сложных выражениях, используйте для задания последовательности круглые скобки так же, как вы это делаете в алгебраических выражениях. Убедитесь путем наблюдений, что некоторые операции, такие как присваивание (`=`), имеют ассоциативность не слева направо, а справа налево.

Советы по повышению эффективности

1. Использование функций стандартной библиотеки ANSI вместо написания собственных версий тех же функций может повысить эффективность программ, поскольку эти функции написаны специально с учетом эффективности их выполнения.

Замечания по мобильности

- 1.1. Поскольку C — стандартизованный, аппаратно-независимый, широко доступный язык, приложения, написанные на C, часто могут выполняться с минимальными модификациями или даже без них на самых различных компьютерных системах.
- 1.2. Использование функций стандартной библиотеки ANSI вместо написания собственных версий тех же функций может повысить мобильность программ, поскольку эти функции включены практически во все реализации C++.
- 1.3. Написать мобильную (переносимую) программу можно, но для разных компиляторов C и C++ и для разных компьютеров возникнет множество проблем, которые могут сделать мобильность труднодостижимой. Само по себе написание программ на C и C++ еще не обеспечивает мобильность. Программист будет вынужден часто считаться с особенностями разновидностей компиляторов и компьютеров.

Замечания по технике программирования

- 1.1. Используйте для создания программ методологию стандартных блоков. Избегайте заново изобретать колесо. Используйте существующие кусочки — это называется повторным использованием программного обеспечения и служит основой объектно-ориентированного программирования.
- 1.2. При программировании на C++ вы обычно будете использовать следующие стандартные блоки: классы из библиотек классов и функции из стандартной библиотеки ANSI C, классы и функции, созданные вами самими, и классы и функции, созданные другими людьми, но доступные вам.

Упражнения для самопроверки

- 1.1. Заполнить пробелы в следующих утверждениях:
 - а) Компания, которая популяризировала персональные вычисления, была _____.

- b) Компьютер, который сделал персональные вычисления признанными в бизнесе и промышленности, был _____.
- c) Компьютеры обрабатывают данные под управлением наборов команд называемых компьютерными _____.
- d) Шестью ключевыми логическими блоками компьютера являются _____, _____, _____, _____, _____ и _____.
- e) Тремя классами языков, рассмотренных в этой главе, являются _____, _____ и _____.
- f) Программы, которые транслируют программы на языках высокого уровня в машинные языки, называются _____.
- g) С широко известен как язык создания операционной системы _____.
- h) Язык _____ был разработан Виртом для изучения структурного программирования в университетах.
- i) Министерство обороны США разработало язык Ада, обладающий возможностью, называемой _____, что позволяет программистам определять множество действий, выполняющихся параллельно.

1.2. Заполните пустые места в следующих утверждениях о среде программирования C++.

- a) Программа на C++ обычно вводится в компьютер с помощью программы _____.
- b) В системе C++ перед началом этапа компиляции выполняется программа _____.
- c) Программа _____ объединяет результат работы компилятора с различными библиотечными функциями, чтобы создать исполняемый загрузочный модуль.
- d) Программа _____ загружает исполняемый модуль с диска в память.

1.3. Заполнить пробелы в следующих утверждениях:

- a) Выполнение каждой программы на C++ начинается с функции _____.
- b) _____ начинает тело каждой функции, а _____ заканчивает тело каждой функции.
- c) Каждый оператор заканчивается _____.
- d) Управляющая последовательность \n представляет символ _____, который вызывает перемещение курсора к началу следующей строки на экране.
- e) Оператор _____ используется для принятия решений.

1.4. Укажите, что из нижеследующего верно или неверно. Если неверно, то объясните, почему.

- a) Комментарии вызывают печать компьютером на экране текста после символа // при выполнении программы.

- b) Если вывод осуществляется в `cout`, то последовательность вывода `\n` вызывает перемещение курсора к началу следующей строки на экране.
- c) Все переменные должны быть объявлены до того, как они используются.
- d) Всем переменным, когда они объявляются, должен быть присвоен тип.
- e) C++ рассматривает переменные `number` и `NuMbEr` как одинаковые.
- f) Объявления в теле функции C++ могут появляться почти везде.
- g) Операция вычисления остатка (%) может быть использована только с целыми числами.
- h) Все арифметические операции *, /, %, + и — имеют одинаковый уровень приоритета.
- i) Программа на C++, которая выводит три строки, должна содержать три оператора вывода, использующих `cout`.
- 1.5. Напишите один оператор C++, соответствующий следующему:
- Объявите переменные `c`, `thisisAVariable`, `q76354` и `number` типа `int`.
 - Предложите пользователю ввести целое число. Закончите сообщение о вашем приглашении двоеточием (:), за которым следует пробел, и установите курсор после пробела.
 - Прочтите целое число с клавиатуры и запомните введенное значение в целой переменной `age`.
 - Если переменная `number` не равна 7, напечатайте «Значение переменной `number` не равно 7.».
 - Напечатайте сообщение «Это программа на C++.» на одной строке.
 - Напечатайте сообщение «Это программа на C++.» на двух строках, где первая строка заканчивается на «программа».
 - Напечатайте сообщение «Это программа на C++.» так, чтобы на каждой строке было только одно слово.
 - Напечатайте сообщение «Это программа на C++» так, чтобы каждое слово было отделено от следующего знаком табуляции.
- 1.6. Напишите операторы или комментарии, соответствующие следующему:
- Заявить, что программа будет вычислять произведение трех целых чисел.
 - Объявить переменные `x`, `y`, `z` и `results` типа `int`.
 - Предложить пользователю ввести три целых числа.
 - Прочитать три целых числа с клавиатуры и сохранить их в переменных `x`, `y` и `z`.
 - Вычислить произведение трех целых чисел, содержащихся в переменных `x`, `y` и `z`, и присвоить результат переменной `result`.

- f) Напечатать «Произведение равно » и потом значение переменной **result**.
- g) Возвратить из функции **main** значение, свидетельствующее об успешном завершении программы.
- 1.7. Используя написанные в упражнении 1.6 операторы, напишите полную программу, которая рассчитывает и печатает произведение трех чисел.
- 1.8. Укажите и исправьте ошибки в каждом из следующих операторов:
- a) if (c < 7);
 cout << "с меньше 7\n";
- b) if (c => 7);
 cout << "с равно или больше 7\n";
- 1.9. Заполните пустые места терминами «языка объектов»:
- а) Люди могут посмотреть на телевизор и увидеть цветные точки, или они могут сделать шаг назад и увидеть трех людей, сидящих за столом конференции; это пример способности, называемой _____.
- б) Если вы рассматриваете автомобиль как объект, тот факт, что у автомобиля откидной верх, является атрибутом или чертой поведения (указать одно) _____ автомобиля.
- с) Факты, что автомобиль может разгоняться и тормозить, поворачивать направо и налево, ехать вперед или назад являются примерами _____ объекта «автомобиль».
- д) Восприятие новым типом класса характеристик нескольких разных типов существующих классов называется _____ наследованием.
- е) Объекты связываются, посылая друг другу _____.
- ф) Объекты связываются друг с другом посредством хорошо определенного _____.
- г) Каждый объект обычно не может узнать, как проектируется другой объект; это свойство называется _____.
- х) _____ в описании системы помогает программисту на C++ определять классы, которые будут нужны для проектирования системы.
- и) Компоненты данных класса называются _____, а компоненты функций класса называются _____.
- ж) Экземпляр определенного пользователем типа называется _____.

Ответы на упражнения для самопроверки

- 1.1. а) Apple. б) IBM PC. в) программами. г) блок ввода, блок вывода, блок памяти, арифметико-логическое устройство, центральное процессорное устройство, блок вспомогательных запоминающих устройств. е) машинные языки, языки ассемблера, языки высокого

- уровня. f) компиляторами. g) UNIX. h) Паскаль. i) многозадачностью.
- 1.2. a) редактор. b) препроцессор. c) компоновщик. d) загрузчик.
- 1.3. a) main. b) Левая фигурная скобка ({), правая фигурная скобка (}).
c) точкой с запятой. d) новая строка. e) if.
- 1.4. a) Неверно. Комментарии не вызывают каких-либо действий при выполнении программы. Они используются для документирования программы и улучшения ее читаемости.
b) Верно.
c) Верно.
d) Верно.
e) Неверно. C++ — чувствителен к регистру, так что эти переменные различны.
f) Верно.
g) Верно.
h) Неверно. Операции *, / и % имеют одинаковый уровень приоритета, а операции + и — имеют более низкий уровень.
i) Неверно. Один оператор вывода, использующий cout и содержащий несколько символов новой строки \n, может напечатать несколько строк.
- 1.5. a) int c, thisIsAVariable, q76354, number;
b) cout << "Введите целое число: ";
c) cin >> age;
d) if (number != 7)
 cout << "Значение переменной не равно 7.\n";
e) cout << "Это программа на C++.\n";
f) cout << "Это программа\nна C++.\n";
g) cout << "Это\nпрограмма\nна\nC++.\n";
h) cout << "Это\tпрограмма\tна\tC++.\n";
- 1.6. a) //Вычисление произведения трех целых чисел
b) int x, y, z, result;
c) cout << "Введите три целых числа: ";
d) cin >> x >> y >> z;
e) result = x * y * z;
f) cout << "Произведение равно " << result << '\n';
g) return 0;
- 1.7. // Вычисление произведения трех целых чисел
#include <iostream.h>
- ```
main()
{
```

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

cout << "Введите три целых числа: ";
cin >> x >> y >> z;
result = x * y * z;
cout << "Произведение равно " << result << '\n';

return 0;
}
```

- 1.8. a) Ошибка: точка с запятой после правой круглой скобки условия в операторе if. Исправление: удалите точку с запятой после правой круглой скобки. Замечание: в результате этой ошибки оператор вывода будет выполняться независимо от истинности условия в операторе if. Точка с запятой после правой круглой скобки считается пустым оператором — оператором, который ничего не делает. Мы узнаем больше о пустом операторе в следующей главе.
- b) Ошибка: операция сравнения =>. Исправление: измените => на >=.
- 1.9. a) абстрагирование. b) атрибутом. c) поведения. d) множественным. e) сообщения. f) интерфейсы. g) скрытие информации. h) имена существительные. i) данными-элементами, функциями-элементами. j) объект.

## Упражнения

- 1.10. Отнесите следующие элементы к категории аппаратных средств или программного обеспечения:
- a) ЦПУ
  - b) компилятор C++
  - c) АЛУ
  - d) препроцессор C++
  - e) блок ввода
  - f) программа редактор
- 1.11. Почему вам хотелось бы писать программу на машинно-независимом языке вместо машинно-зависимого? Почему машинно-зависимый язык может оказаться предпочтительней для написания программ определенного типа?
- 1.12. Заполните пустые места в каждом из следующих предложений:
- a) Какой логический блок компьютера принимает информацию извне для использования в компьютере? \_\_\_\_\_
  - b) Процесс составления инструкций компьютеру для решения специфических проблем называется \_\_\_\_\_.  
\_\_\_\_\_.
  - c) Какой тип компьютерного языка использует англо-подобные аббревиатуры для команд на машинном языке? \_\_\_\_\_.
  - d) Какой логический блок компьютера посылает уже обработанную компьютером информацию различным устройствам для использования вне компьютера? \_\_\_\_\_.

- е) Какой логический блок компьютера хранит информацию? \_\_\_\_\_.
- ф) Какой логический блок компьютера выполняет вычисления? \_\_\_\_\_.
- г) Какой логический блок компьютера принимает логические решения? \_\_\_\_\_.
- х) Уровень компьютерного языка, удобный программисту для быстрого и легкого написания программ — \_\_\_\_\_.
- и) Единственный язык, непосредственно понятный компьютеру, называется \_\_\_\_\_.
- ж) Какой логический блок компьютера координирует действия всех других логических блоков? \_\_\_\_\_.

1.13. Укажите смысл каждого из следующих объектов:

- а) `cin`  
б) `cout`  
в) `cerr`

1.14. Почему так много внимания уделяется сегодня объектно-ориентированному программированию вообще и в C++ в частности?

1.15. Заполните пустые места в каждом из следующих предложений:

- а) \_\_\_\_\_ используются для документирования программы и улучшения ее читаемости.
- б) Объект, используемый для вывода информации на экран, называется \_\_\_\_\_.
- с) Оператор C++, принимающий решение, называется \_\_\_\_\_.
- д) Вычисления обычно выполняются с помощью оператора \_\_\_\_\_.
- е) Объект \_\_\_\_\_ вводит информацию с клавиатуры.

1.16. Напишите один или несколько операторов C++, выполняющих указанные ниже задания:

- а) Напечатайте сообщение «Ведите два числа: »
- б) Присвойте произведение переменных `a` и `b` переменной `c`.
- с) Объявите, что программа выполняет расчеты платежных ведомостей (т.е. используйте текст помогающий документировать программу).
- д) Введите три целых значения с клавиатуры и поместите эти значения в целые переменные `a`, `b` и `c`.

1.17. Укажите, что из нижеследующего верно или неверно. Объясните ваши ответы.

- а) Операции в C++ выполняются слева направо.
- б) Все следующие далее имена переменных верны: `_under_bar_, m928134, t5, j7, her_sales, his_account_njnfk, a, b, c, z, z2`.
- с) Оператор `cout << "A = 5;"` — типичный пример оператора присваивания.

- d) Правильное арифметическое выражение на C++ без круглых скобок выполняется слева направо.  
e) Все следующие имена переменных правильные: 3g, 87, 67h2, h22, 2h.

**1.18.** Заполните следующие пустые места:

- a) Какие арифметические операции имеют такой же уровень приоритета как умножение? \_\_\_\_\_  
b) Какие из вложенных круглых скобок выполняются в арифметическом выражении первыми? \_\_\_\_\_  
c) Области в памяти компьютера, которые могут содержать разные значения в разное время в процессе выполнения программы, называются \_\_\_\_\_.

**1.19.** Что печатается, если это вообще возможно, при выполнении каждого из следующих операторов. Если ничего не печатается, то ответьте «ничего». Предполагайте, что  $x = 2$ ,  $y = 3$ .

- a) `cout << x;`  
b) `cout << x + x;`  
c) `cout << "x=";`  
d) `cout << x" = " << x;`  
e) `cout << x + y << " = " << y + x;`  
f) `z = x + y;`  
g) `cin >> x >> y;`  
h) // `cout << "x + y = " << x + y;`  
i) `cout << "\n";`

**1.20.** Какие из следующих операторов C++ содержат переменные, значения которых уничтожаются?

- a) `cin >> b >> c >> d >> e >> f;`  
b) `p = i + j + k = 7;`  
c) `cout << "переменные, значения которых уничтожаются"`  
d) `cout << "a = 5";`

**1.21.** Какие из следующих операторов C++ верны для уравнения  $y = ax^3 + 7$ :

- 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);`

**1.22.** Укажите порядок выполнения действий в каждом из следующих операторов C++ и назовите значения x после их выполнения:

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

- b)  $x = 2 \% 2 + 2 * 2 - 2 / 2;$   
c)  $x = (3 * 9 * (3 + (9 * 3 / (3))));$

**1.23.** Напишите программу, которая просит пользователя ввести два числа, получает числа от пользователя и затем печатает сумму, произведение, разность и частное этих чисел.

**1.24.** Напишите программу, которая печатает числа от 1 до 4 на одной и той же строке, так что соседние числа разделены одним пробелом. Напишите программу, используя следующие способы:

- a) Используя один оператор вывода с одним оператором поместить в поток.  
b) Используя один оператор вывода с четырьмя операторами поместить в поток.  
c) Используя четыре оператора вывода.

**1.25.** Напишите программу, которая просит пользователя ввести два числа, получает числа от пользователя и затем печатает большее число после слова «больше». Если числа равны, напечатайте сообщение «Эти числа равны».

**1.26.** Напишите программу, которая вводит три целых числа с клавиатуры и печатает сумму, среднее значение, произведение, меньшее и большее из этих чисел. Диалог на экране должен выглядеть следующим образом:

```
Введите три различных целых числа: 13 27 14
Сумма равна 54
Среднее значение равно 18
Произведение равно 4914
Наименьшее равно 13
Наибольшее равно 27
```

**1.27.** Напишите программу, которая считывает радиус круга и печатает диаметр круга, длину окружности и площадь. Используйте значение константы 3.14159 для числа  $\pi$ . Выполните эти вычисления в операторе вывода. (Замечание: В этой главе мы обсудили только целые константы и переменные. В главе 3 мы обсудим числа с плавающей запятой, т.е. величины, которые могут иметь десятичную запятую).

**1.28.** Напишите программу, которая печатает прямоугольник, овал, стрелу и ромб в следующем виде:

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

**1.29.** Что печатает следующий оператор?

```
cout << "*\n**\n***\n****\n*****\n";
```

- 1.30.** Напишите программу, которая считывает пять целых чисел, определяет и печатает наибольшее и наименьшее из них. Используйте только те приемы программирования, которые вы изучили в этой главе.
- 1.31.** Напишите программу, которая считывает целое число, определяет и печатает, четное оно или нечетное. (Подсказка: Используйте операцию вычисления остатка. Четное число кратно двум. Любое число, кратное двум, при делении на 2 дает в остатке нуль.).
- 1.32.** Напишите программу, которая считывает два целых числа, определяет и печатает, является ли первое число кратным второму. (Подсказка: используйте операцию вычисления остатка)
- 1.33.** Отобразите модель шахматной доски восемью операторами вывода и затем отобразите ту же модель наименьшим возможным количеством операторов вывода:

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

- 1.34.** Укажите различие между терминами неисправимая и исправимая ошибка. Почему вы предпочитаете получить неисправимую ошибку, чем исправимую?

- 1.35.** Заглянем вперед. В этой главе вы узнали о целых числах и типе `int`. C++ может также представлять прописные и строчные буквы и значительное многообразие специальных символов. Для представления каждого отдельного символа C++ использует небольшие целые числа. Компьютер использует множество символов и соответствующие целые представления для этих символов называются набором символов компьютера. Вы можете напечатать символ, просто заключив его в одиночные кавычки

```
cout << 'A';
```

Вы можете напечатать целочисленный эквивалент символа, записав перед ним (`int`) — это называется приведением к типу (более подробно о приведении к типу мы поговорим в главе 2).

```
cout << (int) 'A';
```

Когда выполняется предшествующий оператор, он печатает значение 65 (в системе, которая использует так называемый набор символов ASCII). Напишите программу, которая печатает целочисленные эквиваленты ряда прописных и строчных букв, цифр и специальных символов. Как минимум, определите целочисленные эквиваленты следующих символов: A B C a b c 0 1 2 \$ \* + / и пробела.

- 1.36.** Напишите программу, которая вводит число из пяти цифр, разделяет число на отдельные цифры и печатает их отдельно друг от друга с тремя пробелами между ними. Например, если пользователь вводит в программу 42339, то должно быть напечатано

4 2 3 3 9

- 1.37.** Используя только технику программирования, изученную в этой главе, напишите программу, которая вычисляет квадрат и куб чисел от 0 до 10 и использует табуляцию для печати следующей таблицы значений:

| число | квадрат | куб  |
|-------|---------|------|
| 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 |

- 1.38.** Дайте краткий ответ на каждый из следующих вопросов «объектного мышления»:

- а) Почему детальное обсуждение структурного программирования в этой книге предшествует глубокому рассмотрению объектно-ориентированного программирования?
- б) Каковы упомянутые в книге типовые шаги процесса объектно-ориентированного проектирования?
- в) Как множественное наследование отражено в жизни человека?
- г) Какого рода сообщения люди посылают друг другу?
- д) Объекты посылают сообщения друг другу через хорошо определенный интерфейс. Какой интерфейс предоставляет радио автомобиля (объект) своему пользователю (объекту человек)?

- 1.39.** Вы, возможно, носите на запястье один из наиболее типичных во всем мире объектов — часы. Подумайте, как каждый из следующих терминов и концепций приложимы к понятию «часы»: объект, атрибуты, поведение, класс, наследование (рассмотрите, например, будильник), абстракция, моделирование, сообщения, инкапсуляция, интерфейс, скрытие информации, данные-элементы и функции-элементы.

# 2

## Управляющие структуры



### Ц е л и

- Понять основные проблемы технологии принятия решений.
- Научиться создавать алгоритмы по методологии нисходящей разработки с пошаговой детализацией.
- Научиться использовать структуры выбора **if**, **if/else** и **switch** для выбора альтернативного варианта действий.
- Научиться использовать структуры повторения **while**, **do/while** и **for** для повторного выполнения операторов программы.
- Понять повторения, управляемые счетчиком и меткой.
- Научиться использовать операции инкремента, декремента, присваивания и логические операции.
- Научиться использовать операторы программного управления **break** и **continue**.

## План

- 2.1. Введение
- 2.2. Алгоритмы
- 2.3. Псевдокод
- 2.4. Управляющие структуры
- 2.5. Структура выбора *if* (ЕСЛИ)
- 2.6. Структура выбора *if/else* (ЕСЛИ-ИНАЧЕ)
- 2.7. Структура повторения *while* (ПОКА)
- 2.8. Разработка алгоритмов: учебный пример 1 (повторение, управляемое счетчиком)
- 2.9. Нисходящая разработка алгоритмов с пошаговой детализацией: учебный пример 2 (повторение, управляемое меткой)
- 2.10. Нисходящая разработка алгоритмов с пошаговой детализацией: учебный пример 3 (вложенные управляющие структуры)
- 2.11. Операции присваивания
- 2.12. Операции инкремента и декремента
- 2.13. Основы повторения, управляемого счетчиком
- 2.14. Структура повторения *for* (ЦИКЛ)
- 2.15. Примеры использования структуры *for*
- 2.16. Структура множественного выбора *switch*
- 2.17. Структура повторения *do/while*
- 2.18. Операторы *break* и *continue*
- 2.19. Логические операторы
- 2.20. Ошибки случайной подмены операций проверки равенства (==) и присваивания (=)
- 2.21. Заключение по структурному программированию
- 2.22. Размышления об объектах: идентификация объектов задачи

*Резюме* • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

## 2.1. Введение

Прежде, чем писать программу для решения конкретной задачи, необходимо досконально понять эту задачу и тщательно спланировать пути ее решения. Когда пишется программа, в равной степени важно понимать типы имеющихся в распоряжении стандартных блоков и использовать обоснованные правила построения программы. В данной главе мы обсудим все эти вопросы в процессе знакомства с теорией и принципами структурного программирования. Технология, которую мы будем изучать, применима к большинству языков высокого уровня, включая C++. Когда в главе 6 мы будем рассматривать интерпретацию объектно-ориентированного программирования в C++, мы увидим, что структуры управления, изученные в главе 2, очень полезны при создании объектов и манипулировании ими.

## 2.2. Алгоритмы

Любая вычислительная задача может быть решена путем выполнения в определенной последовательности некоторой совокупности операций. *Процедура решения задачи в терминах*

1. *операций*, которые должны выполняться, и
2. *последовательности*, в которой эти операции должны выполняться, называется *алгоритмом*. Приведенный ниже пример показывает, насколько важно правильно определить последовательность выполнения операций.

Рассмотрим «алгоритм утреннего пробуждения», который выполняет молодой человек, вставая с постели и отправляясь на работу: (1) встать с постели, (2) снять пижаму, (3) принять душ, (4) одеться, (5) позавтракать, (6) поехать на работу.

Выполнение этой программы позволяет хорошо подготовиться к работе и быть готовым к принятию важных решений. Предположим, однако, что те же шаги выполняются в несколько другой последовательности: (1) встать с постели, (2) снять пижаму, (3) одеться, (4) принять душ, (5) позавтракать, (6) поехать на работу.

В этом случае наш молодой человек отправится на работу слегка мокрым. Определение последовательности, в которой должны выполняться операции в компьютерной программе, называется *программным управлением*. В этой главе мы изучим возможности программного управления в C++.

## 2.3. Псевдокод

*Псевдокод* — это искусственный и неформальный язык, который помогает программисту разрабатывать алгоритмы. Псевдокод, который мы рассматриваем, часто используется для разработки алгоритмов, которые потом должны быть преобразованы в структурированную программу на C++. Псевдокод подобен разговорному языку; он удобный и дружелюбный, но это не язык программирования.

Программы на псевдокоде не могут выполняться на компьютере. Их назначение — помочь программисту «обдумать программу» прежде, чем попытаться написать ее на таком языке программирования, как C++. В этой главе мы приведем несколько примеров того, как можно эффективно использовать псевдокод при создании структурированных программ на C++.

Тот псевдокод, который мы рассматриваем, состоит исключительно из символов, так что программист может писать на нем свои программы, используя любой текстовый редактор, и просматривать их на экране. Тщательно подготовленная программа на псевдокоде может быть легко преобразована в соответствующую программу на C++. Во многих случаях для этого достаточно просто заменить предложения псевдокода их эквивалентами в языке C++.

Псевдокод включает только исполняемые операторы — те, которые выполняются, когда программа переведена из псевдокода на C++ и запущена на счет. Объявления не являются исполняемыми операторами. Например, объявление

```
int i;
```

просто сообщает компилятору тип переменной *i* и дает указание зарезервировать для нее место в памяти. Но это объявление не вызывает при выполнении программы какого-то действия — ввода данных, их вывода или каких-нибудь вычислений. Некоторые программисты предпочитают составлять список переменных с кратким описанием их назначения и помещать его в начале программы на псевдокоде.

## 2.4. Управляющие структуры

Обычно операторы программы выполняются друг за другом в той последовательности, в которой они написаны. Это называется *последовательным выполнением*. Однако, как мы скоро увидим, различные операторы C++ позволяют программисту указать, что следующим должен выполняться не очередной оператор в тексте программы, а какой-то другой. Это называется *передачей управления*.

В 60-е годы стало ясно, что неограниченное использование передач управления является источником множества неприятностей при групповой разработке программного обеспечения. Вина была возложена на *оператор goto*, который позволяет программисту передавать управление в очень широких пределах. Идеи так называемого структурного программирования стали почти синонимами требования «не применять *goto*».

Исследование Бома и Джакопини\* показало, что программы могут быть написаны без использования оператора *goto*. В результате для программистов настала эра перехода к стилю программирования «с минимальным использованием *goto*». Такого не было до 70-х годов, пока программисты не стали серьезно применять структурное программирование. Результаты получились впечатляющие: группы разработчиков программного обеспечения сообщали, что время разработок сократилось, производительность труда выросла и проекты стали чаще укладываться в рамки бюджета. Ключом к успеху явилось то, что структурированные программы стали более прозрачными, легче под-

---

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

давались отладке и модификации и, что самое главное, в них стало меньше ошибок.

Работа Бома и Джакопини показала, что все программы могут быть написаны с использованием всего трех *управляющих структур*, названных *структурой следования, структурой выбора и структурой повторения*. Структура следования встроена в C++. Пока не указано иное, компьютер выполняет операторы C++ один за другим в той последовательности, в которой они записаны. Фрагмент блок-схемы на рис. 2.1 иллюстрирует типичную структуру следования, в которой две вычислительных операции выполняются последовательно.

Блок-схема — это графическое представление алгоритма или фрагмента алгоритма. Блок-схема рисуется с использованием специальных символов, таких, как прямоугольники, ромбы, овалы и малые окружности; эти символы соединяются стрелками, называемыми *линиями связи*.

Подобно псевдокоду блок-схемы часто используются при разработке и описании алгоритмов, хотя большинство программистов предпочитает псевдокод. Блок-схемы наглядно показывают, как действуют управляющие структуры; поэтому мы будем использовать их в дальнейшем изложении.

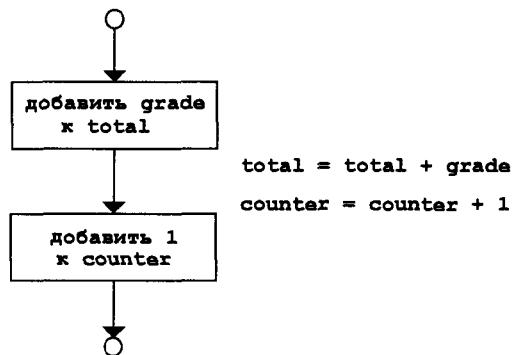


Рис. 2.1. Блок-схема структуры следования C++

Рассмотрим фрагмент блок-схемы структуры следования на рис. 2.1. Мы используем символ *прямоугольника*, называемый *узлом обработки*, чтобы обозначить какие-то действия, включая вычисления или операции ввода-вывода. Линии связи на рисунке показывают последовательность, в которой должны осуществляться операции обработки — сначала grade прибавляется к total, а затем 1 прибавляется к counter. C++ позволяет нам иметь в структуре следования столько узлов обработки, сколько мы хотим. Как мы скоро увидим, везде, где можно поместить один узел обработки (один оператор), можно поместить последовательно и несколько узлов.

Когда рисуется блок-схема, представляющая полный алгоритм, то первым ее символом является *символ овала*, содержащий слово «Начало»; символ овала, содержащий слово «Конец», является последним символом. Когда рисуется только фрагмент алгоритма, как на рис. 2.1, символы овалов заменяются *символами малых окружностей*, называемых также *символами слияния*.

Наиболее важным символом блок-схем является, пожалуй, *символ ромба*, называемый также *узлом проверки условия*, который показывает, какое из

возможных решений принято. Мы рассмотрим символ ромба в следующем разделе.

C++ обеспечивает три типа структур выбора альтернатив; мы рассмотрим все их в данной главе. Структура выбора `if` выполняет некоторое действие (обработку), если проверяемое условие истинно, или пропускает его, если условие ложно. Структура выбора `if/else` выполняет одно действие, если условие истинно, и выполняет другое действие, если оно ложно. Структура выбора `switch` выполняет одно из множества действий в зависимости от значения проверяемого выражения.

Структура `if` (ЕСЛИ) называется *структурой с единственным выбором*, поскольку она выбирает или игнорирует единственное действие. Структура `if/else` (ЕСЛИ-ИНАЧЕ) называется *структурой с двойным выбором*, так как она осуществляет выбор между двумя различными действиями. Структура `switch` (ВЫБОР) называется *структурой с множественным выбором*, так как она осуществляет выбор среди множества различных действий.

C++ обеспечивает три типа структур повторения, называемых `while` (ПОКА), `do/while` (ЦИКЛ-ПОКА) и `for` (ЦИКЛ). Все приведенные слова, такие, как `if`, `else`, `switch`, `while`, `do` и `for` являются *ключевыми словами C++*. Ключевые слова зарезервированы в языке, чтобы отражать какие-то его особенности, в частности, управляющие структуры C++. Ключевые слова нельзя использовать как идентификаторы, например, для обозначения имен переменных. Полный список ключевых слов C++ приведен на рис. 2.2.

### Типичная ошибка программирования 2.1

Использование ключевого слова в качестве идентификатора.

| Ключевые слова C++ |          |         |           |          |          |
|--------------------|----------|---------|-----------|----------|----------|
| С 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 C++         |          |         |           |          |          |
| asm                | catch    | class   | delete    | friend   | inline   |
| new                | operator | private | protected | public   | template |
| this               | throw    | try     | virtual   |          |          |

Рис. 2.2. Ключевые слова C++

И это все. C++ имеет только семь управляющих структур: следования, три типа выбора и три типа повторения. Любая программа на C++ формируется из такого количества комбинаций каждого типа управляющих структур, которое нужно для осуществления соответствующего алгоритма. Как и структура следования на рис. 2.1, каждая управляющая структура на блок-схеме содержит два символа малых окружностей: один — для точки входа

структурой и один — для точки выхода. Подобные *управляющие структуры с одним входом и одним выходом* облегчают построение программы — управляющие структуры связываются друг с другом соединением точки выхода одной из них с точкой входа другой. Это подобно тому, как ребенок складывает кубики, и мы назвали это *пакетированием управляющих структур (stacking)*. В дальнейшем мы выясним, что помимо рассмотренного есть только один путь соединения управляющих структур — их *вложение*. Таким образом, любая программа на C++, которую вы будете создавать, может быть построена всего на семи типах управляющих структур, соединенных всего двумя способами.

## 2.5. Структура выбора if (ЕСЛИ)

Структура выбора используется для выбора среди альтернативных путей обработки информации. Например, предположим, что проходной бал на экзамене — 60. Предложение на псевдокоде

*ЕСЛИ оценка студента больше или равна 60  
Напечатать "Зачет"*

определяет, истинно или ложно условие «оценка студента больше или равна 60». Если это условие истинно, то печатается «Зачет» и «выполняется» следующее по порядку предложение псевдокода (напомним, что псевдокод — это в действительности не язык программирования). Если же данное условие ложно, то предложение печати игнорируется и сразу выполняется следующее по порядку предложение псевдокода. Заметьте, что вторая строка структуры выбора напечатана с отступом. Подобные отступы не обязательны, но их настоятельно рекомендуется делать, так как они подчеркивают структуры структурированных программ. Компилятор C++ игнорирует такие символы-разделители, как пробелы, символы табуляции, перевода строки, используемые для структурированного расположения текста и его вертикальной разрядки.

### Хороший стиль программирования 2.1

Неукоснительно соблюдайте правила ступенчатой записи во всех ваших программах, это существенно улучшит их читаемость. Мы советуем делать отступы фиксированного размера примерно 0,5 см или три пробела на отступ.

Соответствующий приведенному псевдокоду оператор if может быть записан на языке C++ как

```
if (grade >= 60)
 cout << "Зачет" << endl;
```

Отметим, что данный код C++ очень близок псевдокоду. Это одно из свойств псевдокода, которое делает его столь удобным для разработки программ.

### Хороший стиль программирования 2.2

Псевдокод часто используется при «обдумывании» программ в процессе их разработки. Затем программа на псевдокоде преобразуется в программу на C++.

Блок-схема на рис. 2.3 иллюстрирует структуру с единственным выбором *if*. Эта блок-схема содержит может быть самый важный символ блок-схем — символ ромба, называемый также символом *узла проверки условия*, который показывает, какой выбор должен быть сделан. Символ узла проверки условия содержит выражение — условие, которое может быть истинным или ложным. Символ имеет две выходящие из него линии связи. Одна показывает направление, которое выбирается, если выражение в символе истинно («*true*», «да»); другая показывает направление, которое выбирается, если выражение ложно («*false*», «нет»). В первой главе мы увидели, что выбор может осуществляться на основании условия, содержащего операции отношения или проверки на равенство. В действительности, выбор может основываться на любом выражении — если результат выражения равен нулю, то это трактуется как ложь, а если результат выражения отличен от нуля, то это трактуется как истина.

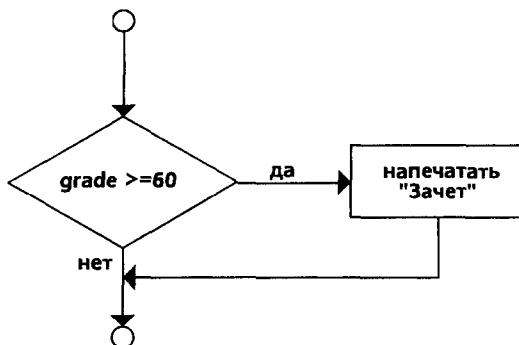


Рис. 2.3. Блок-схема структуры с единственным выбором *if*

Отметим, что структура *if*, как и другие, является структурой с одним входом и одним выходом. Скоро мы выясним, что блок-схемы остальных управляемых структур также состоят (кроме символов малых окружностей и линий связи) только из символов прямоугольников, указывающих на производимое действие, и символов ромбов, указывающих на проводимый выбор. Это модель программирования *действие-выбор*, на которую мы хотим обратить особое внимание.

Мы можем представить себе семь бункеров, каждый из которых содержит управляющие структуры только одного типа. Эти структуры пустые. Ничего не написано ни в прямоугольниках, ни в ромбах. Задача программиста — собрать программу из множества управляющих структур различных типов в соответствии с требованием алгоритма, соединить эти структуры одним из двух возможных способов (пакетированием или вложением) и затем записать действия и условия выбора, соответствующие алгоритму. Мы рассмотрим позднее различные способы записи действий и условий выбора.

## 2.6. Структура выбора *if/else* (ЕСЛИ-ИНАЧЕ)

Структура выбора *if* выполняет указанное в ней действие только, если условие истинно, и пропускает его в ином случае. Структура выбора *if/else* позволяет программисту определить различные действия, которые должны

выполняться в случаях, если условие истинно или ложно. Например, предложение псевдокода

*ЕСЛИ оценка студента больше или равна 60*

*Напечатать "Зачет"*

*ИНАЧЕ*

*Напечатать "Незачет"*

печатает *Зачет*, если оценка студента больше или равна 60, и печатает *Незачет*, если оценка меньше 60. В обоих случаях после печати «выполняется» следующее по порядку предложение псевдокода. Отметим, что тело ИНАЧЕ также записывается с отступом.

### Хороший стиль программирования 2.3

Записывайте с отступом оба предложения структуры ЕСЛИ-ИНАЧЕ (*if/else*).

Какой бы стиль отступов вы ни приняли, необходимо строго придерживаться его во всех программах. Трудно читать программы, в которых в отступах не поддерживается постоянное количество пробелов.

### Хороший стиль программирования 2.4

Если есть несколько уровней отступов, каждый уровень должен иметь постоянное число пробелов.

Рассмотренный псевдокод структуры *if/else* может быть записан на C++ следующим образом:

```
if (grade >= 60)
 count << "Зачет" << endl;
else
 count << "Незачет" << endl;
```

Блок-схема на рис. 2.4 хорошо иллюстрирует управляющую логику структуры *if/else*. Еще раз отметим, что единственными символами на этой схеме кроме малых окружностей и линий связи являются прямоугольники (для обозначения действий) и ромб (для обозначения выбора). Мы продолжаем подчеркивать модель вычислений действие-выбор. Представим себе снова глубокий бункер, содержащий столько структур с двойным выбором, сколько может потребоваться для построения любой программы на C++. Задача программиста сводится к соединению этих структуры выбора (пакетированием или вложением) с другими управляющими структурами, требуемыми алгоритмом, и к заполнению пустых прямоугольников и ромбов необходимыми действиями и условиями выбора.

C++ имеет еще *условную операцию* (?), которая близка к структуре *if/else*. Условная операция — единственная *трехчленная (тернарная) операция* в C++, имеющая три операнда. Эти операнды вместе с условной операцией формируют *условное выражение*. Первый операнд является условием, второй операнд содержит значение условного выражения в случае, если условие истинно, а третий операнд равен значению условного выражения, если условие ложно. Например, оператор вывода

```
cout << (grade >= 60 ? "Зачет" : "Незачет") << endl;
```

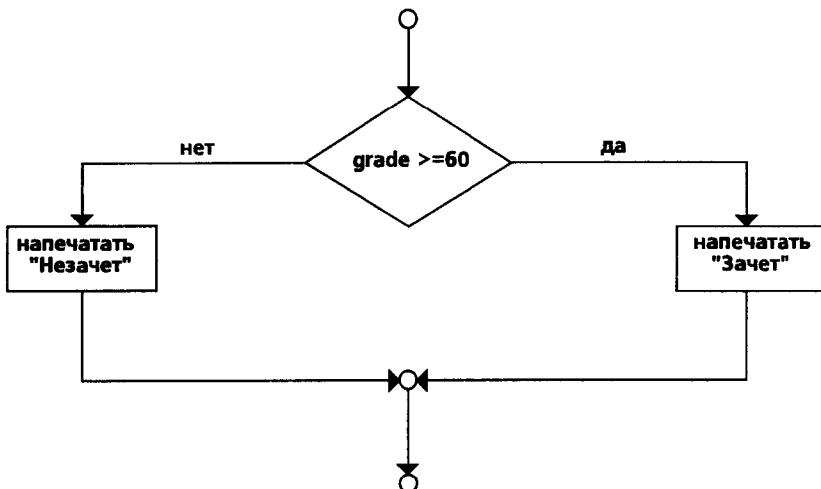


Рис. 2.4. Блок-схема структуры с двойным выбором if/else

содержит условное выражение, которое равно строке "Зачет", если условие `grade >= 60` истинно, и равно строке "Незачет", если оно ложно. Таким образом, этот оператор с условной операцией выполняет фактически те же функции, что и приведенный ранее оператор `if/else`. Как мы увидим далее, условная операция имеет низкое старшинство и поэтому в приведенном выражении потребовались скобки. Без них компилятор будет пытаться произвести вывод величины оценки `grade` операцией `cout << grade`, что, естественно, не входит в намерения программиста. Затем последует ряд синтаксических ошибок, включая попытку компилятора найти значение для "Незачет" `<< endl`.

Значения условного выражения могут быть также какими-то исполняемыми действиями. Например, условное выражение

```
grade >= 60 ? cout << "Зачет\n" : cout << "Незачет\n";
```

читается так: «Если оценка `grade` больше или равна 60, то `cout << "Зачет\n"`, иначе `cout << "Незачет\n"`». Это также сравнимо с рассмотренной структурой `if/else`. Мы увидим в дальнейшем, что иногда условная операция может использоваться в таких ситуациях, когда применение оператора `if/else` невозможно.

Для множественного выбора можно использовать вложенные структуры `if/else`, помещая одну структуру `if/else` внутрь другой. Например, следующее предложение псевдокода будет печатать А при экзаменационной оценке больше или равной 90, В — при оценке, лежащей в диапазоне от 80 до 89, С — при оценке в диапазоне 70–79, D — при оценке в пределах 60–69 и F — при других оценках.

**ЕСЛИ** оценка студента больше или равна 90

Напечатать "А"

**ИНАЧЕ**

ЕСЛИ оценка студента больше или равна 80

Напечатать "В"

ИНАЧЕ

```

ЕСЛИ оценка студента больше или равна 70
 Напечатать "C"
ИНАЧЕ
 ЕСЛИ оценка студента больше или равна 60
 Напечатать "D"
 ИНАЧЕ
 Напечатать "F"

```

Этот псевдокод может быть записан на языке C++ в виде:

```

if (grade >= 90)
 cout << "A" << endl;
else
 if (grade >= 80)
 cout << "B" << endl;
 else
 if (grade >= 70)
 cout << "C" << endl;
 else
 if (grade >= 60)
 cout << "D" << endl;
 else
 cout << "F" << endl;

```

Если оценка `grade` больше или равна 90, то четыре условия истинны, но будет выполнен только оператор `cout`, расположенный после проверки первого условия. После того, как этот оператор `cout` выполнен, часть `else` внешнего оператора `if/else` пропускается. Большинство программистов на C++ предпочтут записать предыдущую структуру `if` в виде:

```

if (grade >= 90)
 cout << "A" << endl;
else if (grade >= 80)
 cout << "B" << endl;
else if (grade >= 70)
 cout << "C" << endl;
else if (grade >= 60)
 cout << "D" << endl;
else
 cout << "F" << endl;

```

Обе формы записи эквивалентны. Последняя форма популярна, поскольку она позволяет избежать сдвига кода далеко вправо. Подобный сдвиг часто оставляет мало места в строке, заставляя дробить и переносить строки, что ухудшает читаемость программы.

Структура выбора `if` обычно предполагает наличие в своем теле только одного оператора. Чтобы включить несколько операторов в тело структуры, заключите их в фигурные скобки: «{» и «}». Множество операторов, заключенных в фигурные скобки, называется *составным оператором*.

### Замечание по технике программирования 2.1

Составной оператор может быть помещен в любом месте программы, в котором может размещаться единичный оператор.

Следующий пример включает составной оператор в часть `else` структуры `if/else`.

```
if (grade >= 60)
 count << "Зачет." << endl;
else {
 count << "Незачет." << endl;
 count << "Вы должны изучить этот курс снова." << endl;
}
```

В этом случае при оценке `grade` меньше 60 программа выполнит оба оператора в теле `else` и напечатает:

**Незачет.**  
**Вы должны изучить этот курс снова.**

### Типичная ошибка программирования 2.2

Пропуск одной или обеих фигурных скобок, ограничивающих составной оператор.

**Синтаксическая ошибка** обнаруживается компилятором. **Логическая ошибка** проявляется только во время выполнения программы. **Неисправимая логическая ошибка** вызывает сбой в работе программы и ее преждевременное завершение. При *исправимой логической ошибке* программа продолжает работать, но выдает неправильные результаты.

### Замечание по технике программирования 2.2

Справедливо, что составной оператор может быть помещен в любом месте программы, в котором может размещаться единичный оператор, но также справедливо и то, что можно вообще обойтись без оператора, т.е. поместить пустой оператор. Для этого надо поместить символ точки с запятой (;) в том месте, где нормально должен находиться оператор.

### Типичная ошибка программирования 2.3

Запись точки с запятой после условия в структуре `if` приводит к логической ошибке в структуре с единственным выбором и к синтаксической ошибке в структуре с двойным выбором (если часть `if` в действительности содержит оператор).

### Хороший стиль программирования 2.5

Некоторые программисты предпочитают сначала записать открывающую и закрывающую скобки составного оператора, а уже потом писать внутри их требуемые операторы. Это позволяет избежать пропуска одной или обеих скобок.

В данном разделе мы ввели понятие составного оператора. Составной оператор может содержать объявления (например, это делается в теле `main`). В этом случае он называется **блок**. Объявления обычно размещаются в блоке на первом месте, до каких-либо исполняемых операторов; но они могут и перемежаться с исполняемыми операторами. Мы обсудим использование блоков в главе 3. До этого времени читателю лучше избегать использования блоков, кроме, конечно, тела `main`.

## 2.7. Структура повторения while (ПОКА)

Структура повторения позволяет программисту определить действие, которое должно повторяться, пока некоторое условие остается истинным. Предложение псевдокода

*ПОКА имеются элементы в моем списке покупок*

*Сделать следующую покупку и вычеркнуть ее из списка*

описывает повторные действия при посещении магазина. Условие «имеются элементы в моем списке покупок» может быть истинным или ложным. Если оно истинно, то осуществляется действие «Сделать следующую покупку и вычеркнуть ее из списка». Это действие будет повторяться до тех пор, пока условие остается истинным. Оператор (или операторы), записанные в теле структуры повторения **while** (ПОКА) составляют тело **while**, которое может быть отдельным или составным оператором. Очевидно, что когда-нибудь условие станет ложным (когда будет осуществлена и вычеркнута из списка последняя покупка). Начиная с этого момента повторение прерывается и выполняется первое предложение псевдокода, следующее за структурой повторения.

### Типичная ошибка программирования 2.4

В теле структуры **while** не предусматривается действие, которое приведет к тому, что со временем условие **while** станет ложным. Выполнение подобной структуры повторения никогда не прервется — такая ошибка называется «зацикливание».

### Типичная ошибка программирования 2.5

Запись ключевого слова **while** как **While** с символом **W** в верхнем регистре (помните, что язык C++ чувствителен к регистру). Все зарезервированные ключевые слова C++, такие, как **while**, **if** и **else**, содержат только символы нижнего регистра.

Как пример использования **while** рассмотрим фрагмент программы, определяющий первое значение степени 2, превышающее 1000. Предположим, что имеется целая переменная **product**, с начальным значением 2. Когда приведенная ниже структура повторения **while** закончит свою работу, **product** будет содержать искомую величину:

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

Блок-схема на рис. 2.5 иллюстрирует управляющую логику структуры повторения **while**. Еще раз отметим, что единственными символами на этой схеме кроме малых окружностей и линий связи являются прямоугольник и ромб. Представьте себе снова глубокий бункер, наполненный структурами **while**, которые могут соединяться пакетом или вкладываться в другие управляющие структуры, чтобы сформировать структурированное представление управления потоками в алгоритме. Затем пустые прямоугольники и ромбы заполняются соответствующими действиями и условиями выбора. Блок-схема ясно показывает повторение. Линия связи, выходящая из прямоугольника, возвращает назад к условию выбора, которое проверяется в каждом цикле, пока оно не окажется ложным. В этот момент структура

**while** завершает работу и управление передается следующему оператору программы.

Когда осуществляется вход в структуру **while**, значение переменной **product** равно 2. Эта переменная умножается в цикле на 2, принимая последовательно значения 4, 8, 16, 32, 64, 128, 256, 512, 1024. Когда **product** достигает 1024, условие структуры **while** — **product <= 1000** становится ложным. Это прекращает повторение и окончательное значение **product** будет равно 1024. Программа продолжает работу с первого оператора, следующего за **while**.

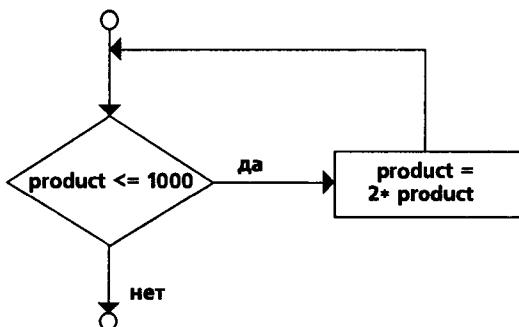


Рис. 2.5. Блок-схема структуры повторения **while**

## 2.8. Разработка алгоритмов: учебный пример 1 (повторение, управляемое счетчиком)

Чтобы проиллюстрировать процесс разработки алгоритма, рассмотрим несколько вариантов решения классической задачи усреднения. Пусть задача сформулирована следующим образом:

*Проведен опрос класса из десяти студентов. Вам известны оценки по этому опросу (целые числа в диапазоне 0–100). Надо определить среднюю оценку класса.*

Средняя по классу оценка равна сумме оценок, деленной на число студентов. Алгоритм решения этой задачи на компьютере должен включать ввод каждой оценки, расчет средней величины и печать результата.

Давайте используем псевдокод, чтобы сформировать список действий, которые должны быть выполнены, и определить последовательность их выполнения. Мы используем **повторение, контролируемое счетчиком**, для поочередного ввода оценок. Эта технология использует переменную, называемую **counter**, для задания числа повторений выполнения группы операторов. В нашем примере повторение прерывается, когда значение переменной **counter** превысит 10. В данном разделе мы представим псевдокод алгоритма (рис. 2.6) и соответствующую ему программу (рис. 2.7). В следующем разделе мы покажем, как разрабатываются псевдокоды алгоритмов. Повторение, контролируемое счетчиком, часто называют **повторением с заданным числом циклов**, поскольку число повторений известно до того, как цикл начнет исполняться.

*Установить значение суммы total в нуль  
Установить счетчик оценок counter в единицу*

*ПОКА счетчик оценок меньше или равен 10*

*Ввести следующую оценку*

*Добавить эту оценку к сумме*

*Добавить единицу к счетчику*

*Определить среднюю по классу оценку как сумму, деленную на 10  
Напечатать среднюю по классу оценку*

Рис. 2.6. Псевдокод алгоритма, использующего повторение, управляемое счетчиком, для расчета среднего значения

```
// Программа для расчета среднего значения
// с повторением, управляемым счетчиком
#include <iostream.h>

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

 // этап задания начальных значений
 total = 0;
 counter = 1;

 // этап обработки данных
 while (counter <= 10) {
 cout << "Введите оценку: ";
 cin >> grade;
 total = total + grade;
 counter = counter + 1;
 }

 // завершающий этап
 average = total / 10; // целочисленное деление
 cout << "Средняя оценка равна " << average << endl;

 return 0; // указание, что программа успешно завершена
}
```

---

```
Введите оценку: 98
Введите оценку: 76
Введите оценку: 71
Введите оценку: 87
Введите оценку: 83
Введите оценку: 90
Введите оценку: 57
Введите оценку: 79
Введите оценку: 82
Введите оценку: 94
Средняя оценка равна 81
```

Рис. 2.7. Программа C++ для расчета среднего значения с помощью повторения, управляемого счетчиком, и пример ее выполнения

Отметим применение в этом алгоритме переменных `total` и `counter`. Переменная `total` используется для наращивания суммы ряда величин. Переменная `counter` используется для счета — в данном случае для счета числа введенных оценок. Переменным, применяемым для хранения суммы, обычно должны быть заданы нулевые начальные значения перед их использованием в программе; иначе сумма может включать в себя то первоначальное значение, которое содержалось в области памяти, которая была отведена под эту переменную. Переменным счетчиков обычно задают начальные значения нуль или единица в зависимости от их использования (в дальнейшем мы приведем примеры обоих этих способов). Переменная, которой не задано начальное значение, содержит «мусор» — последнее значение, сохраненное в той области памяти, которая зарезервирована для данной переменной.

### Типичная ошибка программирования 2.6

Если для счетчика или переменной суммы не задады начальные значения, то результат работы вашей программы будет, вероятно, неправильным. Это пример логической ошибки.

### Хороший стиль программирования 2.6

Задавайте начальные значения всем счетчикам и переменным сумм.

Отметим, что расчет среднего значения в данной программе дает целый результат. В действительности сумма оценок в приведенном примере 817, что дает после деления на 10 значение 81,7, т.е. число с десятичной запятой. Мы увидим, как обращаться с такими числами (называемыми числами с плавающей запятой), в следующем разделе.

## **2.9. Нисходящая разработка алгоритмов с пошаговой детализацией: учебный пример 2 (повторение, управляемое меткой)**

Давайте обобщим задачу расчета среднего значения оценки. Рассмотрим такую задачу:

*Разработать программу расчета среднего значения оценки, которая работала бы с произвольным количеством оценок при каждом запуске программы.*

В предыдущем примере число оценок (10) было заранее известно. В данном примере не дается указаний о числе вводимых оценок. Программа должна работать при произвольном их числе. Как программа может определить, когда закончится ввод оценок? Откуда ей знать, когда надо рассчитывать среднее значение и печатать результат?

Один из путей решения этой проблемы — использование специальной величины, называемой *меткой* (иногда она называется также *знаком*, *фиктивной величиной*, *флагом*), которая указывает на конец ввода данных. Пользователь вводит оценки до тех пор, пока они все не исчерпаны. Затем он вводит метку, чтобы показать, что последняя оценка уже введена. Повторе-

ние, управляемое меткой, часто называют *неопределенным повторением*, поскольку число повторений не известно до начала исполнения цикла.

Очевидно, что метка должна быть выбрана так, чтобы ее значение не могла быть спутано с любым возможным значением входной величины. Поскольку оценки за опрос всегда не отрицательные величины, в качестве метки в данной задаче может быть выбрана величина **-1**. Тогда при запуске программы расчета средней оценки входные данные могут быть такими: **95, 96, 75, 74, 89 и -1**. После этого программа должна рассчитать и напечатать среднее значение оценок **95, 96, 75, 74 и 89 (-1 — метка и она не должна участвовать в расчете средней оценки)**.

### **Типичная ошибка программирования 2.7**

Выбор такого значения метки, которое могут принимать и входные данные.

Мы подойдем к созданию программы расчета среднего значения, используя технологию нисходящей (сверху-вниз) разработки с пошаговой детализацией. Эта технология является неотъемлемой частью создания хорошо структурированных программ. Начнем с записи псевдокода *вершины*:

*Определить среднюю оценку за опрос по классу*

Вершина — это единственное предложение (оператор), выраждающее общее назначение программы. Таким образом, вершина отображает всю программу в целом. К сожалению, вершина (как в данном случае) редко отображает достаточное количество деталей, на основании которых можно написать программу на C++. Поэтому надо начинать процесс детализации. Разделим вершину на ряд более мелких задач и запишем их в том порядке, в котором они должны выполняться. В результате получим следующую *первую детализацию*:

*Задать начальные значения переменных*

*Ввести оценки, просуммировать их и сосчитать их количество*

*Рассчитать и напечатать среднее значение*

Здесь использована только структура следования — записанные шаги должны выполняться последовательно один за другим.

### **Замечание по технике программирования 2.3**

Каждая детализация, так же, как и сама вершина, является полным описанием алгоритма; меняется только уровень детализации.

Чтобы перейти к следующему — *второму уровню детализации* зафиксируем спецификацию переменных. Нам нужны переменные для суммы оценок (**total**), для подсчета количества оценок (**counter**), переменная для восприятия каждой оценки в момент ее ввода (**grade**) и переменная для хранения подсчитанной средней оценки (**average**). Предложение псевдокода

*Задать начальные значения переменных*

может быть детализировано следующим образом:

*Задать начальное значение 0 переменной total*

*Задать начальное значение 1 переменной counter*

Отметим, что задавать начальные значения надо только переменные `total` и `counter`; переменные `average` и `grade` не нуждаются в задании начальных значений, поскольку они будут записываться в момент вычисления или ввода.

Предложение псевдокода

*Ввести оценки, просуммировать их и сосчитать их количество*

требует структуры повторения (т.е. цикла) для последовательного ввода каждой оценки. Поскольку мы не знаем заранее, сколько оценок надо обработать, мы будем использовать повторение, управляемое меткой. При каждом повторении цикла пользователь будет вводить одно допустимое значение оценки. После того, как введена последняя оценка, он должен будет ввести метку. Программа будет проверять, не является ли меткой очередное введенное значение, и прервет цикл после ввода метки. Вторая детализация этого предложения имеет вид:

*Ввести первую оценку*

*ПОКА пользователь не ввел метку*

*Добавить введенную оценку к текущему значению total*

*Добавить единицу с счетчику оценок counter*

*Ввести следующую оценку (возможно, что это будет метка)*

Отметим, что в псевдокоде мы не используем фигурные скобки для выделения предложений, составляющих тело структуры ПОКА (while). Мы просто записываем эти предложения с отступом относительно ПОКА, чтобы показать, что все они относятся именно к этому ПОКА. Вспомним, что псевдокод — всего лишь неформализованный способ помочь в разработке программы.

Предложение псевдокода

*Рассчитать и напечатать среднее значение*

может быть детализировано следующим образом:

*ЕСЛИ счетчик counter не равен нулю*

*Рассчитать среднюю оценку average как total, деленная на counter*

*Напечатать среднюю оценку*

*ИНАЧЕ*

*Напечатать "Оценки не введены"*

Обратите внимание, что мы проявили осторожность, введя проверку возможности деления на нуль — т.е. возможности возникновения *неисправимой логической ошибки*. Если не предусмотреть возможность появления подобной ошибки, она приведет к сбою программы (часто называемому «бомбой» или «аварийным отказом»). Полностью вторая детализация псевдокода для расчета средней оценки приведена на рис.2.8.

### Типичная ошибка программирования 2.8

Попытка деления на нуль вызывает сбой программы.

### Хороший стиль программирования 2.7

Когда осуществляется деление на выражение, значение которого может равняться нулю, надо предварительно проверить эту возможность и обработать ее должным образом (например, напечатать сообщение об ошибке), не допуская возникновения неисправимой ошибки.

*Задать начальное значение 0 переменной total*

*Задать начальное значение 1 переменной counter*

*Ввести первую оценку*

*ПОКА пользователь не ввел метку*

*Добавить введенную оценку к текущему значению total*

*Добавить единицу с счетчику оценок counter*

*Ввести следующую оценку (возможно, что это будет метка)*

*ЕСЛИ счетчик counter не равен нулю*

*Рассчитать среднюю оценку average как total, деленная на counter*

*Напечатать среднюю оценку*

*ИНАЧЕ*

*Напечатать "Оценки не введены"*

**Рис. 2.8.** Псевдокод алгоритма, использующего повторение, управляемое меткой, для решения задачи расчета среднего значения

На рис. 2.6 и 2.8 мы включили в псевдокод пустые строки, чтобы облегчить его чтение. Пустые строки разбивают эти программы, выделяя в них отдельные этапы.

#### Замечание по технике программирования 2.4

Многие программы могут быть логически разделены на три этапа: этап задания начальных значений, в котором задаются начальные значения переменных программы; этап обработки данных, в котором вводятся данные и устанавливаются значения соответствующих переменных программы; заключительный этап, в котором вычисляются и печатаются окончательные результаты.

Приведенный на рис. 2.8 псевдокод алгоритма решает достаточно общую задачу расчета среднего значения. Для разработки этого алгоритма потребовалось всего два уровня детализации. Часто требуется большее количество уровней.

#### Замечание по технике программирования 2.5

Программист завершает процесс нисходящей разработки с пошаговой детализацией, когда алгоритм на псевдокоде настолько детализирован, чтобы его псевдокод можно было бы преобразовать в программу на C++. Реализованная программа на C++ окажется в этом случае простой и наглядной.

Программа на C++ и пример ее выполнения приведены на рис. 2.9. Несмотря на то, что вводятся только целые значения оценок, расчет средней величины оценки вероятно даст число с десятичной запятой (т.е. действительное число). Тип `int` не может описывать действительные числа. Поэтому в программу вводится переменная типа `float`, чтобы оперировать с числами с десятичной запятой (называемыми также числами с плавающей запятой), и вводится специальная операция, называемая *операцией приведения к типу* для обработки вычисления среднего значения. Эти особенности будут объяснены более детально после рассмотрения программы.

```

// Программа для расчета среднего значения
// с повторением, управляемым меткой
#include <iostream.h>
#include <iomanip.h>

main()
{
 float average; // число с десятичной запятой
 int counter, grade, total;

 // этап задания начальных значений
 total = 0;
 counter = 0;

 // этап обработки данных
 cout << "Введите оценку или -1 для завершения: ";
 cin >> grade;

 while (grade != -1) {
 total = total + grade;
 counter = counter + 1;
 cout << "Введите оценку или -1 для завершения: ";
 cin >> grade;
 }

 // завершающий этап
 if (counter != 0) {
 average = (float) total / counter;
 cout << "Среднее значение равно " << setprecision(2)
 << setiosflags(ios::fixed | ios::showpoint)
 << average << endl;
 }
 else
 cout << "Оценки не введены" << endl;
}

return 0; // указание, что программа успешно завершена
}

```

---

```

Введите оценку или -1 для завершения: 75
Введите оценку или -1 для завершения: 94
Введите оценку или -1 для завершения: 97
Введите оценку или -1 для завершения: 88
Введите оценку или -1 для завершения: 70
Введите оценку или -1 для завершения: 64
Введите оценку или -1 для завершения: 83
Введите оценку или -1 для завершения: 89
Введите оценку или -1 для завершения: -1
Средняя оценка равна 82.50

```

**Рис. 2.9.** Программа C++ для расчета среднего значения с помощью повторения, управляемого меткой

Отметим, составной оператор в цикле `while` на рис. 2.9. Без фигурных скобок последние три оператора тела цикла выпали бы из этого цикла, приведя к такой неправильной интерпретации компилятором этого кода:

```
while (grade != -1)
 total = total + grade;
counter = counter + 1;
cout << "Введите оценку или -1 для завершения: ";
cin >> grade;
```

Это привело бы к бесконечному циклу (зацикливанию), если бы только пользователь не ввел `-1` в качестве первой оценки.

### Хороший стиль программирования 2.8

В цикле, управляемом меткой, приглашение к вводу данных должно явно напоминать пользователю, какое значение используется как метка.

Средние значения не всегда выражаются целыми числами. Часто среднее значение имеет величину типа `7,2` или `-93,5`, содержащую дробную часть. Подобные числа описываются как числа с плавающей запятой и представляются типом данных `float`. Переменная `average` объявлена как переменная типа `float`, чтобы учесть дробную часть результатов вычислений. Однако, результат вычисления `total / counter` является целым числом, поскольку и `total` и `counter` — переменные целого типа. Деление двух целых чисел осуществляется как *целочисленное деление*, при котором любая дробная часть результата теряется (т.е. отсекается). Поскольку сначала осуществляется деление, дробная часть потерянная прежде, чем результат будет присвоен переменной `average`. Чтобы осуществлять над целыми числами вычисления с плавающей запятой, надо создавать для вычислений временные величины с плавающей запятой. В C++ для решения этой задачи вводится *унарная операция приведения к типу*. Оператор

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

включает операцию приведения к типу (`float`), которая создает временную копию с плавающей запятой своего операнда `total`. Подобное использование операции приведения к типу называется *явным преобразованием*. Величина, сохраняемая в `total`, остается целой. А вычисления теперь сводятся к делению величины с плавающей запятой (временной копии `total` типа `float`) на целую величину `counter`.

Компилятор C++ знает только, как вычислять выражения с операндами, имеющими идентичные типы. Чтобы обеспечить одинаковый тип operandov, компилятор осуществляет операцию *преобразования по умолчанию* (называемую также *неявным преобразованием типов*) над выделенными operandами. Например, в выражении, содержащем данные типов `int` и `float`, operand типа `int` преобразуется в тип `float`. В нашем примере после того, как `counter` будет преобразован во `float`, осуществляются вычисления и результат деления с плавающей запятой присваивается переменной `average`. Позднее в данном разделе мы обсудим все стандартные типы данных и последовательность их преобразования.

Операции приведения к типу применимы к любым типам данных. Операция оформляется как имя типа данных, помещенное в круглые скобки. Приведение к типу является *унарной операцией*, т.е. операцией, имеющей всего один operand. В главе 1 мы изучили бинарные арифметические операции. C++ поддерживает также унарные операции унарный плюс (+) и унарный минус (-), благодаря которым программист может писать выражения типа `-7` или `+5`. Операции приведения к типу имеют ассоциативность справа

налево и тот же приоритет, что и у унарных операций унарный + и унарный -. Это тот же приоритет, что у мультипликативных операций \*, / и %, и он на один уровень ниже, чем у скобок. Мы укажем операцию приведения к типу в нотации (тип) в таблице приоритетов.

Возможности форматирования, использованные в программе рис. 2.9, подробно будут изучены в главе 11, а пока ограничимся их кратким обсуждением. Обращение `setprecision(2)` в операторе вывода

```
cout << "Среднее значение равно " << setprecision(2)
 << setiosflags(ios::fixed | ios::showpoint)
 << average << endl;
```

указывает, что переменная `average` типа `float` должна быть напечатана с точностью до двух разрядов после десятичной точки (т.е. 92.37). Это обращение является *параметризованным манипулятором потока*. Программы, использующие подобные обращения, должны содержать директиву

```
#include <iomanip.h>
```

Отметим, что `endl` является *непараметризованным манипулятором потока* и не требует заголовочного файла `iomanip.h`. Если точность не задана, то число с плавающей запятой выводится с точностью в шесть разрядов (т.е. с точностью по умолчанию), хотя мы сразу же увидим исключение из этого правила.

Манипулятор потока `setiosflags(ios::fixed | ios::showpoint)` в приведенном выше операторе задает две опции выходного формата, а именно — `ios::fixed` и `ios::showpoint`. Символ вертикальной черты (|) разделяет множество опций в обращении `setiosflags` (мы более подробно объясним нотацию | в главе 6). Опция `ios::fixed` приводит к выводу числа с плавающей запятой в так называемом формате с фиксированной точкой (в отличие от «научной» нотации, которая будет обсуждаться в главе 11). Опция `ios::showpoint` приводит к выводу десятичной точки и нулевых младших разрядов даже если величина в действительности является целым числом, например, 88.00. Без опции `ios::showpoint` подобная величина была бы напечатана как 88, т.е. без нулевых младших разрядов и без десятичной точки. Когда приведенный выше формат используется в программе, печатаемая величина округляется до указанного числа десятичных разрядов, хотя ее значение в памяти остается неизменным. Например, числа 87.945 и 67.543 будут напечатаны соответственно в виде 87.95 и 67.54.

### Типичная ошибка программирования 2.9

Использование чисел с плавающей запятой в предположении, что они совершенно точные, может приводить к некорректным результатам. Числа с плавающей запятой на большинстве компьютеров являются приближенными.

### Хороший стиль программирования 2.9

Не следует сравнивать числа с плавающей запятой на их равенство или неравенство друг другу. Лучше проверять, не меньше ли их разность некоторой заданной малой величины.

Несмотря на то, что числа с плавающей запятой не всегда «стопроцентно точные», они широко используются на практике. Например, когда мы говорим, что «нормальная» температура тела 98.6, нас не интересует значение температуры с большим числом разрядов. Когда мы смотрим на термометр и видим, что он показывает температуру 98.6, в действительности она может быть равна 98.5999473210643. Однако, для большинства приложений достаточно считать, что эта температура равна 98.6.

Числа с плавающей запятой могут получаться также в результате деления. Когда мы делим 10 на 3, результат равен 3.333333... с бесконечно повторяемой периодической частью 3. Компьютер располагает только фиксированным объемом памяти для хранения подобных величин, так что, естественно, сохраняемое значение будет только приближенным.

## 2.10. Нисходящая разработка алгоритмов с пошаговой детализацией: учебный пример 3 (вложенные управляемые структуры)

Давайте теперь поработаем над другой задачей. Мы снова будем разрабатывать алгоритм, используя псевдокод, нисходящее проектирование и пошаговую детализацию для создания программы на C++. Мы уже видели, что управляемые структуры могут собираться в пакеты, одна за другой (последовательно), как ребенок собирает кубики. А в данном учебном примере мы увидим другой путь соединения управляемых структур в C++ — *вложенные структуры*.

Рассмотрим следующую постановку задачи:

*Колледж предлагает курс, который готовит студентов для государственного экзамена на получения лицензии брокера. В прошлом году несколько студентов, прослушавших этот курс, сдавали такой экзамен. Естественно, колледж хочет знать, насколько хорошо их студенты его сдали. Вас попросили написать программу обработки результатов экзамена. Вам дали список 10-и студентов. После каждой фамилии записано 1, если студент успешно сдал экзамен, и 2, если он его не смог сдать.*

Ваша программа должна проанализировать результаты экзамена следующим образом:

1. Ввести каждый результат тестирования (т.е. 1 или 2). Перед вводом каждого результата программа должна выводить на экран приглашение «Введите результат».
2. Подсчитать число результатов каждого типа.
3. Вывести на экран суммарный результат, указав число студентов, выдержавших и не выдержавших экзамен.
4. Если более 8 студентов экзамен выдержали, напечатать сообщение «Попытесь плату за обучение».

Прочтя внимательно формулировку задания на решение данной задачи, вы можете отметить следующее:

1. Программа должна обработать 10 результатов тестирования. Можно будет использовать повторение, управляемое счетчиком.

2. Каждый результат тестирования — число, равное или 1, или 2. При каждом чтении очередного результата программа должна определять, равно ли введенное число 1, или 2. Мы будем в нашем алгоритме проверять, не равно ли число 1. Если число не 1, то мы будем предполагать, что оно равно 2. (Упражнение в конце главы рассмотрит последствия этого предположения.)
3. Надо использовать два счетчика — один для подсчета числа студентов, сдавших экзамен, и второй для подсчета числа студентов, не выдержавших испытание.
4. После того, как программа обработает все результаты, она должна решить, больше ли 8-и студентов успешно сдали экзамен.

Давайте проведем нисходящую разработку с пошаговой детализацией. Мы начнем с псевдокода, описывающего самый верхний уровень:

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

Снова подчеркнем, что самый верхний уровень описывает программу в целом, но обычно необходимо пройти через ряд детализаций, прежде чем псевдокод можно будет естественным образом преобразовать в программу на C++. Наша первая детализация:

*Задать начальные значения переменных*

*Ввести десять оценок и посчитать число сдавших и не сдавших экзамен*

*Напечатать суммарные результаты экзамена и решить, надо ли повышать плату за обучение*

Мы полностью описали всю программу, но, конечно, необходимы дальнейшие детализации. Определим необходимые нам переменные. Нам нужны счетчики для записи числа сдавших (*passes*) и не сдавших (*failures*) экзамен, счетчик для управления циклом (*student*) и переменная для хранения введенной оценки (*result*). Поэтому предложение псевдокода

*Задать начальные значения переменных*

можно детализировать следующим образом:

*Задать начальное значение 0 переменной passes*

*Задать начальное значение 0 переменной failures*

*Задать начальное значение 0 переменной student*

Отметим, что инициируются только счетчики. Предложение псевдокода

*Ввести десять оценок и посчитать число сдавших и не сдавших экзамен*

требует цикла для ввода результата каждого экзамена. В данном случае заранее известно, что число вводимых результатов ровно десять, так что можно использовать повторение, управляемое счетчиком. Внутри цикла потребуется структура (*вложенная*) двойного выбора, которая будет определять для каждой введенной оценки ее тип и увеличивать на единицу соответствующий счетчик. Таким образом, детализация приведенного предложения псевдокода имеет вид:

*ПОКА счетчик student меньше или равен десяти*

*Ввести очередную оценку экзамена*

```

ЕСЛИ студент сдал экзамен
 Добавить единицу к passes
ИНАЧЕ
 Добавить единицу к failures

```

*Добавить единицу к счетчику student*

Отметим использование пустых строк, отделяющих структуру ЕСЛИ — ИНАЧЕ и улучшающих читаемость программы. Предложение псевдокода

*Напечатать суммарные результаты экзамена и решить, надо ли повышать плату за обучение*

может быть детализировано следующим образом:

```

Напечатать число passes
Напечатать число failures
ЕСЛИ восемь или более студентов сдали экзамен
 Напечатать "Повысить плату за обучение"

```

Полностью вторая детализация приведена на рис. 2.10. Отметим пустые строки, которые использованы, чтобы отделить структуру ПОКА для лучшей читаемости программы.

```

Задать начальное значение 0 переменной passes
Задать начальное значение 0 переменной failures
Задать начальное значение 0 переменной student

```

*ПОКА счетчик student меньше или равен десяти  
Ввести очередную оценку экзамена*

```

ЕСЛИ студент сдал экзамен
 Добавить единицу к passes
ИНАЧЕ
 Добавить единицу к failures

```

*Добавить единицу к счетчику student*

```

Напечатать число passes
Напечатать число failures
ЕСЛИ восемь или более студентов сдали экзамен
 Напечатать "Повысить плату за обучение"

```

Рис. 2.10. Псевдокод обработки результатов экзаменов

Этот псевдокод теперь достаточно детализирован, чтобы его можно было преобразовать в программу на С++. Эта программа и два примера ее выполнения приведены на рис. 2.11. Отметим, что мы воспользовались полезным свойством С++, позволяющим объединять задание начальных значений переменных с их описанием. Однако, циклические программы могут требовать задания начальных значений при каждом повторении; в этом случае задание начального значения осуществляется обычным оператором присваивания.

## Хороший стиль программирования 2.10

Задание начальных значений переменных одновременно с их объявлением помогает программисту избежать проблем, связанных с неопределенными значениями данных.

## Замечание по технике программирования 2.6

Опыт показывает, что наиболее трудной частью решения задач на компьютерах является разработка алгоритма решения. После того, как корректный алгоритм получен, процесс создания на его основе работающей программы на C++ продвигается успешно.

## Замечание по технике программирования 2.7

Многие опытные программисты пишут программы, не используя такой инструмент разработки, как псевдокод. Эти программисты полагают, что их конечная цель – решение задачи на компьютере и что написание псевдокода только задержит достижение конечного результата. Это может быть иногда оправдано для простых и хорошо знакомых задач, но может приводить к серьезным ошибкам в больших и сложных проектах.

```
// Анализ результатов экзамена
#include <iostream.h>

main()
{
 // Задание начальных значений переменных,
 // совмещенное с их объявлением
 int passes = 0, failures = 0, student = 1, result;

 // обработка 10 студентов; цикл, управляемый счетчиком
 while (student <= 10) {
 cout << "Введите результат (1=сдал, 2=не сдал): ";
 cin >> result;

 if (result == 1) // оператор if/else, вложенный
 // в оператор while
 passes = passes + 1;
 else
 failures = failures + 1;

 student = student + 1;
 }

 cout << "Сдали " << passes << endl;
 cout << "Не сдали " << failures << endl;

 if (passes > 8)
 cout << "Повысить оплату за обучение" << endl;

 return 0; // успешное окончание
}
```

**Рис. 2.11.** Программа C++ для анализа результатов экзамена и примеры ее выполнения (часть 1 из 2)

```

Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 2
Ведите результат (1=сдал, 2=не сдал) : 2
Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 2
Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 2
Сдали 6
Не сдали 4

Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 2
Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 2
Ведите результат (1=сдал, 2=не сдал) : 1
Ведите результат (1=сдал, 2=не сдал) : 2
Сдали 9
Не сдали 1
Повысить оплату за обучение

```

**Рис. 2.11.** Программа C++ для анализа результатов экзамена и примеры ее выполнения (часть 2 из 2)

## 2.11. Операции присваивания

В C++ имеется несколько операций присваивания, позволяющих сокращать запись присваиваемых выражений. Например, оператор

```
c = c + 3;
```

может быть сокращен применением *составной операции сложения*  $+=$ :

```
c += 3;
```

Операция  $+=$  прибавляет значение выражения, записанного справа от операции, к величине переменной, указанной слева, и сохраняет результат в этой переменной. Любой оператор вида

```
переменная = переменная операция выражение;
```

где операция — одна из бинарных операций  $+$ ,  $-$ ,  $*$ ,  $/$  или  $\%$  (или иные операции, которые будут рассмотрены позднее), может быть записан в виде

```
переменная операция = выражение;
```

Таким образом, присваивание  $c += 3$  добавляет 3 к с. Рис. 2.12 показывает арифметические операции присваивания, примеры выражений с этими операциями и их расширенное толкование.

| Операция присваивания                                | Пример     | Пояснение    | Результат присваивания |
|------------------------------------------------------|------------|--------------|------------------------|
| Предположим: int c = 3, d = 5, e = 4, f = 6, g = 12; |            |              |                        |
| $+=$                                                 | $c += 7$   | $c = c + 7$  | $c = 10$               |
| $-=$                                                 | $d -= 4$   | $d = d - 4$  | $d = 1$                |
| $*=$                                                 | $e *= 5$   | $e = e * 5$  | $e = 20$               |
| $/=$                                                 | $f /= 3$   | $f = f / 3$  | $f = 2$                |
| $\% =$                                               | $g \% = 9$ | $g = g \% 9$ | $g = 3$                |

Рис. 2.12. Арифметические операции присваивания

### Совет по повышению эффективности 2.1

Программисты могут писать программы несколько быстрее и компилятор может компилировать их немного быстрее, если использовать «сокращенные» операции присваивания (составные присваивания). Некоторые компиляторы генерируют код, который выполняется быстрее при использовании «сокращенных» операций присваивания.

### Совет по повышению эффективности 2.2

Многие из советов по повышению эффективности, которые мы будем давать в этой книге, приводят к незначительному улучшению, так что читатель может иметь искушение пренебречь ими. Значительное повышение эффективности получается, если такие небольшие улучшения находятся внутри цикла и могут повторяться много раз.

## 2.12. Операции инкремента и декремента

В C++ имеется унарная *операция инкремента* `++` (увеличение на 1) и унарная *операция декремента* `--` (уменьшение на 1), свойства которых приведены на рис. 2.13. Если переменная `c` должна быть увеличена на 1, лучше применить оператор `++`, чем выражения `c=c+1` или `c+=1`. Если операция инкремента или декремента помещена перед переменной, говорят о *префиксной форме записи* инкремента или декремента. Если операция инкремента или декремента записана после переменной, то говорят о *постфиксной форме записи*. При префиксной форме переменная сначала увеличивается или уменьшается на единицу, а затем это ее новое значение используется в том выражении, в котором она встретилась. При постфиксной форме в выражении используется текущее значение переменной, и только после этого ее значение увеличивается или уменьшается на единицу.

Программа на рис. 2.14 демонстрирует различие между префиксной и постфиксной формами операции инкремента `++`. Постфиксная форма записи операции инкремента вызывает увеличение переменной с после того, как она использована в операторе вывода. При префиксной форме переменная `s` изменяется до того, как будет использована в операторе вывода.

Программа выводит на экран значения `s` до и после применения операции `++`. Операция `--` действует аналогично.

| Операция | Название операции            | Пример выражения | Пояснение                                                                                                              |
|----------|------------------------------|------------------|------------------------------------------------------------------------------------------------------------------------|
| +        | префиксная форма инкремента  | <code>++a</code> | Величина <b>a</b> увеличивается на 1 и это новое значение <b>a</b> используется в выражении, в котором оно встретилось |
| +        | постфиксная форма инкремента | <code>a++</code> | В выражении используется текущее значение <b>a</b> , а затем величина <b>a</b> увеличивается на 1                      |
| --       | префиксная форма декремента  | <code>--b</code> | Величина <b>b</b> уменьшается на 1 и это новое значение <b>b</b> используется в выражении, в котором оно встретилось   |
| --       | постфиксная форма декремента | <code>b--</code> | В выражении используется текущее значение <b>b</b> , а затем величина <b>b</b> уменьшается на 1                        |

Рис. 2.13. Операции инкремента и декремента

```
//Префиксная и постфиксная формы операции инкремента
#include <iostream.h>

main()
{
 int c;

 c=5;
 cout << c << endl;
 cout << c++ << endl; // Постфиксная форма инкремента
 cout << c << endl;

 c=5;
 cout << c << endl;
 cout << ++c << endl; // Префиксная форма инкремента
 cout << c << endl;

 return 0; // успешное окончание
}
```

---

5  
5  
6  
  
5  
6  
6

Рис. 2.14. Различие между префиксной и постфиксной формами операции инкремента

### Хороший стиль программирования 2.11

Унарные операции должны размещаться после их operandов без разделяющего пробела.

Три оператора присваивания на рис. 2.11

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

могут быть записаны более кратко в виде:

```
passes += 1;
failures += + 1;
student += 1;
```

или с использованием операции инкремента в префиксной форме:

```
++passes;
++failures;
++student;
```

или в постфиксной форме:

```
passes++;
failures++;
student++;
```

Важно отметить, что когда инкремент или декремент переменной осуществляется в виде отдельного оператора, то префиксная и постфиксная формы приводят к одному результату. И только если переменная появляется в контексте более сложного выражения, тогда префиксная и постфиксная формы приводят к разным результатам.

В операциях инкремента и декремента могут использоваться как операнды только имена простых переменных (в дальнейшем мы увидим, что в операциях инкремента и декремента могут использоваться так называемые L-величины).

#### Типичная ошибка программирования 2.10

Попытка использовать в операции инкремента или декремента операнд, отличный от имени простой переменной, например, выражение  $++(x+1)$  является синтаксической ошибкой.

В таблице на рис. 2.15 приведены сведения о приоритетах и ассоциативности всех операций, описанных к настоящему моменту. Операции приведены сверху-вниз по мере уменьшения их приоритета. Второй столбец указывает ассоциативность операций на каждом уровне приоритета. Отметим, что условная операция (?), унарные операции инкремента (++) и декремента (--), плюс (+), минус (-) и приведения к типу, операции присваивания =, +=, -=, \*=, /= и %= имеют ассоциативность справа налево. Все остальные операции в таблице на рис. 2.15 имеют ассоциативность слева направо. В третьем столбце таблицы приведены названия различных групп операций.

## 2.13. Основы повторения, управляемого счетчиком

Для организации повторения, управляемого счетчиком, требуется задать:

1. Имя управляющей переменной (или счетчика циклов).

2. Начальное значение управляющей переменной.
3. *Приращение* (или *уменьшение*), на которое изменяется управляющая переменная в каждом цикле.
4. Условие проверки, не достигнуто ли конечное значение управляющей переменной (т.е. надо ли продолжать циклы).

Рассмотрим простую программу, показанную на рис. 2.16, которая печатает числа от 1 до 10. Объявление

```
int counter = 1;
```

определяет имя управляющей переменной (*counter*), объявляет переменную как целую, резервируя для нее соответствующее место в памяти, и задает ее начальное значение 1. Объявления, в которых задается начальное значение, являются фактически выполняемыми операторами.

Это объявление и задание начального значения переменной *counter* можно было бы сделать операторами

```
int counter;
counter = 1;
```

Здесь само объявление не является выполняемым оператором, а присваивание осуществляется отдельным оператором. Можно использовать оба эти метода для задания начальных значений переменных.

Оператор

```
++counter
```

*увеличивает* счетчик на 1 при каждом выполнении цикла. Условие продолжения цикла в структуре *while* проверяет, меньше или равно значение управляющей переменной числа 10 (последнего значения, при котором условие истинно). Заметьте, что тело этой структуры *while* выполняется и при значении управляющей переменной, равном 10. Выполнение цикла заканчивается, когда значение управляющей переменной превысит 10 (т.е. переменная *counter* станет равной 11).

| Операция         | Ассоциативность | Тип операций          |
|------------------|-----------------|-----------------------|
| ()               | слева направо   | круглые скобки        |
| + - -- + - (тип) | справа налево   | унарные               |
| * / %            | слева направо   | мультипликативные     |
| + -              | слева направо   | аддитивные            |
| << >>            | слева направо   | поместить в/взять из  |
| < <= > >=        | слева направо   | отношение             |
| == !=            | слева направо   | проверка на равенство |
| ? :              | справа налево   | условная              |
| = += -= *= /= %= | справа налево   | присваивание          |

Рис. 2.15. Приоритеты операций, описанных ранее в данной книге

```
//Повторение, управляемое счетчиком
#include <iostream.h>

main()
{
 int counter = 1; // Задание начального значения

 while (counter <= 10) { // условие повторения
 cout << counter << endl;
 ++counter; // увеличение
 }

 return 0; // успешное окончание
}

1
2
3
4
5
6
7
8
9
10
```

**Рис. 2.16.** Повторение, управляемое счетчиком

Программа рис. 2.16 может быть сделана более компактной, если переменной `counter` задать начальное значение 0 и заменить структуру `while` следующей:

```
while (++couner <= 10)
 cout << counter << endl;
```

Этот код экономит один оператор, поскольку инкремент выполняется непосредственно в условии структуры `while` до того, как это условие проверяется. Этот код также устраняет скобки, заключающие в себе тело `while`, поскольку `while` содержит всего один оператор. Кодирование в подобной сжатой манере дается практикой.

### Типичная ошибка программирования 2.11

Поскольку числа с плавающей запятой являются приближенными, контроль количества выполнений цикла с помощью переменной с плавающей запятой может приводить к неточному значению счетчика и неправильному результату проверки условия окончания.

### Хороший стиль программирования 2.12

Управляйте количеством повторений цикла с помощью целой переменной.

### Хороший стиль программирования 2.13

Записывайте с отступом операторы тела каждой управляющей структуры.

### **Хороший стиль программирования 2.14**

Размещайте пустую строку до и после каждой большой управляющей структуры, чтобы она выделялась в программе.

### **Хороший стиль программирования 2.15**

Слишком большая глубина вложенности может сделать программу трудной для понимания. Как правило, старайтесь избегать более трех уровней отступов.

### **Хороший стиль программирования 2.16**

Вертикальные пробелы над и под управляющими структурами вместе с отступами в телах этих структур по отношению к их заголовкам создают двумерный эффект, облегчающие чтение программы.

## **2.14. Структура повторения for (ЦИКЛ)**

Структура повторения **for** содержит все элементы, необходимые для повторения, управляемого счетчиком. Чтобы проиллюстрировать мощь структуры **for**, давайте перепишем программу рис. 2.16. Результат этого представлен на рис. 2.17. Ниже описана работа этой программы.

Когда структура **for** начинает выполняться, управляющей переменной **counter** задается начальное значение 1. Затем проверяется условие продолжения цикла **counter <= 10**. Поскольку начальное значение **counter** равно 1, это условие удовлетворяется, так что оператор тела структуры печатает значение **counter**, равное 1. Затем управляющая переменная **counter** увеличивается на единицу в выражении **counter++** и цикл опять начинается с проверки условия его продолжения. Поскольку значение **counter** теперь 2, предельная величина не превышена, так что программа снова выполняет тело цикла. Этот процесс продолжается, пока управляющая переменная **counter** не увеличится до 11 — это приведет к тому, что условие продолжения цикла нарушится и повторение прекратится. Выполнение программы продолжится с первого оператора, расположенного после структуры **for** (в данном случае с оператора **return** в конце программы).

Рис. 2.18 в сжатом виде показывает структуру **for** из программы рис. 2.17. Заметьте, что структура **for** «делает все» — она определяет каждый элемент, необходимый для повторения, управляемого счетчиком с управляющей переменной. Если в теле **for** имеется более одного оператора, то для определения тела цикла требуются фигурные скобки.

Отметим, что рис. 2.17 использует условие продолжения цикла **counter <= 10**. Если программист некорректно напишет **counter < 10**, то цикл выполнится всего 9 раз. Это типичная логическая ошибка, называемая *ошибкой занижения (или завышения) на единицу*.

### **Типичная ошибка программирования 2.12**

Использование неправильной операции отношения или использование неправильного конечного значения счетчика цикла в условиях структур **while** или **for** может приводить к ошибке занижения (или завышения) на единицу.

```
//Повторение, управляемое счетчиком, со структурой for

#include <iostream.h>

main()
{
 // задание начального значения, условие повторения и
 // приращение - все это включено в заголовок структуры for
 for (int counter = 1; counter <= 10; counter++)
 cout << counter << endl;

 return 0; // успешное окончание
}
```

Рис. 2.17. Повторение, управляемое счетчиком, со структурой for

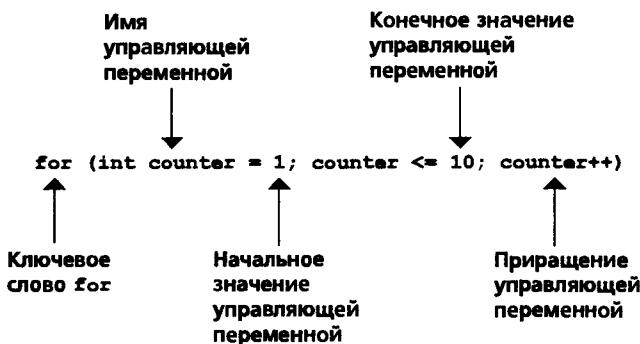


Рис. 2.18. Компоненты типичного заголовка for

### Хороший стиль программирования 2.17

Использование конечного значения управляемой переменной в условиях структур **while** и **for** и использование операции отношения `<=` поможет избежать ошибок занижения на единицу. Например, для цикла, используемого при печати чисел от 1 до 10 условие продолжения цикла следует записать `counter <= 10`, а не `counter < 10` (что является ошибкой занижения на единицу) или `counter < 11` (что тем не менее корректно).

### Общая форма структуры for:

```
for (выражение1; выражение2; выражение3)
 оператор
```

где *выражение1* задает начальное значение переменной, управляемой циклом, *выражение2* является условием продолжения цикла, а *выражение3* изменяет управляемую переменную. В большинстве случаев структуру for можно представить эквивалентной ей структурой while следующим образом:

```
выражение1;

while (выражение2) {
 оператор
 выражение3;
}
```

Исключения из этого правила будут рассмотрены в разделе 2.18.

Иногда *выражение1* и *выражение3* представляются как списки выражений, разделенных запятой. В данном случае запятая используется как *операция запятая* или *операция последования*, гарантирующая, что список выражений будет вычисляться слева направо. *Операция последования* имеет самый низкий приоритет среди всех операций C++. Значение и тип списка выражений, разделенных запятыми, равны значению и типу самого правого выражения в списке. Операция последования наиболее часто используется в структуре **for**. Ее основное назначение — помочь программисту использовать несколько выражений задания начальных значений и (или) несколько выражений приращения переменных. Например, в одной структуре **for** может быть несколько управляющих переменных, которым надо задавать начальное значение и которые надо изменять.

### Хороший стиль программирования 2.18

Помещайте в разделы задания начальных значений и изменения переменных структуры **for** только выражения, относящиеся к управляющей переменной. Манипуляции с другими переменными должны размещаться или до цикла (если они выполняются только один раз подобно операторам задания начальных значений), или внутри тела цикла (если они должны выполняться в каждом цикле, как, например, операторы инкремента или декремента).

Выражения в структуре **for** являются необязательными. Если *выражение2* отсутствует, C++ предполагает, что условие продолжения цикла всегда истинно и таким образом создается бесконечно повторяющийся цикл. Иногда может отсутствовать *выражение1*, если начальное значение управляющей переменной задано где-то в другом месте программы. Может отсутствовать и *выражение3*, если приращение переменной осуществляется операторами в теле структуры **for** или если приращение не требуется. Выражение для приращения переменной в структуре **for** действует так же, как автономный оператор в конце тела **for**. Следовательно, все выражения

```
counter = counter + 1
counter += 1
++counter
counter++
```

эквивалентны в той части структуры **for**, которая определяет приращение. Многие программисты предпочитают форму **counter++**, поскольку приращение срабатывает после выполнения тела цикла. Поэтому постфиксная форма представляется более естественной. Поскольку изменяемая переменная здесь не входит в какое-то выражение, обе формы инкремента равноправны.

В структуре **for** должны применяться точки с запятой.

### Типичная ошибка программирования 2.13

Использование запятых вместо точек с запятой в заголовке структуры **for**.

### **Типичная ошибка программирования 2.14**

Размещение точки с запятой сразу после правой закрывающей скобки заголовка **for** делает тело структуры пустым оператором. Обычно это логическая ошибка.

Части структуры **for** — задание начального значения, условие продолжения цикла и изменение переменной могут содержать арифметические выражения. Например, предположим, что  $x = 2$  и  $y = 10$ . Если  $x$  и  $y$  не изменяются в теле цикла, то оператор

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

эквивалентен оператору

```
for (int j = 2; j <= 80; j += 5)
```

«Приращение» структуры **for** может быть отрицательным (в этом случае в действительности происходит не приращение, а уменьшение переменной, управляющей циклом).

Если условие продолжения цикла с самого начала не удовлетворяется, то операторы тела структуры **for** не выполняются и управление передается оператору, следующему за **for**.

Управляющая переменная иногда печатается или используется в вычислениях в теле структуры **for**, но обычно это делается без изменений ее величины. Чаще управляющая переменная используется только для контроля числа повторений и никогда не упоминается в теле структуры.

### **Хороший стиль программирования 2.19**

Хотя управляющая переменная может изменяться в теле цикла **for**, избегайте делать это, так как такая практика приводит к неявным, неочевидным ошибкам.

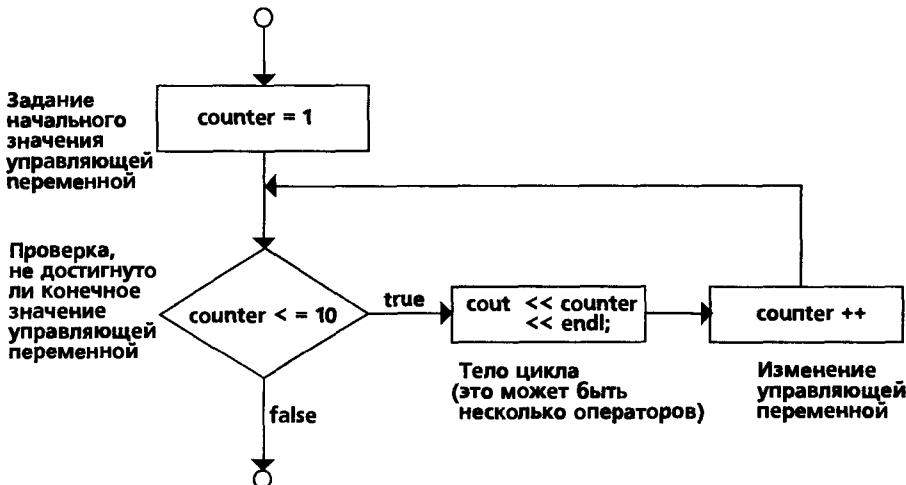
Структура **for** изображается на блок-схеме очень похоже на структуру **while**. Например, блок-схема оператора

```
for (int counter = 1; counter <= 10; counter++)
 cout << counter << endl;
```

приведена на рис. 2.19. Эта блок-схема ясно показывает, что задание начального значения производится только один раз и что изменение переменной происходит каждый раз *после выполнения* оператора тела цикла. Заметьте, что (не считая маленьких окружностей и стрелок) блок-схема содержит только символы прямоугольников и ромбов. Представим себе снова, что программист имеет доступ к глубокому бункеру, содержащему пустые структуры **for** — столько, сколько может потребоваться программисту для пакетирования и вложения в другие управляющие структуры, чтобы структурированно отобразить управляющую логику алгоритма. И затем эти прямоугольники и ромбы заполняются необходимыми действиями и условиями выбора в соответствии с алгоритмом.

## **2.15. Пример использования структуры **for****

Следующие примеры покажут способы изменения управляющей переменной в структуре **for**. В каждом случае мы напишем соответствующий

Рис. 2.19. Блок-схема типичной структуры **for**

заголовок **for**. Обратите внимание на отличие операций отношения в циклах с уменьшением управляющей переменной.

а) Изменение управляющей переменной от 1 до 100 с шагом 1.

```
for (int i = 1; i <= 100; i++)
```

б) Изменение управляющей переменной от 100 до 1 с шагом -1 (с уменьшением на 1).

```
for (int i = 100; i >= 1; i--)
```

#### Типичная ошибка программирования 2.15

Использование несоответствующей операции отношения в условии продолжения цикла при счете циклов сверху вниз (например, использование  $i \leq 1$  при счете циклов сверху до 1).

с) Изменение управляющей переменной от 7 до 77 с шагом 7.

```
for (int i = 7; i <= 77; i += 7)
```

д) Изменение управляющей переменной от 20 до 2 с шагом -2.

```
for (int i = 20; i >= 2; i -= 2)
```

е) Изменение управляющей переменной в следующей последовательности: 2, 5, 8, 11, 14.

```
for (int j = 2; j <= 20; j += 3)
```

ф) Изменение управляющей переменной в следующей последовательности: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for (int j = 99; j >= 0; j -= 11)
```

Следующие два примера содержат простые приложения структуры **for**. Программа на рис. 2.20 использует структуру **for** для суммирования всех четных чисел от 2 до 100.

```
// Суммирование с помощью for
#include <iostream.h>

main()
{
 int sum = 0;

 for (int number = 2; number <= 100; number += 2)
 sum += number;
 cout << "Сумма равна " << sum << endl;

 return 0;
}
```

**Сумма равна 2550**

**Рис. 2.20.** Суммирование с помощью **for**

Отметим, что тело структуры **for** на рис. 2.20 в действительности может быть включено в правую часть заголовка **for** с помощью операции последования (запятая):

```
for (int number = 2; number <= 100; sum += number, number += 2);
```

Задание начального значения **sum = 0** также может быть включено в раздел задания начального значения структуры **for**.

### Хороший стиль программирования 2.20

Хотя операторы, предшествующие **for** и операторы тела **for** могут часто включаться в заголовок **for**, избегайте делать это, чтобы не затруднять чтение программы.

### Хороший стиль программирования 2.21

Ограничивайте, если возможно, размер заголовка управляющей структуры так, чтобы он умещался на одной строке.

Следующий пример вычисляет с помощью структуры **for** сложный процент. Рассмотрим следующую постановку задачи:

*Некто внес \$1000.00 на депозитный счет под 5 процентов годовых. Предполагая, что доход оставляется на депозите, рассчитать и напечатать суммы денег на счете в конце каждого года на протяжении 10 лет. Использовать для расчета следующую формулу:*

$$a=p(1+r)^n$$

*где*

*p — первоначальный (основной) вклад*

*r — ставка годового дохода*

*n — число лет*

*a — сумма на депозите к концу n-го года.*

Эта задача подразумевает использование цикла, в котором будут рассчитываться ежегодные суммы на депозите в течение 10-и лет. Решение приведено на рис. 2.21.

Структура `for` выполняет операторы своего тела 10 раз, причем управляющая переменная изменяется от 1 до 10 с шагом 1. Хотя C++ не имеет операции возвведения в степень, но ее можно осуществить с помощью функции `pow` из стандартной библиотеки. Функция `pow` имеет два аргумента типа `double` и возвращает результат типа `double`. Тип `double` — это тип с плавающей точкой во многом похожий на `float`, но переменные типа `double` могут хранить значения, много большие, чем переменные типа `float`, и с большей точностью. Константы (такие, как 1000.0 и .05 на рис. 2.21) требуют в C++ использования типа `double`.

Эта программа не будет компилироваться без включения `math.h`. Функция `pow` требует аргументов типа `double`. Заметьте, что `year` — целое. Файл `math.h` содержит информацию, которая дает указание компилятору преобразовать величину `year` во временное представление `double` перед вызовом функции. Эта информация содержится в *прототипе функции pow*. Прототипы функций будут рассмотрены в главе 3. Мы дадим сведения о функции `pow` и других функциях из библиотеки математических функций в главе 3.

```
// Расчет сложного процента

#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main()
{
 double amount, principal = 1000.0, rate = .05;

 cout << "Год" << setw(21) << "Сумма депозита" << endl;

 for (int year = 1; year <= 10; year++) {
 amount = principal * pow(1.0 + rate, year);
 cout << setw(3) << year
 << setiosflags(ios::fixed | ios::showpoint)
 << setw(21) << setprecision(2) << amount << endl;
 }

 return 0;
}
```

---

| Год | Сумма депозита |
|-----|----------------|
| 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. 2.21. Расчет сложного процента с помощью `for`

### Типичная ошибка программирования 2.16

Забывают включить файл **math.h** в программы, использующие библиотеку математических функций.

Обратите внимание, что мы объявили переменные **amount**, **principal** и **rate** типа **double**. Мы сделали это для простоты, поскольку мы имеем дело с дробной частью долларов и нам нужен тип с плавающей запятой. К сожалению, это может вызвать затруднения. Поясним, к каким недоразумениям может приводить использование **float** или **double** при представлении суммы долларов (в предположении, что печать осуществляется с **setprecision(2)**). Две суммы долларов типа **float**, хранящиеся в памяти, могут иметь значения 14.234 (печатается как 14.23) и 18.673 (печатается как 18.67). Когда эти суммы складываются, результат внутри компьютера получается 32.907, что будет напечатано как 32.91. Таким образом, печать может иметь вид:

```
14.23
+ 18.67

32.91
```

но пользователь, складывающий эти числа в том виде, в котором они напечатаны, ожидает получить сумму 32.90!

### Хороший стиль программирования 2.21

Не используйте переменные типов **float** и **double** для денежных расчетов. Неточность чисел с плавающей запятой может привести к ошибкам, которые проявятся в итоге в неправильной сумме денег. В качестве упражнения попробуйте использовать для денежных расчетов целые числа. Заметим: библиотека классов C++ позволяет должностным образом осуществлять денежные расчеты.

### Оператор вывода

```
cout << setw(3) << year
 << setiosflags(ios::fixed | ios::showpoint)
 << setw(21) << setprecision(2) << amount << endl;
```

печатает значения переменных **year** и **amount** в формате, заданном параметризованными манипуляторами потока **setw**, **setiosflags** и **setprecision**. Обращение **setw(3)** определяет, что следующая выходная величина будет напечатана с шириной (размером) поля 3, т.е. ее значение будет содержать по крайней мере 3 символьных позиции. Если длина выходной величины менее 3 символов, она по умолчанию будет выравнена в поле по правому символу. Если длина выходной величины более 3, размер поля будет увеличен, чтобы вместить полную величину. Можно использовать обращение **setiosflags(ios::left)** для того, чтобы задать выравнивание по левому символу.

Остальная часть задания формата в приведенном операторе вывода указывает, что переменная **amount** должна печататься как значение с фиксированной точкой и с обязательной печатью этой точки (задается выражением **setiosflags(ios::fixed | ios::showpoint)**) с правым выравниванием в поле размером в 21 символ (задается выражением **setw(21)**) и с двумя цифрами после десятичной точки (задается выражением **setprecision(2)**). Мы подробнее рас-

смотрим богатые возможности C++ по форматированию ввода-вывода в главе 11.

Заметьте, что вычисление выражения `1.0 + rate`, являющегося аргументом функции `pow`, производится в теле оператора `for`. В действительности вычисление этого выражения дает одинаковый результат в каждом цикле, так что его повторные вычисления расточительны.

### **Совет по повышению эффективности 2.3**

Избегайте размещать выражения, значение которых не изменяется, внутри цикла. Но даже, если вы сделаете это, то многие современные сложные оптимизирующие компиляторы автоматически разместят подобные выражения вне цикла во время генерации машинного кода.

## **2.16. Структура множественного выбора switch**

Мы рассмотрели структуру с единственным выбором `if` и структуру с двойным выбором `if/else`. Но порой алгоритм может содержать ряд альтернативных решений, причем некоторую переменную (или выражение) надо проверять отдельно для каждого постоянного целого значения, которое она может принимать; в зависимости от результатов этой проверки должны выполняться различные действия. Для принятия подобных решений в C++ имеется структура множественного выбора `switch`.

Структура `switch` состоит из ряда меток `case` и необязательной метки `default` (умолчание). Программа на рис. 2.22 использует `switch` для расчета числа различных буквенных оценок, полученных студентами на экзамене.

В этой программе пользователь вводит буквенные оценки. Внутри заголовка `while`

```
while ((grade = cin.get()) != EOF)
```

сначала выполняется присваивание, заключенное в скобки. Функция `cin.get()` читает один символ, введенный с клавиатуры, и сохраняет его в целой переменной `grade`. Использованная в `cin.get()` нотация с точкой будет объяснена в главе 6, «Классы». Обычно символы хранятся в переменной типа `char`. Однако важной особенностью C++ является то, что символы могут храниться в любом целом типе данных, поскольку они представляются в компьютере как однобайтовое целое. Таким образом, мы можем трактовать символ или как целое, или как символ в зависимости от его использования. Например, оператор

```
cout << "Символ (" << 'a' << ") имеет значение "
<< (int) 'a' << endl;
```

напечатает символ `a` и его целочисленное представление:

`Символ (a) имеет значение 97`

Целое число 97 является численным отображением символа в компьютере. Многие компьютеры сегодня используют *множество символов ASCII* (American Standard Code for Information Interchange), в котором букве '`a`' в нижнем регистре соответствует число 97. Список символов ASCII и их десятичных значений представлен в приложении B.

```

// Подсчет числа буквенных оценок
#include <iostream.h>

main()
{
 int grade;
 int aCount = 0, bCount = 0, cCount = 0,
 dCount = 0, fCount = 0;

 cout << "Введите буквенную оценку." << endl
 << "Введите символ EOF по окончании ввода." << endl;

 while ((grade = cin.get()) != EOF) {

 switch (grade) { // switch, вложенный в while

 case 'A': case 'a': // Grade равна A в верхнем регистре
 ++aCount; // или а в нижнем регистре.
 break;
 case 'B': case 'b': // Grade равна B в верхнем регистре
 ++bCount; // или б в нижнем регистре.
 break;
 case 'C': case 'c': // Grade равна C в верхнем регистре
 ++cCount; // или с в нижнем регистре.
 break;
 case 'D': case 'd': // Grade равна D в верхнем регистре
 ++dCount; // или d в нижнем регистре.
 break;
 case 'F': case 'f': // Grade равна F в верхнем регистре
 ++fCount; // или f в нижнем регистре.
 break;
 case '\n': case ' ': // этот ввод игнорируется
 break;

 default: // при любых других символах
 cout << "Введена неправильная буквенная оценка."
 << " Введите новую оценку." << endl;
 break;
 }

 cout << endl << "Количество различных оценок:"
 << endl << "A: " << aCount << endl << "B: " << bCount
 << endl << "C: " << cCount << endl << "D: " << dCount
 << endl << "F: " << fCount << endl;
 }

 return 0;
}

```

**Рис. 2.22.** Пример использования **switch** (часть 1 из 2)

```
Введите буквенную оценку.
Введите символ EOF по окончании ввода.
A
B
C
C
A
D
F
C
E
Введена неправильная буквенная оценка. Введите новую оценку.
D
A
B

Количество различных оценок:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Рис. 2.22. Пример использования **switch** (часть 2 из 2)

Оператор присваивания в ряде случаев можно рассматривать как единое целое, как некое выражение, значение которого равно значению, присвоенному переменной слева от символа `=`. Таким образом, значение присваивания `grade = cin.get()` равно значению, возвращенному функцией `cin.get()` и присвоенному переменной `grade`.

Тот факт, что операторы присваивания имеют значение, можно использовать для задания одного и того же начального значения сразу нескольким переменным. Например, в операторе

```
a = b = c = 0;
```

сначала выполняется присваивание `c = 0` (так как операция присваивания `=` имеет ассоциативность справа налево). Затем переменной `b` присваивается значение присваивания `c = 0` (которое равно 0). Затем переменной `a` присваивается значение присваивания `b = (c = 0)` (которое тоже равно 0). В нашей программе значение присваивания `grade = cin.get()` сравнивается со значением `EOF` — символа, который установлен для «конца файла». Мы используем `EOF` (который обычно имеет значение `-1`) как значение метки. Пользователь нажимает зависящую от системы комбинацию клавиш, означающую «конец файла», т.е. как бы говорит: «У меня нет больше данных для ввода». `EOF` — символическая целая константа, определенная в головном файле `<iostream.h>`. Если значение, присвоенное переменной `grade`, равно `EOF`, то программа заканчивается. В данной программе мы выбрали для символа представление `int`, так как `EOF` — целая величина (повторим, что обычно она равна `-1`).

### Замечание по мобильности 2.1

Комбинация клавиш для ввода признака конца файла зависит от системы.

### Замечание по мобильности 2.2

Проверка на символическую константу **EOF**, а не на -1 делает программу более мобильной. Стандарт ANSI устанавливает, что **EOF** имеет целое отрицательное значение (но не обязательно -1). Так что **EOF** может иметь различные значения в разных системах.

В системе UNIX и многих других признак конца файла вводится комбинацией

*<ctrl-d>*

в последней строке. Эта нотация означает одновременное нажатие клавиши **ctrl** и клавиши **d**. В других системах, таких, как VAX VMS корпорации DEC или MS-DOS корпорации Microsoft признак конца файла вводится нажатием

*<ctrl-z>*

Пользователь вводит оценки с клавиатуры. Когда он нажимает клавишу возврата каретки или ввода, символы читаются функцией **cin.get()** по одному за раз. Если введенный символ не признак конца файла, начинает работать структура **switch**. За ключевым словом **switch** следует в скобках имя переменной **grade**. Это так называемое *управляющее выражение*. Предположим, пользователь ввел в качестве оценки букву С. С автоматически сравнивается с каждым условием **case** в структуре **switch**. Если встречается совпадение (**case 'C' :**), то выполняется оператор, следующий за этой меткой **case**. В случае буквы С переменная **cCount** увеличивается на 1 и работа структуры **switch** немедленно завершается по оператору **break**.

Оператор **break** вызывает передачу программного управления на первый оператор после структуры **switch**. Оператор **break** используется потому, что в противном случае условия **case** в операторе **switch** работают совместно. Если везде в структуре **switch** не использовать **break**, тогда каждый раз, когда одно из условий **case** удовлетворяется, будут выполняться операторы всех последующих меток **case**. Если ни одно условие не выполнено, то выполняются операторы после метки **default** (умолчание), печатающие в нашей программе сообщение об ошибке.

После каждой метки **case** может быть предусмотрено одно или более действий. Структура **switch** отличается от всех других структур тем, что при нескольких действиях после **case** не требуется заключать их в фигурные скобки. В общем случае структура множественного выбора **switch** при использовании **break** в каждом разделе **case** соответствует блок-схеме, приведенной на рис. 2.23.

Из этой блок-схемы видно, что каждый оператор **break** в конце **case** вызывает немедленный выход из структуры **switch**. Отметим снова, что (не считая маленьких окружностей и линий связи) блок-схема содержит только символы прямоугольников и ромбов. Представьте себе опять, что программист имеет доступ к глубокому бункеру, содержащему пустые структуры **switch** — столько, сколько может потребоваться программисту для складывания их пачками и вложения в другие управляющие структуры, чтобы структурированно отобразить управляющую логику алгоритма. И затем эти прямоугольники и ромбы заполняются необходимыми действиями и условиями выбора в соответствии с алгоритмом. Вложенные управляющие структуры — дело обычное, но вложенные структуры **switch** в программах встречаются редко.

### Типичная ошибка программирования 2.17

Забывают вставить оператор **break**, когда он нужен в структуре **switch**.

### Типичная ошибка программирования 2.18

Пропуск пробела между ключевым словом **case** и целым значением, которое проверяется в структуре **switch**, может вызвать логическую ошибку. Например, запись `case3:` вместо `case 3:` просто создаст неиспользуемую метку (мы поговорим об этом подробнее в главе 18). Дело в том, что в этой структуре **switch** не будут совершены соответствующие действия, когда управляющее выражение **switch** будет иметь значение 3.

### Хороший стиль программирования 2.23

Вставляйте метку **default** в оператор **switch**. Случай неудачных проверок в операторе **switch** без метки **default** будут игнорироваться. Включение метки **default** фиксирует внимание программиста на необходимости обрабатывать исключительную ситуацию. Но бывают ситуации, в которых никакой обработки по метке **default** не требуется.

### Хороший стиль программирования 2.24

Хотя предложения **case** и **default** могут размещаться в структуре **switch** в произвольном порядке, стоит учесть практику качественного программирования — помещать **default** в конце.

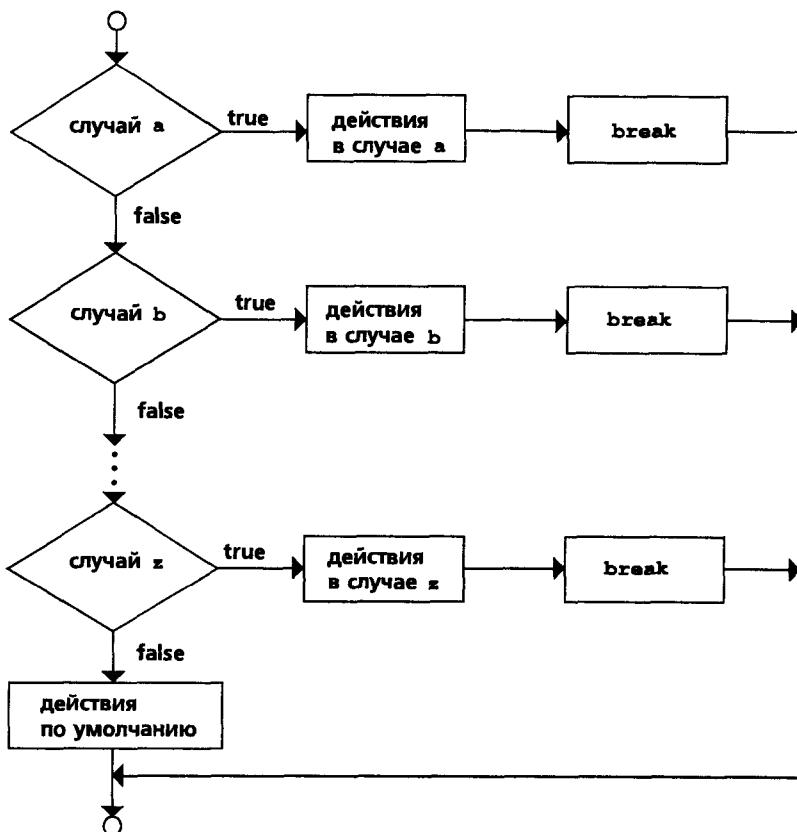


Рис. 2.23. Структура множественного выбора **switch**

### Хороший стиль программирования 2.25

Если в структуре **switch** предложение **default** помещено последним в списке, то оператор **break** в нем не требуется. Но некоторые программисты включают **break** и тут для четкости и для симметрии с другими случаями.

В структуре **switch** на рис. 2.22 строки

```
case '\n': case ' ':
 break;
```

позволяют программе пропускать символы новой строки и пробелы. Чтение по одному символу за раз создает некоторые проблемы. Чтобы программа могла прочитать символы, они должны быть посланы в компьютер нажатием клавиши возврат каретки (ввод) на клавиатуре. Это вызывает во входном потоке символ новой строки после символа, который мы хотим ввести. Часто этот символ новой строки должен быть специально обработан, чтобы работа программы была корректной. Включая указанный случай в нашу структуру **switch**, мы предотвращаем сообщения об ошибках, которые печатались бы каждый раз по метке **default** при вводе новой строки или пробела.

### Типичная ошибка программирования 2.19

Отсутствие обработки символа новой строки при вводе, если символы читаются по одному, может привести к логической ошибке.

### Хороший стиль программирования 2.26

Не забывайте обеспечить обработку возможности появления во входном потоке символа перехода на новую строку и других символов-разделителей, если обрабатываете по одному символу за раз.

Отметим, что последовательное перечисление нескольких меток (например, `case 'D' : case 'd' :` на рис. 2.22) просто означает, что для каждого из этих случаев должны выполняться одни и те же действия.

Когда используете структуру **switch**, помните, что она может применяться только для проверки на совпадение с *константным целым выражением*, т.е. с любой комбинацией символьных и целых констант, которая имеет целое постоянное значение. Символьная константа представляется как соответствующий символ, заключенный в одиночные кавычки, например, 'A'. Целая константа — просто целое число.

Когда мы дойдем до части книги, посвященной объектно-ориентированному программированию, мы представим более элегантный способ реализации логики **switch**. Мы будем использовать технологию, называемую полиморфизмом, для создания программ, которые часто более ясные, более удобные для сопровождения и легче расширяются, чем программы, использующие логику **switch**.

Машинно-независимый язык типа C++ должен иметь гибкость относительно размеров типов данных. Различные приложения могут требовать целые данные различного размера. C++ имеет несколько типов для представления целых чисел. Диапазон целых значений для каждого типа данных зависит от типа конкретного компьютера. В дополнение к типам **int** и **char** C++ имеет типы **short** (сокращение от **short int**) и **long** (сокращение от

`long int`). Минимальный диапазон значений для целых типа `short` равен 32767. Для огромного большинства вычислений с целыми числами достаточно типа `long`. Минимальный диапазон значений для целых типа `long` равен 2147483647. В большинстве компьютеров `int` эквивалентен или `short`, или `long`. Диапазон значений для целых `int` по крайней мере такой же, как для `short`, и не больше, чем для `long`. Данные типа `char` могут использоваться для представления любых символов из множества символов компьютера. Тип `char` можно также использовать для представления небольших целых.

### Замечание по мобильности 2.3

Поскольку размер типа `int` варьируется от системы к системе, используйте тип `long`, если вы предусматриваете обработку целых, значения которых могут лежать вне диапазона 32767, и вы, вероятнее всего, сможете выполнять свою программу на нескольких различных компьютерных системах.

### Совет по повышению эффективности 2.4

В ситуациях, где важна эффективность, где жесткие требования к памяти или критична скорость исполнения программы, может оказаться желательным использовать целые минимальных размеров.

## 2.17. Структура повторения `do/while`

Структура повторения `do/while` похожа на структуру `while`. В структуре `while` условие продолжения циклов проверяется в начале цикла, до того, как выполняется тело цикла. В структуре `do/while` проверка условия продолжения циклов производится *после* того, как тело цикла выполнено, следовательно, тело цикла будет выполнено по крайней мере один раз. Когда `do/while` завершается, выполнение программы продолжается с оператора, следующего за предложением `while`. Отметим, что в структуре `do/while` нет необходимости использовать фигурные скобки, если тело состоит только из одного оператора. Но фигурные скобки обычно все же ставят, чтобы избежать путаницы между структурами `while` и `do/while`. Например,

```
while (условие)
```

обычно рассматривается как заголовок структуры `while`. Структура `do/while` без фигурных скобок и с единственным оператором в теле имеет вид

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

что может привести к путанице. Последняя строка — `while (условие);` может ошибочно интерпретироваться как заголовок структуры `while`, содержащий пустой оператор. Таким образом, чтобы избежать путаницы, структура `do/while` даже с одним оператором часто записывается в виде

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

### Хороший стиль программирования 2.27

Некоторые программисты всегда включают фигурные скобки в структуру **do/while**, даже если в них нет необходимости. Это помогает устраниТЬ двусмысленность, проЙСТекающую из совпадения предложений структуры **while** и структуры **do/while**, содержащей один оператор.

### Типичная ошибка программирования 2.20

Если условие продолжения цикла в структурах **while**, **for** или **do/while** никогда не становится ложным, то возникает зацикливание. Чтобы предотвратить это, убедитесь, что нет точки с запятой сразу после заголовка структуры **while**. В цикле, управляемом счетчиком, убедитесь, что управляющая переменная увеличивается (или уменьшается) в теле цикла. В цикле, управляемом меткой, убедитесь, что значение метки в конце концов будет введено.

Программа на рис. 2.24 использует структуру **do/while**, чтобы напечатать числа от 1 до 10. Обратите внимание, что к управляющей переменной **counter** в проверке окончания цикла применяется инкремент в префиксной форме. Отметьте также использование фигурных скобок, заключающих единственный оператор в теле **do/while**.

Блок-схема структуры **do/while** приведена на рис. 2.25. Она наглядно показывает, что условие продолжения циклов не проверяется, пока тело цикла не выполнится хотя бы один раз. Снова отметим, что (не считая маленьких окружностей и линий связи) блок-схема содержит только символы прямоугольников и ромбов. Снова представьте себе, что программист имеет доступ к глубокому бункеру, содержащему пустые структуры **do/while** — столько, сколько может потребоваться программисту для складывания их пачками и вложения в другие управляющие структуры, чтобы структурированно отобразить управляющую логику алгоритма. И снова эти прямоугольники и ромбы заполняются необходимыми действиями и условиями выбора в соответствии с алгоритмом.

```
// Применение структуры повторения do/while
#include <iostream.h>

main()
{
 int counter = 1;

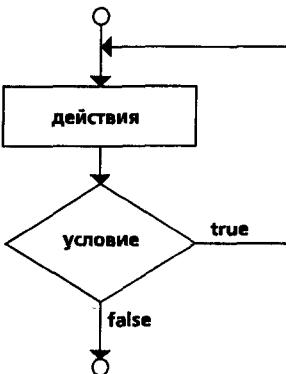
 do {
 cout << counter << " ";
 } while (++counter <= 10);

 return 0;
}
```

---

1 2 3 4 5 6 7 8 9 10

Рис. 2.24. Применение структуры **do/while**

Рис. 2.25. Структура повторения `do/while`

## 2.18. Операторы `break` и `continue`

Операторы `break` и `continue` изменяют поток управления. Когда оператор `break` выполняется в структурах `while`, `for`, `do/while` или `switch`, происходит немедленный выход из структуры. Программа продолжает выполнение с первого оператора после структуры. Обычное назначение оператора `break` — досрочно прерывать цикл или пропустить оставшуюся часть структуры `switch` (как на рис. 2.22). Рис. 2.26 демонстрирует прерывание в структуре повторения `for`. Когда структура `if` определяет, что `x` стал равен 5, выполняется оператор `break`. Это вызывает окончание работы оператора `for` и программа продолжает выполнение с `cout` после `for`. Цикл выполняется полностью только четыре раза.

Оператор `continue` в структурах `while`, `for` или `do/while` вызывает пропуск оставшейся части тела структуры и начинается выполнение следующей итерации цикла. В структурах `while` и `do/while` немедленно после выполнения оператора `continue` производится проверка условия продолжения цикла. В структуре `for` выполняется выражение приращения, а затем осуществляется проверка условия продолжения. Ранее мы установили, что в большинстве случаев структура `while` может использоваться вместо `for`. Единственным исключением является случай, когда выражение приращения в структуре `while` следует за оператором `continue`. В этом случае приращение не выполняется до проверки условия продолжения цикла и структура `while` работает не так, как `for`. В программе на рис. 2.27 оператор `continue` используется в структуре `for`, чтобы пропустить оператор вывода и начать следующую итерацию цикла.

### Хороший стиль программирования 2.28

Некоторые программисты чувствуют, что `break` и `continue` нарушают структурное программирование. А поскольку результат работы этих операторов, как мы скоро увидим, может быть достигнут технологией структурного программирования, то эти программисты не применяют `break` и `continue`.

```
// Применение оператора break в структуре for
#include <iostream.h>

main()
{
 for (int x = 1; x <= 10; x++) {
 if (x == 5)
 break; // прерывание цикла только при x == 5
 cout << x << " ";
 }

 cout << endl << "Цикл прерван при x == " << x << endl;
 return 0;
}
1 2 3 4
Цикл прерван при x == 5
```

**Рис. 2.26.** Применение оператора **break** в структуре **for**

```
// Применение оператора continue в структуре for
#include <iostream.h>

main()
{
 for (int x = 1; x <= 10; x++) {
 if (x == 5)
 continue; // пропуск оставшейся части цикла
 // только при x == 5
 cout << x << " ";
 }

 cout << endl << "Использован continue для пропуска печати при x == 5"
 << endl;
 return 0;
}

1 2 3 4 6 7 8 9 10
Использован continue для пропуска печати при x == 5
```

**Рис. 2.27.** Применение оператора **continue** в структуре **for**

### Совет по повышению эффективности 2.5

При надлежащем использовании операторы **break** и **continue** выполняются быстрее, чем соответствующие приемы структурного программирования, которые мы скоро изучим.

### Замечание по технике программирования 2.8

Имеется некоторое противоречие между стремлением к технике высококачественного программирования и стремлением к наилучшей эффективности программного обеспечения. Часто достижение одной из этих целей достигается за счет другой.

## 2.19. Логические операции

Пока мы изучали только *простые условия*, такие, как `counter <= 10`, `total > 1000` и `number != sentinelValue`. Мы выражали эти условия в терминах операций отношения `>`, `<`, `>=`, `<=` и операций проверки на равенство `==` и `!=`. Каждое решение принималось на основе одного условия. Если мы хотели в процессе принятия решения проверять несколько условий, мы должны были реализовывать эти проверки отдельными операторами или вложенными структурами `if` или `if/else`.

C++ предоставляет нам *логические операции*, которые могут использоваться для формирования сложных условий путем комбинирования простых условий. Логическими операциями являются `&&` (логическое *И*), `||` (логическое *ИЛИ*) и `!` (логическое *НЕ*, называемое также *логическое отрицание*). Рассмотрим примеры каждой из них.

Предположим, мы хотим обеспечить в некоторой точке программы, чтобы определенный путь расчета выбирался только в случае, когда два условия *одновременно истинны*. Тогда мы можем применить логическую операцию `&&`, например:

```
if (gender == 1 && age >= 65)
 ++seniorFemales;
```

Этот оператор `if` содержит два простых условия. Условие `gender == 1` может проверяться, например, чтобы определить, является ли данное лицо женской. Условие `age >= 65` проверяется, чтобы определить, является ли человек пожилым гражданином. Проверка этих двух простых условий выполняется первой, поскольку приоритет обеих операций `==` и `>=` выше, чем приоритет операции `&&`. Затем оператор `if` рассматривает комбинацию этих условий

```
(gender == 1) && (age >= 65)
```

Таблица на рис. 2.28 поясняет действие операции `&&`. Эта таблица показывает все четыре возможных сочетания комбинаций нулевого (`false` — ложь) и ненулевого (`true` — истина) значений первого и второго выражений. Подобные таблицы часто называются *таблицами истинности*. C++ воспринимает как 0 или как 1 все выражения, содержащие операции отношения, проверки на равенство или логические операции. Хотя C++ возвращает 1, если результат равен `true`, он воспринимает как `true` любое ненулевое значение.

| выражение 1 | выражение 2 | выражение 1 && выражение 2 |
|-------------|-------------|----------------------------|
| 0           | 0           | 0                          |
| 0           | ненулевое   | 0                          |
| ненулевое   | 0           | 0                          |
| ненулевое   | ненулевое   | 1                          |

Рис. 2.28. Таблица истинности операции `&&` (логическое И)

Теперь давайте рассмотрим операцию `||` (логическое *ИЛИ*). Предположим, мы хотим обеспечить в некоторой точке программы, чтобы определенный путь расчета выбирался только в случае, когда хотя бы *одно* из двух условий истинно. В этом случае мы можем применить логическую операцию `||`, например:

```
if (semesterAverage >= 90 || finalExam >= 90)
 cout << "Оценка студента - А" << endl;
```

Этот оператор также содержит два простых условия. Условие `semesterAverage >= 90` проверяется, чтобы определить, заслужил ли студент оценки А за курс в результате постоянной работы в течение семестра. Условие `finalExam >= 90` проверяется, чтобы определить, заслужил ли студент оценки А за курс вследствие выдающихся результатов на заключительном экзамене. Затем оператор `if` рассматривает комбинацию этих условий

```
semesterAverage >= 90 || finalExam >= 90
```

и присуждает студенту оценку «А», если любое из этих условий или оба они истины. Отметим, что сообщение «Оценка студента - А» не печатается только, если оба простых условия ложны (если их значения равны 0). На рис. 2.29 приведена таблица истинности для логической операции ИЛИ (`||`).

Операция `&&` имеет более высокий приоритет, чем операция `||`. Обе эти операции имеют ассоциативность слева направо. Выражение, содержащее операции `&&` и `||`, оценивается только до тех пор, пока его истинность или ложность не станет очевидной. Таким образом, анализ выражения

```
gender == 1 && age >= 65
```

будет немедленно остановлен, если значение `gender` не равно 1 (т.е. условие в целом заведомо ложно), и продолжится, если значение `gender` равно 1 (т.е. в целом может оказаться истинным, если будет истинным условие `age >= 65`).

### Типичная ошибка программирования 2.21

В выражениях, использующих операцию `&&`, может оказаться, что одно условие – назовем его зависимым – может требовать, чтобы другое условие было значимо для оценке зависимого условия. В этом случае зависимое условие должно быть помещено после другого условия, в противном случае может произойти ошибка.

### Совет по повышению эффективности 2.6

В выражениях, использующих операцию `&&`, если отдельные условия независимы друг от друга, записывайте комбинированное условие так, чтобы самым левым было то простое условие, которое вероятнее всего окажется ложным. В выражениях, использующих операцию `||`, записывайте комбинированное условие так, чтобы самым левым было то простое условие, которое вероятнее всего окажется истинным. Это может сократить время выполнения программы.

| выражение 1 | выражение 2 | выражение 1    выражение 2 |
|-------------|-------------|----------------------------|
| 0           | 0           | 0                          |
| 0           | ненулевое   | 1                          |
| ненулевое   | 0           | 1                          |
| ненулевое   | ненулевое   | 1                          |

Рис. 2.29. Таблица истинности операции `||` (логическое ИЛИ)

C++ содержит операцию `!` (логическое отрицание), чтобы программист мог изменить значение условия на «противоположное». В отличие от операций `&&` и `||`, которые комбинируют два условия (`и`, следовательно, являются бинарными операциями), операция отрицания имеет в качестве операнда только одно условие (`и`, следовательно, является унарной операцией). Операция логического отрицания помещается перед соответствующим условием, когда мы хотим выбрать некоторый вариант расчета в случае, если первоначально (без учета операции логического отрицания) это условие ложно. Приведем пример фрагмента программы:

```
if (!(grade == sentineValue))
 cout << "Следующая оценка - " << grade << endl;
```

Скобки, в которые помещено условие `grade == sentineValue` необходимы, так как операция логического отрицания имеет более высокий приоритет, чем операция проверки равенства. На рис. 2.30 приведена таблица истинности операции логического отрицания.

| выражение | логическое отрицание выражения |
|-----------|--------------------------------|
| 0         | 1                              |
| 1         | 0                              |

Рис. 2.30. Таблица истинности операции `!` (логическое отрицание)

В большинстве случаев программист может избежать применения логического отрицания, изменив выражение условия с помощью соответствующих операций отношения и проверки равенства. Например, предыдущий оператор может быть записан в виде

```
if (grade != sentineValue)
 cout << "Следующая оценка - " << grade << endl;
```

Подобная гибкость часто может помочь программисту выразить условие в наиболее естественном и удобном виде.

Таблица на рис. 2.31 показывает приоритеты и ассоциативность операций C++, рассмотренных к настоящему моменту. Операции представлены сверху вниз в порядке их старшинства.

## 2.20. Ошибки случайной подмены операций проверки равенства (`==`) и присваивания (`=`)

Есть один тип ошибок, который программисты на C++, независимо от их квалификации, делают так часто, что мы решили уделять ему отдельный раздел. Речь идет о случайной подмене операций проверки равенства (`==`) и присваивания (`=`). Подобные подмены очень неприятны, поскольку обычно они не приводят к синтаксическим ошибкам. Ошибочные операторы обычно нормально компилируются, программа запускается на выполнение, но результаты получаются ошибочными из-за логических ошибок выполнения.

| Операция         | Ассоциативность | Тип операций           |
|------------------|-----------------|------------------------|
| ()               | слева направо   | круглые скобки         |
| + - + - ! (тип)  | справа налево   | унарные                |
| * / %            | слева направо   | многипликативные       |
| +                | слева направо   | аддитивные             |
| << >>            | слева направо   | поместить в/взять из   |
| < <= > >=        | слева направо   | отношение              |
| == !=            | слева направо   | проверка на равенство  |
| &&               | слева направо   | логическое И           |
|                  | слева направо   | логическое ИЛИ         |
| ? :              | справа налево   | условная               |
| = += -= *= /= %= | справа налево   | присваивание           |
| ,                | слева направо   | запятая (последование) |

Рис. 2.31. Приоритеты и ассоциативность операций

В C++ существуют два обстоятельства, создающих подобные проблемы. Одно из них заключается в том, что любое выражение, имеющее некоторое значение, может использоваться в решающей части любой управляющей структуры. Если значение выражения равно 0, оно трактуется как `false`, если же значение не равно 0, оно трактуется как `true`. Второе обстоятельство связано с тем, что в C++ присваивание, как уже говорилось, имеет значение, а именно — то значение, которое присваивается переменной, расположенной слева от операции присваивания. Например, предположим, что мы намерены написать

```
if (payCode == 4)
 cout << "Вы получили премию!" << endl;
```

но случайно написали

```
if (payCode = 4)
 cout << "Вы получили премию!" << endl;
```

Первый оператор `if` должным образом присуждает премию тому, чей `payCode` равен 4. Второй оператор `if` (ошибочный) оценивает значение операции присваивания в условии `if` постоянным значением 4. Поскольку любое не-нулевое значение интерпретируется как `true`, условие в данном операторе `if` всегда истинно, и премии будут получать все, независимо от действительной величины `payCode`! Более того, переменная `payCode` будет изменена, хотя предполагается, что она должна только проверяться!

### Типичная ошибка программирования 2.22

Использование операции `==` для присваивания или операции `=` для проверки равенства.

Обычно программисты записывают подобные условия как `x == 7`, т.е. помещают имя переменной слева, а константу — справа. Если переставить их так, чтобы константа была слева, а имя переменной справа, т.е. записать

`7 == x`, то программист, по ошибке заменивший операцию `==` операцией `=`, будет защищен компилятором. Компилятор воспримет такую ошибочную подмену как синтаксическую ошибку, так как слева в операторе присваивания может стоять только имя переменной. Это во всяком случае защитит от крайне неприятной логической ошибки во время выполнения.

Имена переменных относятся к так называемым *L-величинам* (*lvalue* — левая величина, левое значение), которые могут использоваться слева от операции присваивания. Константы относятся к так называемым *R-величинам* (*rvalue* — правая величина, правое значение), которые могут использоваться только справа от операции присваивания. Отметим, что L-величины могут использоваться как R-величины, но не наоборот.

### **Хороший стиль программирования 2.29**

Если выражение проверки равенства содержит переменную и константу, например, `x == 1`, некоторые программисты предпочитают записывать подобные выражения, помещая константу слева, а имя переменной справа, чтобы предохранить себя от логической ошибки при случайной подмене операции `==` операцией `=`.

Противоположная ошибка может быть столь же неприятной. Предположим, программист хотел присвоить значение переменной простым оператором типа

`x = 1;`

но случайно написал

`x == 1;`

Это тоже не является синтаксической ошибкой. Компилятор просто расценит это как условное выражение. Если `x` равно 1, это условие истинно и выражение возвращает значение 1. Если `x` не равно 1, условие ложно и выражение возвращает значение 0. Независимо от того, какое значение возвращается, операция присваивания отсутствует, так что это значение просто потерянется и значение `x` останется неизменным, что, вероятно, вызовет логическую ошибку выполнения. К сожалению, мы не знаем простого приема, который мог бы помочь вам в решении этой проблемы!

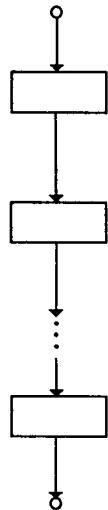
## **2.21. Заключение по структурному программированию**

Архитектор проектирует здания, используя совокупную мудрость и опыт, накопленные в его профессии. Так же должен поступать и программист. Наша область деятельности моложе, чем архитектура, и наш коллективный опыт существенно более ограничен. Мы усвоили, что структурное программирование позволяет создавать программы более простые для понимания, чем неструктурированные, более простые для проверки, отладки, модификации и даже более корректные в математическом смысле.

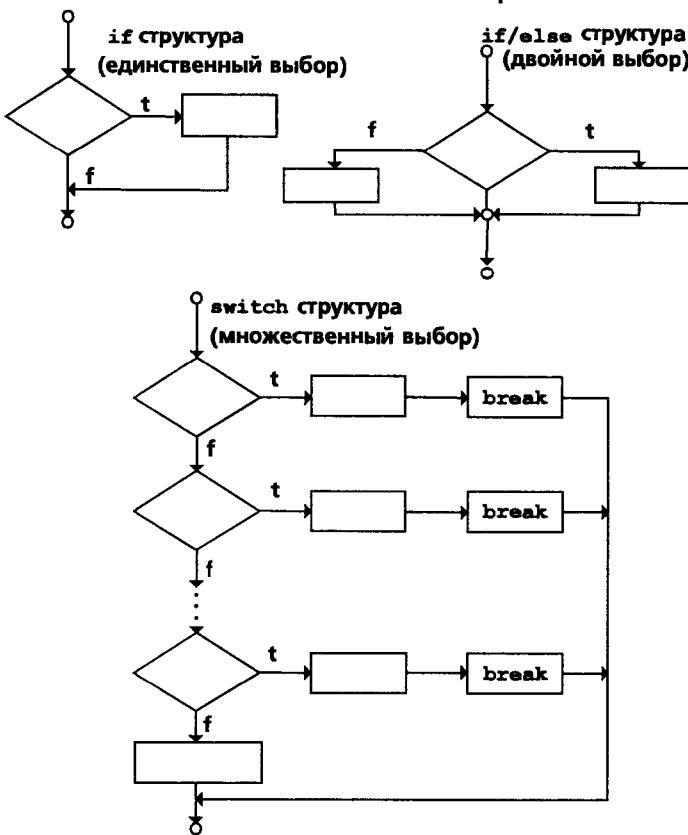
Рис. 2.32 суммирует сведения по управляющим структурам в C++. Малые окружности использованы, чтобы отметить точки единственного входа и единственного выхода каждой структуры. Произвольное соединение отдельных символов блок-схем может привести к неструктурированным программам. Следовательно, профессиональное программирование заключается в выборе

Рис. 2.32. Структуры следования, выбора и повторения с одним входом и одним выходом в C++

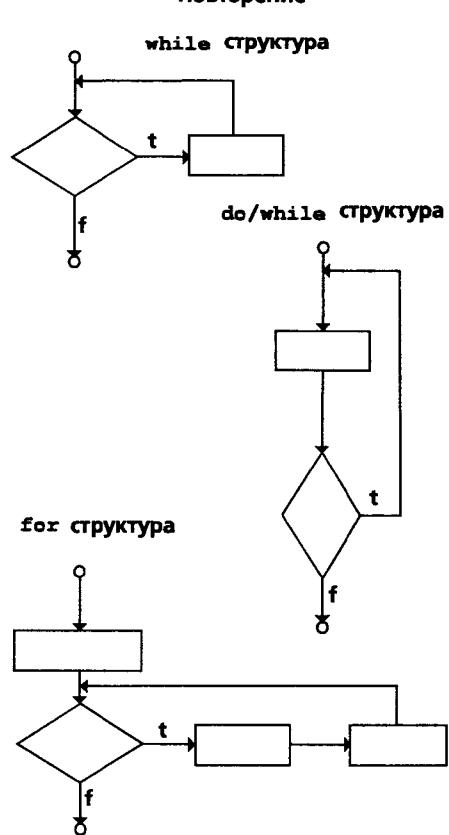
### Следование



### Выбор



### Повторение



комбинаций символов, соответствующих ограниченному множеству управляющих структур, и в построении структурированных программ соответствующим комбинированием управляющих структур двумя простыми способами. Для упрощения используются только структуры с единственным входом и единственным выходом, то есть имеющие только одну точку входа и одну точку выхода. Это упрощает формирование структурированных программ последовательным соединением управляющих структур: выход одной структуры подключается непосредственно ко входу следующей, т.е. управляющие структуры просто размещаются в программе одна за другой. Мы уже называли такой способ соединения «пакетированием управляющих структур». Правила построения структурированных программ позволяют также вкладывать структуры друг в друга.

На рис. 2.33 приведены правила формирования должным образом структурированных программ. Эти правила предполагают, что символ прямоугольника на блок-схеме может использоваться для того, чтобы указать любые действия, включая ввод и вывод информации.

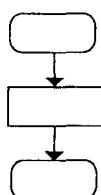
Применение правил рис. 2.33 всегда приводит к структурированной блок-схеме, построенной чисто по принципу компоновки из стандартных блоков. Например, повторное применение правила 2 к простейшей блок-схеме приводит к структурированной блок-схеме, содержащей множество последовательных прямоугольников (рис. 2.35). Отметим, что правило 2 генерирует пакет управляющих структур; это правило можно назвать *правилом пакетирования*.

Правило 3 называется *правилом вложения*. Повторное применение правила 3 к простейшей блок-схеме приводит к блок-схеме чисто вложенных управляющих структур. Например, на рис. 2.36 прямоугольник простейшей блок-схемы сначала замещается структурой двойного выбора (*if/else*). Затем правило 3 снова прикладывается к обоим прямоугольникам в структуре двойного выбора, заменяя каждый из них новой структурой двойного выбора. Пунктирная рамка вокруг каждой из этих структур двойного выбора представляет *прямоугольник*, который был замещен.

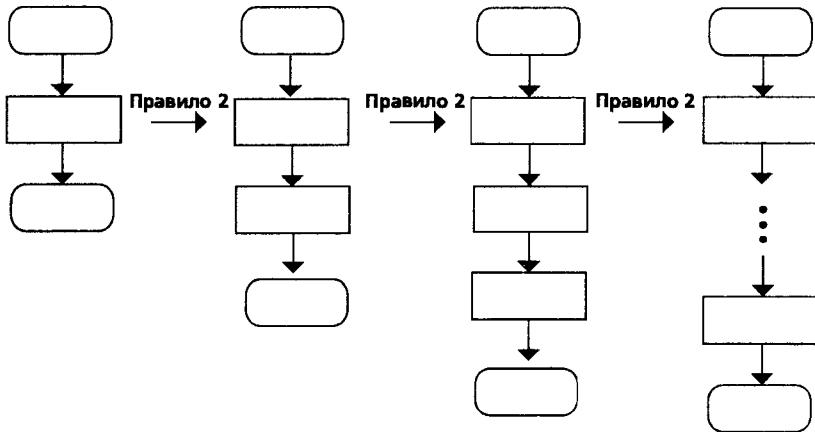
#### Правила формирования структурированных программ

- 1) Начинайте с простейшей блок-схемы (рис. 2.34).
- 2) Каждый прямоугольник (действие) может быть замещен двумя последовательными прямоугольниками (действиями)
- 3) Каждый прямоугольник (действие) может быть замещен любой управляющей структурой (следования, *if*, *if/else*, *switch*, *while*, *do/while* или *for*).
- 4) Правила 2 и 3 могут применяться неограниченно и в любой последовательности.

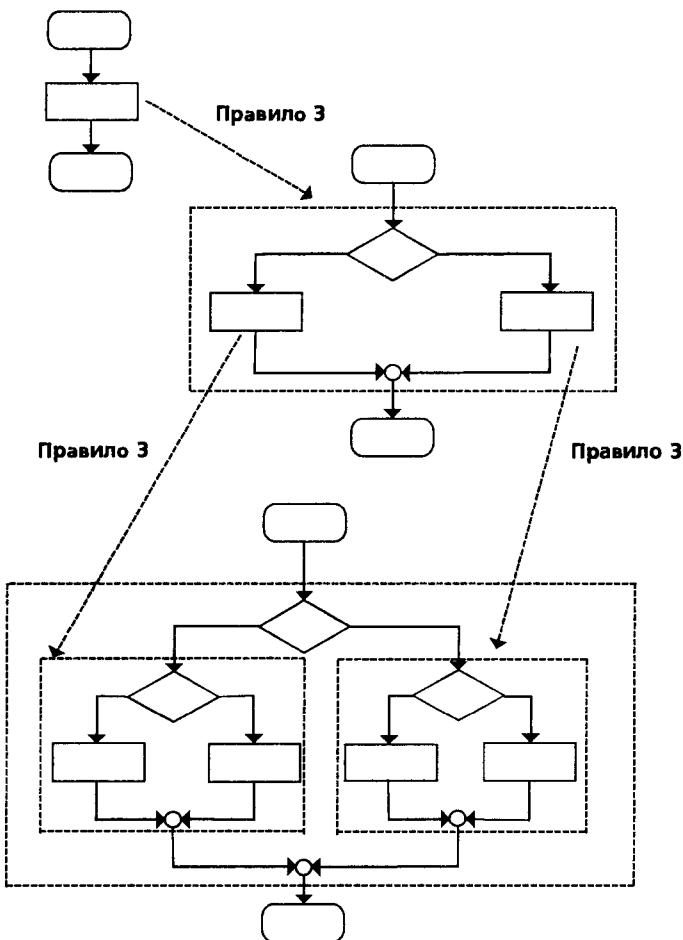
**Рис. 2.33.** Правила формирования структурированных программ



**Рис. 2.34.** Простейшая блок-схема



**Рис. 2.35.** Повторное применение правила 2 к простейшей блок-схеме



**Рис. 2.36.** Повторное применение правила 3 к простейшей блок-схеме

Правило 4 позволяет генерировать большие, сложные структуры с произвольным уровнем вложенности. Блок-схемы, возникающие в результате применения правил рис. 2.33, составляют множество всех возможных структурированных блок-схем и, следовательно, множество всех возможных структурированных программ.

Прелест структурного подхода в том, что мы используем всего семь простых фрагментов с одним входом и одним выходом и соединяем их всего двумя простыми способами. Рис. 2.37 показывает пакеты стандартных блоков, которые создаются при применении правила 2, и вложение стандартных блоков, которые создаются при применении правила 3. Этот рисунок показывает также частично перекрывающиеся стандартные блоки, которые не могут появиться в структурированных блок-схемах, поскольку игнорируется оператор *goto*.

Если следовать правилам рис. 2.33, неструктурированные блок-схемы (подобные приведенным на рис. 2.38) не могут быть созданы. Если вы сомневаетесь, является ли некоторая конкретная блок-схема структурированной, примените правила рис. 2.33 в обратной последовательности и попытайтесь свернуть блок-схему к простейшей. Если блок-схема свернется к простейшей, значит исходная блок-схема структурирована, в противном случае — нет.

Структурное программирование стимулирует простоту. Бом и Джакопини доказали, что достаточно всего трех форм управления:

- Следование
- Выбор
- Повторение

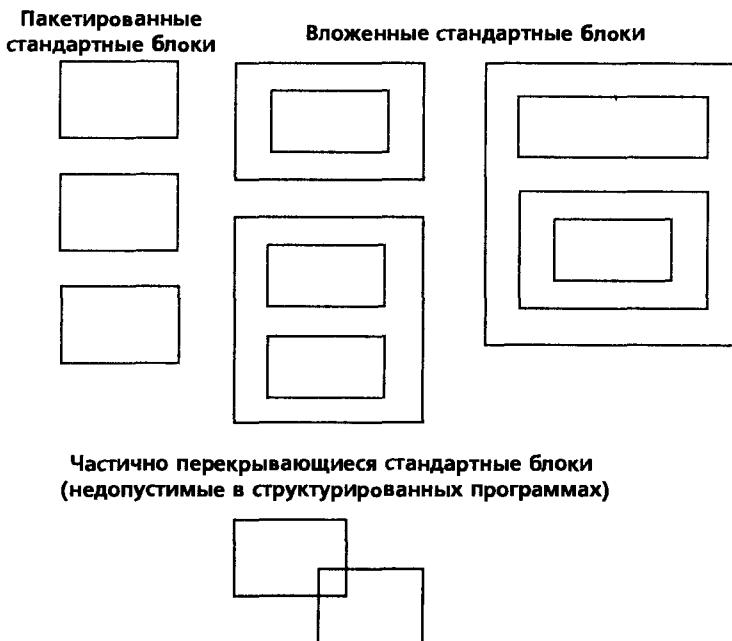


Рис. 2.37. Пакетированные, вложенные и частично перекрывающиеся стандартные блоки

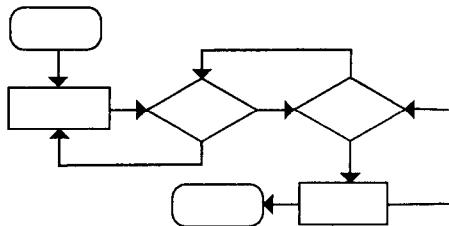


Рис. 2.38. Неструктурированная блок-схема

Следование тривиально. Выбор осуществляется одним из трех способов

- структурой **if** (единственный выбор)
- структурой **if/else** (двойной выбор)
- структурой **switch** (множественный выбор)

В действительности можно доказать, что для реализации любой формы выбора достаточно одной простой структуры **if** — все, что можно сделать структурами **if/else** и **switch**, можно представить комбинацией структур **if** (правда, может быть не так удобно).

Повторение обеспечивается одним из трех способов:

- структурой **while**
- структурой **do/while**
- структурой **for**

В действительности можно доказать, что для реализации любой формы повторения достаточно одной структуры **while**. Все, что можно сделать структурами **do/while** и **for**, можно представить структурой **while** (правда, может быть не так удобно).

В результате можно утверждать, что любую форму управления, которая может потребоваться в программе на C++, можно выразить в терминах

- следование
- структура **if** (выбор)
- структура **while** (повторение)

И эти управляющие структуры могут комбинироваться только двумя путями: пакетированием или вложением. Действительно, структурное программирование способствует простоте!

В данной главе мы рассмотрели, как составлять программы из управляющих структур, содержащих только действия и выбор. В главе 3 мы познакомимся с другой структурной единицей программ, называемой *функцией*. Мы научимся составлять большие программы, комбинируя функции, которые, в свою очередь, состоят из управляющих структур. Мы увидим также, как применение функций способствует повторному использованию программных средств. В главе 6 мы ознакомимся еще с одной структурной единицей программ, называемой *классом*. В дальнейшем мы будем создавать объекты из классов и возобновим наше исследование объектно-ориентированного программирования. А пока мы продолжим знакомство с объектами, представив на рассмотрение проблему, которую читатель будет решать технологией объектно-ориентированного проектирования.

## 2.22. Размышления об объектах: идентификация объектов задачи

Настало время поговорить об этой удивительной технологии, называемой объектной ориентацией. Специальные разделы в конце этой и нескольких следующих глав призваны облегчить вам проникновение в объектную ориентацию путем решения интересной и сложной задачи, взятой из реальной жизни — построение программы, моделирующей лифт.

В главах 2–5 вы пройдете различные этапы объектно-ориентированного проектирования (OOD). Начиная с главы 6 вы осуществите программу, моделирующую лифт, используя технологию объектно-ориентированного программирования (OOP) на C++. Пока такое задание может показаться вам несколько сложным. Не беспокойтесь. В данной главе мы рассмотрим только малую часть этой проблемы.

### Постановка задачи

Некоторая компания намерена построить двухэтажный офис и оборудовать его лифтом по последнему слову техники. Компания предлагает вам разработать объектно-ориентированное программное обеспечение для моделирования работы этого лифта, чтобы определить, удовлетворяет ли он своему назначению.

Лифт, предназначенный для одного человека, должен быть спроектирован так, чтобы сберегать энергию, так что он должен перемещаться только когда это необходимо. Он начинает день ожиданием с закрытыми дверьми на первом этаже здания. Лифт, конечно, может изменять направление движения — сначала вверх, потом вниз.

Ваша моделирующая программа должна включать часы, которые в начале дня устанавливаются на 0, а затем «отсчитывают» время с тактами в одну секунду. Компонент «планировщик» программы случайным образом планирует появление первого пассажира на любом этаже (в главе 3 вы поймете, как планировать случайное появление). Когда время на часах совпадет со временем первого появления пассажира, моделирующая программа «создаст» нового пассажира и поместит его на указанном этаже. Затем этот пассажир нажмет кнопку «вверх» или «вниз». Этаж, требуемый пассажиру, никогда не равен тому, на котором он появился.

Если первый пассажир появился в этот день на первом этаже, он немедленно войдет в лифт (конечно, после того, как нажмет кнопку «вверх» и подождет, пока откроются двери лифта). Если первый пассажир появился на втором этаже, то лифт отправится на этот этаж чтобы забрать пассажира. Лифту требуется пять тактов на перемещение между этажами.

Лифт сигнализирует о своем появлении на этаже включением световой панели над дверью на данном этаже и звуковым сигналом. Кнопки на этаже и кнопки этого этажа в лифте возвращаются в исходное состояние. Лифт открывает дверь. Если в лифте есть пассажир, цель которого — данный этаж, то он выходит из лифта. Другой пассажир, если он ожидает у дверей, входит в лифт и нажимает кнопку нужного ему этажа. Лифт закрывает двери. Если лифт должен двигаться, то он определяет направление движения (несложная задача для лифта на 2 этажа!) и начинает перемещаться к следующему этажу. Для упрощения предположим, что все эти события происходят мгновенно и интервал времени с того момента, как лифт достиг нужного этажа, до момента

закрывания дверей равен нулю. Лифт всегда знает, на каком он этаже в данный момент и на какой этаж он должен переместиться.

Не более одного человека может ожидать лифт на любом этаже в любой момент времени, так что если этаж занят, когда на нем должен появиться новый пассажир (не тот, который уже вызвал лифт), то его появление перепланируется на одну секунду позже. Для упрощения примем, что в моделирующей программе в любой момент времени может «создаваться» только один человек, так что время появления очередного человека в здании никогда не повторяется. Предположим, что человек случайно появляется на каждом этаже каждые 5–10 секунд — в главе 3 мы изучим, как использовать генерацию случайных чисел для такого моделирования.

Ваша цель в рамках этих специальных разделов глав 2–8 — создать работающую программу моделирования, функционирующую в соответствии с описанной постановкой задачи. Ваша программа должна промоделировать несколько минут работы лифта и определить, удовлетворяет ли лифт ожидаемым требованиям к перевозкам в этом офисном здании.

### Лабораторное задание 1 по лифту

В этом и нескольких следующих заданиях вы должны осуществить отдельные этапы объектно-ориентированного проектирования. Первый этап — *выявление (идентификация) объектов* в вашей задаче. В конечном итоге вы должны будете дать формальное описание этих объектов и представить его на С++. В данном задании вы должны:

1. Выявить объекты задачи моделирования лифта. В постановке задачи указано много совместно функционирующих объектов при моделировании лифта и его взаимодействия с различными людьми, этажами здания, кнопками и т.д. Выделите *имена существительные* в постановке задачи; с большой вероятностью они представляют большинство объектов, необходимых для создания программы, моделирующей лифт.
2. Для каждого идентифицированного вами объекта запишите один точно сформулированный абзац, который бы охватил все факты в постановке задачи, относящиеся к данному объекту.

### Замечания

1. Это неплохое задание для группы. В идеале вы должны бы работать в группе из двух — четырех человек. Это поможет вам и вашим товарищам по группе подкрепить усилия друг друга, обсудить и усовершенствовать каждый проект и подход к его реализации.
2. Ваша группа должна состязаться с другой группой вашего класса в разработке «наилучшего» проекта и его реализации.
3. Вы научитесь реализовывать «случайности» в следующей главе, когда мы изучим генерацию случайных чисел. Генерация случайных чисел поможет вам осуществить моделирование таких процессов, как случайное бросание монетки или игральных костей. Это также поможет вам реализовать случайное появление пассажиров лифта.
4. Мы сделали ряд упрощающих предположений. Вы можете решиться возместить некоторые утраченные при этом дополнительные детали работы лифта.

5. Поскольку реальный мир также объектно ориентирован, для вас совершенно естественно заниматься этим проектом, хотя формально вы еще не изучали объектную ориентацию.
6. Не беспокойтесь по поводу совершенства. Проектирование системы не является каким-то полностью завершенным идеальным процессом, так что вы должны смотреть на ваш проект только как на возможно лучшее приближение.

## Вопросы

1. Какое решение вы примете, если лифт окажется в состоянии осуществить запланированный объем перевозок?
2. Почему намного сложнее сделать проект для здания с тремя и более этажами.
3. В дальнейшем мы увидим, что когда мы создадим один объект лифта, нам будет легко создать их сколько угодно. Какие проблемы вы предвидите в наличии нескольких лифтов, которые могут принимать и выгружать пассажиров на каждом этаже здания?
4. Для упрощения мы предположили, что емкость нашего лифта и каждого этажа — только один человек. Какие проблемы вы предвидите, если увеличить эти емкости?

## Резюме

- Процедура решения задачи в терминах операций, которые должны выполняться, и последовательности, в которой эти операции должны выполняться, называется алгоритмом.
- Определение последовательности, в которой должны выполняться операторы в компьютерной программе, называется программным управлением.
- Псевдокод помогает программисту «обдумать программу» прежде, чем попытаться написать ее на таком языке программирования, как C++.
- Объявления — это сообщения компилятору, которые определяют имена и атрибуты переменных и дают указание компилятору зарезервировать для них место в памяти.
- Структуры выбора используются для поиска среди альтернативных способов действий.
- Структура выбора **if** осуществляет указанное действие только, если ее условие истинно.
- Структура выбора **if/else** определяет различные действия, выполняемые, если условие истинно, и если оно ложно.
- Везде, где нормально ожидается один оператор, а должна выполняться группа операторов, эти операторы должны заключаться в фигурные скобки, образуя составной оператор. Составной оператор может быть помещен в любом месте программы, в котором может размещаться единичный оператор.

- Пустой оператор, показывающий, что не требуется выполнять никаких действий, обозначается записью точки с запятой там, где должен был бы быть оператор.
- Структура повторения обозначает, что некоторое действие должно повторяться до тех пор, пока некоторое условие остается истинным.
- Формат структуры повторения **while** :

```
while (условие)
 оператор
```

- Величины, имеющие дробную часть, называются числами с плавающей запятой и представляются типом данных **float**.
- Унарная операция приведения к типу (**float**) создает временную копию с плавающей запятой своего операнда.
- C++ имеет операции арифметического присваивания **+=**, **-=**, **\*=**, **/=** и **%=**, которые помогают сократить запись выражений определенного типа.
- C++ имеет операции инкремента **(++)** и декремента **(--)**, которые обеспечивают приращение или уменьшение переменной 1. Если эти операции записаны в префиксной форме (перед переменной), то сначала производится увеличение или уменьшение переменной на 1, а затем измененная переменная используется в выражении. Если эти операции записаны в постфиксной форме (после переменной), то переменная сначала используется в выражении, а затем увеличивается или уменьшается на 1.
- Цикл — это группа операторов, которая выполняется повторно до тех пор, пока удовлетворяется некоторое условие. Имеется две формы повторения: повторение, управляемое счетчиком, и повторение, управляемое меткой.
- Счетчик цикла используется, чтобы задавать число повторений цикла. Он обычно увеличивается (или уменьшается) на 1 каждый раз при повторении тела цикла.
- Значение метки в общем случае используется, чтобы управлять повторением, когда число повторений заранее не известно и цикл включает в себя оператор, осуществляющий ввод данных в каждом цикле. Значение метки вводится после того, как все необходимые для программы данные уже введены. Метка должна отличаться от всех возможных значений вводимых данных.
- Структура повторения **for** содержит все компоненты, необходимые для повторения, управляемого счетчиком. Общий формат структуры **for**:

```
for (выражение1; выражение2; выражение3)
 оператор
```

где **выражение1** задает начальное значение переменной, управляющей циклом, **выражение2** является условием продолжения цикла, а **выражение3** задает приращение управляющей переменной.

- Структура повторения **do/while** проверяет условие продолжения в конце цикла, так что тело цикла будет выполнено по крайней мере один раз. Формат структуры **do/while**:

```

do
 оператор
 while (условие);

```

- Когда оператор **break** выполняется в одной из структур повторения (**while**, **for** и **do/while**), происходит немедленный выход из структуры.
- Оператор **continue** в одной из структур повторения (**while**, **for** и **do/while**), вызывает пропуск оставшейся части тела структуры и начинается выполнение следующей итерации цикла.
- Оператор **switch** производит множественный выбор, при котором значение некоторой переменной или выражения проверяется на множестве допустимых значений и в зависимости от результатов проверки предпринимаются различные действия. В большинстве программ после операторов, соответствующих каждой метке **case**, надо применять оператор **break**. Несколько **case** могут соответствовать одним и тем же операторам; тогда перед этими операторами располагается список меток **case**. Структура **switch** может осуществлять проверку на совпадение только с целыми постоянными выражениями.
- В системе UNIX и многих других признак конца файла вводится комбинацией  
`<ctrl-d>`  
в текущей строке. В VMS и DOS признак конца файла вводится нажатием  
`<ctrl-z>`
- Для формирования сложных условий путем комбинирования простых условий могут использоваться логические операции. Логическими операциями являются **&&**, **||** и **!**, означающие логическое И, логическое ИЛИ и логическое отрицание соответственно.
- Истинным значением **true** является любое ненулевое значение; ложным значением **false** является 0.

## Терминология

|                                      |                                        |
|--------------------------------------|----------------------------------------|
| <b>break</b>                         | действие                               |
| <b>char</b>                          | задание начального значения            |
| <b>continue</b>                      | значение метки                         |
| <b>double</b>                        | исправимая ошибка                      |
| <b>EOF</b>                           | ключевое слово                         |
| <b>float</b>                         | логическая операция                    |
| <b>ios::fixed</b>                    | логическая ошибка                      |
| <b>ios::left</b>                     | логическое И (&&)                      |
| <b>ios::showpoint</b>                | логическое ИЛИ (  )                    |
| L-величина (lvalue, левое значение)  | логическое отрицание (!)               |
| <b>long</b>                          | манипулятор потока <b>setiosflags</b>  |
| R-величина (rvalue, правое значение) | манипулятор потока <b>setprecision</b> |
| <b>short</b>                         | манипулятор потока <b>setw</b>         |
| алгоритм                             | метка <b>case</b>                      |
| бесконечный цикл (зацикливание)      | множественный выбор                    |
| блок                                 | множество символов ASCII               |
| вложенные управляющие структуры      | модель действие-решение                |
| выбор                                | неисправимая ошибка                    |

нисходящая пошаговая детализация  
 операции арифметического  
 присваивания `+=`, `==`, `*=`, `/=` и `%=`  
 операция `!`  
 операция `&&`  
 операция `++`  
 операция `--`  
 операция `?:`  
 операция `||`  
 операция декремента `(--)`  
 операция инкремента `(++)`  
 операция приведения к типу  
 ошибка занижения (или завышения)  
 на единицу  
 параметризованный манипулятор  
 потока  
 передача управления  
 повторение  
 повторение заданное число раз  
 повторение неопределенное число раз  
 повторение, управляемое счетчиком  
 последовательное выполнение  
 постфиксная форма операции  
 декремента  
 постфиксная форма операции  
 инкремента  
 префиксная форма операции  
 декремента  
 префиксная форма операции  
 инкремента  
 псевдокод  
 пустой оператор `(;)`

раздел `default` в `switch`  
 решение  
 символы-разделители  
 синтаксическая ошибка  
 составной оператор  
 структура выбора `if`  
 структура выбора `if/else`  
 структура выбора `switch`  
 структура двойного выбора  
 структура множественного выбора  
 структура повторения  
 структура повторения `for`  
 структура повторения `while`  
 структура с единственным выбором  
 структурное программирование  
 счетчик цикла  
 тело цикла  
 трехчленная (тернарная) операция  
 унарная операция  
 управляющая структура  
 управляющие структуры с одним  
 входом и одним выходом  
 условие продолжения цикла  
 условная операция `?:`  
 формат с фиксированной точкой  
 функция `cin.get()`  
 функция `pow`  
 целочисленное деление  
 цикл  
 ширина поля

## Типичные ошибки программирования

- 2.1. Использование ключевого слова в качестве идентификатора.
- 2.2. Пропуск одной или обеих фигурных скобок, ограничивающих составной оператор.
- 2.3. Запись точки с запятой после условия в структуре `if` приводит к логической ошибке в структуре с единственным выбором и к синтаксической ошибке в структуре с двойным выбором (если часть `if` в действительности содержит оператор).
- 2.4. В теле структуры `while` не предусматривается действие, которое приведет к тому, что со временем условие `while` станет ложным. Выполнение подобной структуры повторения никогда не прервется — такая ошибка называется «зацикливание».
- 2.5. Запись ключевого слова `while` как `While` с символом `W` в верхнем регистре (помните, что язык C++ чувствителен к регистру). Все зарезервированные ключевые слова C++, такие, как `while`, `if` и `else`, содержат только символы нижнего регистра.

- 2.6. Если для счетчика или переменной суммы не задады начальные значения, то результат работы вашей программы будет, вероятно, неправильным. Это пример логической ошибки.
- 2.7. Выбор такого значения метки, которое могут принимать и входные данные.
- 2.8. Попытка деления на нуль вызывает сбой программы.
- 2.9. Использование чисел с плавающей запятой в предположении, что они совершенно точные, может приводить к некорректным результатам. Числа с плавающей запятой на большинстве компьютеров являются приближенными.
- 2.10. Попытка использовать в операции инкремента или декремента операнд, отличный от имени простой переменной, например, выражение  $++(x+1)$  является синтаксической ошибкой.
- 2.11. Поскольку числа с плавающей запятой являются приближенными, контроль количества выполнений цикла с помощью переменной с плавающей запятой может приводить к неточному значению счетчика и неправильному результату проверки условия окончания.
- 2.12. Использование неправильной операции отношения или использование неправильной конечной величины счетчика цикла в условиях структур **while** или **for** может приводить к ошибке занижения (или завышения) на единицу.
- 2.13. Использование запятых вместо точек с запятой в заголовке структуры **for**.
- 2.14. Размещение точки с запятой сразу после правой закрывающей скобки заголовка **for** делает тело структуры пустым оператором. Обычно это логическая ошибка.
- 2.15. Использование несоответствующей операции отношения в условии продолжения цикла при счете циклов сверху вниз (например, использование  $i \leq 1$  при счете циклов сверху до 1).
- 2.16. Забывают включить файл **math.h** в программы, использующие библиотеку математических функций.
- 2.17. Забывают вставить оператор **break**, когда он нужен в структуре **switch**.
- 2.18. Пропуск пробела между ключевым словом **case** и целым значением, которое проверяется в структуре **switch**, может вызвать логическую ошибку. Например, запись **case3:** вместо **case 3:** просто создаст неиспользуемую метку (мы поговорим об этом подробнее в главе 18). Дело в том, что в этой структуре **switch** не будут совершены соответствующие действия, когда управляющее выражение **switch** будет иметь значение 3.
- 2.19. Отсутствие обработки новой строки при вводе, если символы читаются по одному, может привести к логической ошибке.
- 2.20. Если условие продолжения цикла в структурах **while**, **for** или **do/while** никогда не становится ложным, то возникает зациклива-

ние. Чтобы предотвратить это, убедитесь, что нет точки с запятой сразу после заголовка структуры `while`. В цикле, управляемом счетчиком, убедитесь, что управляющая переменная увеличивается (или уменьшается) в теле цикла. В цикле, управляемом меткой, убедитесь, что значение метки в конце концов будет введено.

- 2.21. В выражениях, использующих операцию `&&`, может оказаться, что одно условие — назовем его зависимым — может требовать, чтобы другое условие было значимо при оценке зависимого условия. В этом случае зависимое условие должно быть помещено после другого условия, в противном случае может произойти ошибка.
- 2.22. Использование операции `==` для присваивания или операции `=` для проверки равенства.

### Хороший стиль программирования

- 2.1. Неукоснительно соблюдайте правила ступенчатой записи во всех ваших программах, это существенно улучшит их читаемость. Мы советуем делать отступы фиксированного размера примерно 0,5 см или три пробела на отступ.
- 2.2. Псевдокод часто используется при «обдумывании» программ в процессе их разработки. Затем программа на псевдокоде преобразуется в программу на C++.
- 2.3. Записывайте с отступом оба предложения структуры ЕСЛИ-ИНАЧЕ (`if/else`).
- 2.4. Если есть несколько уровней отступов, каждый уровень должен иметь постоянное число пробелов.
- 2.5. Некоторые программисты предпочитают сначала записать открывающую и закрывающую скобки составного оператора, а уже потом писать внутри их требуемые операторы. Это позволяет избежать пропуска одной или обеих скобок.
- 2.6. Задавайте начальные значения всем счетчикам и переменным сумм.
- 2.7. Когда осуществляется деление на выражение, значение которого может равняться нулю, надо предварительно проверить эту возможность и обработать ее должным образом (например, напечатать сообщение об ошибке), не допуская возникновения неисправимой ошибки.
- 2.8. В цикле, управляемом меткой, приглашение к вводу данных должно явно напоминать пользователю, какое значение используется как метка.
- 2.9. Не следует сравнивать числа с плавающей запятой на их равенство или неравенство друг другу. Лучше проверять, не меньше ли их разность некоторой заданной малой величины.
- 2.10. Задание начальных значений переменных одновременно с их объявлением помогает программисту избежать проблем, связанных с неопределенными значениями данных.

- 2.11. Унарные операции должны размещаться после их operandов без разделяющего пробела.
- 2.12. Управляйте количеством повторений цикла с помощью целой переменной.
- 2.13. Записывайте с отступом операторы тела каждой управляющей структуры.
- 2.14. Размещайте пустую строку до и после каждой большой управляющей структуры, чтобы она выделялась в программе.
- 2.15. Слишком большая глубина вложенности может сделать программу трудной для понимания. Как правило, старайтесь избегать более трех уровней отступов.
- 2.16. Вертикальные пробелы над и под управляющими структурами вместе с отступами в тела этих структур по отношению к их заголовкам создают двумерный эффект, облегчающие чтение программы.
- 2.17. Использование конечной величины управляющей переменной в условиях структур `while` и `for` и использование операции отношения `<=` поможет избежать ошибок занижения на единицу. Например, для цикла, используемого при печати чисел от 1 до 10 условие продолжения цикла надо записать `counter <= 10`, а не `counter < 10` (что является ошибкой занижения на единицу) или `counter < 11` (что тем не менее корректно).
- 2.18. Помещайте в разделы задания начального значения и изменения переменных структуры `for` только выражения, относящиеся к управляющей переменной. Манипуляции с другими переменными должны размещаться или до цикла (если они выполняются только один раз подобно операторам задания начальных значений), или внутри тела цикла (если они должны выполняться в каждом цикле, как, например, операторы инкремента или декремента).
- 2.19. Хотя управляющая переменная может изменяться в теле цикла `for`, избегайте делать это, так как такая практика приводит к неявным, неочевидным ошибкам.
- 2.20. Хотя операторы, предшествующие `for` и операторы тела `for` могут часто включаться в заголовок `for`, избегайте делать это, чтобы не затруднять чтение программы.
- 2.21. Ограничивайте, если возможно, размер заголовка управляющей структуры так, чтобы он умешался на одной строке.
- 2.22. Не используйте переменные типов `float` и `double` для денежных расчетов. Неточность чисел с плавающей запятой может привести к ошибкам, которые проявятся в итоге в неправильной сумме денег. В качестве упражнения попробуйте использовать для денежных расчетов целые числа. Заметим: библиотека классов C++ позволяет должным образом осуществлять денежные расчеты.
- 2.23. Вставляйте метку `default` в оператор `switch`. Случаи неудачных проверок в операторе `switch` без метки `default` будут игнорироваться. Включение метки `default` фиксирует внимание программиста на не-

обходимости обрабатывать исключительную ситуацию. Но бывают ситуации, в которых никакой обработки по метке `default` не требуется.

- 2.24. Хотя предложения `case` и `default` могут размещаться в структуре `switch` в произвольном порядке, стоит учесть практику качественного программирования — помещать `default` в конце.
- 2.25. Если в структуре `switch` предложение `default` помещено последним в списке, то оператор `break` в нем не требуется. Но некоторые программисты включают `break` и тут для четкости и для симметрии с другими случаями.
- 2.26. Не забывайте обеспечить обработку возможности появления во входном потоке символа перехода на новую строку и других символов-разделителей, если обрабатываете по одному символу за раз.
- 2.27. Некоторые программисты всегда включают фигурные скобки в структуру `do/while`, даже если в них нет необходимости. Это помогает устранить двусмысленность, пристекающую из совпадения предложений структуры `while` и структуры `do/while`, содержащей один оператор.
- 2.28. Некоторые программисты чувствуют, что `break` и `continue` нарушают структурное программирование. А поскольку результат работы этих операторов, как мы скоро увидим, может быть достигнут технологией структурного программирования, то эти программисты не применяют `break` и `continue`.
- 2.29. Если выражение проверки равенства содержит переменную и константу, например, `x == 1`, некоторые программисты предпочитают записывать подобные выражения, помещая константу слева, а имя переменной справа, чтобы предохранить себя от логической ошибки при случайной подмене операции `==` операцией `=`.

### Советы по повышению эффективности

- 2.1. Программисты могут программы писать несколько быстрее и компилятор может компилировать их немного быстрее, если использовать «сокращенные» операции присваивания (составные присваивания). Некоторые компиляторы генерируют код, который выполняется быстрее при использовании «сокращенных» операций присваивания.
- 2.2. Многие из советов по повышению эффективности, которые мы будем давать в этой книге, приводят к незначительному улучшению, так что читатель может иметь искушение пренебречь ими. Значительное повышение эффективности получается, если такие небольшие улучшения находятся внутри цикла и могут повторяться много раз.
- 2.3. Избегайте размещать выражения, значение которых не изменяется, внутри цикла. Но даже, если вы сделаете это, то многие современные сложные оптимизирующие компиляторы автоматически разместят подобные выражения вне цикла во время генерации машинного кода.

- 2.4. В ситуациях, где важна эффективность, где жесткие требования к памяти или критична скорость исполнения программы, может оказаться желательным использовать целые минимальных размеров.
- 2.5. При надлежащем использовании операторы `break` и `continue` выполняются быстрее, чем соответствующие приемы структурного программирования, которые мы скоро изучим.
- 2.6. В выражениях, использующих операцию `&&`, если отдельные условия независимы друг от друга, записывайте комбинированное условие так, чтобы самым левым было то простое условие, которое вероятнее всего окажется ложным. В выражениях, использующих операцию `||`, записывайте комбинированное условие так, чтобы самым левым было то простое условие, которое вероятнее всего окажется истинным. Это может сократить время выполнения программы.

### Замечания по мобильности

- 2.1. Комбинация клавиш для ввода признака конца файла зависит от системы.
- 2.2. Проверка на символьическую константу `EOF`, а не на `-1` делает программу более мобильной. Стандарт ANSI устанавливает, что `EOF` имеет целое отрицательное значение (но не обязательно `-1`). Так что `EOF` может иметь различные значения в разных системах.
- 2.3. Поскольку размер типа `int` варьируется от системы к системе, используйте тип `long`, если вы предусматриваете обработку целых, значения которых могут лежать вне диапазона `32767`, и вы, вероятнее всего, сможете выполнять свою программу на нескольких различных компьютерных системах.

### Замечания по технике программирования

- 2.1. Составной оператор может быть помещен в любом месте программы, в котором может размещаться единичный оператор.
- 2.2. Справедливо, что составной оператор может быть помещен в любом месте программы, в котором может размещаться единичный оператор, но также справедливо и то, что можно вообще обойтись без оператора, т.е. поместить пустой оператор. Для этого надо поместить символ точки с запятой (`;`) в том месте, где нормально должен находиться оператор.
- 2.3. Каждая детализация, так же как и сама вершина, является полным описанием алгоритма; меняется только уровень детализации.
- 2.4. Многие программы могут быть логически разделены на три этапа: этап задания начальных значений, в котором задаются начальные значения переменных программы; этап обработки данных, в котором вводятся данные и устанавливаются значения соответствующих переменных программы; заключительный этап, в котором вычисляются и печатаются окончательные результаты.

- 2.5. Программист завершает процесс исходящей разработки с пошаговой детализацией, когда алгоритм на псевдокоде настолько детализирован, чтобы его псевдокод можно было преобразовать в программу на C++. Реализованная программа на C++ окажется в этом случае простой и наглядной.
- 2.6. Опыт показывает, что наиболее трудной частью решения задач на компьютерах является разработка алгоритма решения. После того, как корректный алгоритм получен, процесс создания на его основе работающей программы на C++ продвигается успешно.
- 2.7. Многие опытные программисты пишут программы, не используя такой инструмент разработки, как псевдокод. Эти программисты полагают, что их конечная цель — решение задачи на компьютере и что написание псевдокода только задержит достижение конечного результата. Это может быть иногда оправдано для простых и хорошо знакомых задач, но может приводить к серьезным ошибкам в больших и сложных проектах.
- 2.8. Имеется некоторое противоречие между стремлением к технике высококачественного программирования и стремлением к наилучшей эффективности программного обеспечения. Часто достижение одной из этих целей достигается за счет другой.

### Упражнения для самопроверки

Упражнения с 2.1 по 2.10 соответствуют разделам 2.1–2.12.

Упражнения с 2.11 по 2.13 соответствуют разделам 2.13–2.21.

- 2.1. Заполнить пробелы в следующих утверждениях:
- Все программы можно писать в терминах трех типов управляющих структур: \_\_\_\_\_, \_\_\_\_\_ и \_\_\_\_\_.
  - Структура выбора \_\_\_\_\_ используется для выполнения одного действия, если условие истинно, и другого, если условие ложно.
  - Повторение ряда операторов заданное число раз называется \_\_\_\_\_ повторением.
  - Когда заранее не известно, сколько раз должна быть повторена группа операторов, можно использовать \_\_\_\_\_ для окончания повторения.
- 2.2. Напишите четыре различных оператора C++, которые прибавляли бы 1 к целой переменной `x`.
- 2.3. Напишите операторы C++, выполняющие следующие действия:
- Присваивание суммы `x` и `y` переменной `z` и увеличение значения `x` на 1 после этого вычисления.
  - Проверку, больше ли значение переменной `count` числа 10. Если больше, то печать текста: «`count` больше, чем 10».
  - Уменьшение переменной `x` на 1 и затем ее вычитание из переменной `total`.
  - Вычисление остатка от деления `q` на `divisor` и присваивание результата переменной `q`. Запишите два различных варианта такого оператора.

- 2.4. Напишите операторы C++, решающие следующие задачи:
- Объявление переменных `sum` и `x` типа `int`.
  - Задание начального значения 1 переменной `x`.
  - Задание начального значения 0 переменной `sum`.
  - Сложение переменных `x` и `sum` и присваивание результата переменной `sum`.
  - Печать «Сумма равна » и затем значение переменной `sum`.
- 2.5. Объедините операторы, которые вы написали в упражнении 2.4, в программу, которая вычисляла бы и печатала сумму целых чисел от 1 до 10. Используйте структуру `while` для организации цикла, включающего операторы вычисления и приращения. Цикл должен завершаться, когда значение `x` становится равным 11.
- 2.6. Определите значения каждой переменной после осуществления вычислений. Предполагайте, что когда каждый оператор начинает выполняться, все переменные имеют целое значение 5.
- `product *= x++;`
  - `quotient /= ++x;`
- 2.7. Напишите по одному оператору C++, выполняющему следующие действия:
- Ввод целой переменной `x` с помощью `cin` и `>>`.
  - Ввод целой переменной `y` с помощью `cin` и `>>`.
  - Задание начального значения 1 целой переменной `i`.
  - Задание начального значения 1 целой переменной `power`.
  - Умножение переменной `power` на `x` и присваивание полученного результата переменной `power`.
  - Увеличение переменной `y` на 1.
  - Проверка, меньше или равно значение `y` величины `x`.
  - Вывод целой переменной `power` с помощью `cout` и `<<`.
- 2.8. Используя операторы упражнения 2.7 напишите программу на C++, которая рассчитывала бы `x` в степени `y`. Программа должна включать управляющую структуру повторения `while`.
- 2.9. Найдите и исправьте ошибки в следующих операторах:
- `while (c <= 5) {  
 product *= c;  
 ++c;`
  - `cin << value;`
  - `if (gender == 1)  
 cout << "Женщина" << endl;  
else;  
 cout << "Мужчина" << endl;`
- 2.10. Чем ошибочна следующая структура повторения `while`:
- ```
while (z >= 0)
    sum += z;
```

- 2.11.** Установите, верны или ошибочны приведенные ниже утверждения. Если ошибочны, то объясните почему.
- В структуре выбора `switch` должна быть метка `default`.
 - В структуре выбора `switch` в разделе `default` требуется оператор `break`.
 - Выражение (`x > y && a < b`) истинно, если `x > y` или `a < b`.
 - Выражение, содержащее операцию `||`, истинно, если истинны оба операнда этой операции.
- 2.12.** Напишите оператор C++ или несколько операторов C++, которые выполняли бы каждое из следующих действий:
- Суммирование нечетных целых чисел от 1 до 99 с помощью структуры `for`. Предполагайте, что объявлены целые переменные `sum` и `count`.
 - Печать величины 333.546372 в поле шириной 15 символов с точностью 1, 2 и 3 разряда после десятичной точки. Печать всех чисел в одной строке с левым выравниванием каждого числа в своем поле. Какие три значения будут напечатаны?
 - Расчет 2.5 в степени 3 с использованием функции `pow`. Печать результата с точностью 2 разряда после десятичной точки в поле шириной 10 символов. Что будет напечатано?
 - Печать целых чисел с 1 по 20 с использованием цикла `while` и счетчика `x`. Предполагайте, что переменная `x` объявлена, но ее начальное значение не задано. Печатайте только 5 целых чисел в строке. Подсказка: используйте `x % 5`. Если результат равен 0, печатайте признак перехода на новую строку, в противном случае печатайте символ табуляции.
 - Повторите упражнение 2.3 (d) используя структуру `for`.
- 2.13.** Найдите ошибку в каждом из следующих фрагментов программы и объясните, как ее исправить.
- ```
x = 1;
while (x <= 10);
 x++;
}
```
  - ```
for (y = .1; y != 1.0; y += .1)
    cout << y << endl;
```
 - ```
switch (n) {
 case 1:
 cout << "Число равно 1" << endl;
 case 2:
 cout << "Число равно 2" << endl;
 break;
 default:
 cout << "Число не равно ни 1, ни 2" << endl;
 break;
}
```
  - Следующие операторы должны печатать значения от 1 до 10

```
n = 1;
while (n < 10)
 cout << n++ << endl;
```

## Ответы на упражнения для самопроверки

- 2.1.** a) следование, выбор и повторение. b) `if/else`. c) управляемым счетчиком или определенным заранее. d) Метку, сигнал, флаг или ложный сигнал.
- 2.2.**
- ```
x = x + 1;
x += 1;
++x;
x++;
```
- 2.3.** a) `z = x++ + y;`
 b) `if (count > 10)
 count << "count больше, чем 10" << endl;`
 c) `total -= --x;`
 d) `q %= divisor;
 q = q % divisor;`
- 2.4.** a) `int sum, x;`
 b) `x = 1;`
 c) `sum = 0;`
 d) `sum += x; Of sum = sum + x ;`
 e) `cout << "Сумма равна " << sum << endl;`
- 2.5.** // Расчет суммы целых чисел от 1 до 10
`#include <iostream.h>`
- ```
main()
{
 int sum, x;
 x = 1;
 sum = 0;
 while (x <= 10) {
 sum += x;
 ++x;
 }
 cout << "Сумма равна " << sum << endl;
}
```
- 2.6.** a) `product = 25, x = 6;`  
 b) `quotient = 0, x = 6;`
- 2.7.** a) `cin >> x;`  
 b) `cin >> y;`  
 c) `i = 1;`  
 d) `power = 1;`  
 e) `power *= x; или power = power * x;`  
 f) `y++;`  
 g) `if (y <= x)`  
 h) `cout << power << endl;`

```
2.8. // Возвведение x в степень y
#include <iostream.h>

main()
{
 int x, y, i, power;

 i = 1;
 power = 1;
 cin>>x;
 cin >> y;
 while (i <= y) {
 power *= x;
 ++i;
 }
 cout << power << endl;
 return 0;
}
```

- 2.9. a) Ошибка: нет закрывающей фигурной скобки тела **while**.  
 Исправление: добавить закрывающую фигурную скобку после оператора `++c;`.
- b) Ошибка: использована операция *взять из потока* вместо *поместить в поток*.  
 Исправление: заменить `<<` на `>>`.
- c) Ошибка: точка с запятой после **else** ведет к логической ошибке.  
 Второй оператор вывода будет выполняться в любом случае.  
 Исправление: удалить точку с запятой после **else**.
- 2.10. Значение переменной **z** не изменяется в структуре **while**. Следовательно, если условие продолжения цикла (`z >= 0`) истинно, появляется бесконечный цикл. Чтобы избежать бесконечного цикла, **z** должна уменьшаться так, чтобы в конечном счете оказаться меньше 0.
- 2.11. a) Ошибочно. Метка **default** необязательна. Если нет необходимости производить какие-то действия по умолчанию, то метка **default** не нужна.
- b) Ошибочно. Оператор **break** используется для выхода из структуры **switch**. Если метка **default** последняя среди меток, то оператор **break** не требуется.
- c) Ошибочно. При использовании операции `&&` оба выражения отношения должны быть истинными, чтобы было истинным все выражение в целом.
- d) Правильно.

- 2.12. a) `sum = 0;`  
`for (count = 1; count <= 99; count += 2)`  
 `sum += count;`
- b) `cout << setiosflags(ios::fixed | ios::showpoint |`  
`ios::left)`  
 `<< setprecision(1) << setw(15) << 333.546372`  
 `<< setprecision(2) << setw(15) << 333.546372`  
 `<< setprecision(3) << setw(15) << 333.546372 << endl;`

В результате на выходе будет:

333.5    333.55    333.546

c) cout << setiosflags(ios::fixed / ios::showpoint)  
     << setprecision(2) << setw(10) << pow(2.5, 3) << endl;

На выходе будет:

15.63

d) x=1;  
     while (x <= 20) {  
         cout << x;  
         if (x % 5 == 0)  
             cout << endl;  
         else  
             cout << '\t';  
         x++;  
     }  
     e) for (x = 1; x <= 20; x++) {  
         cout << x;  
         if (x % 5 == 0)  
             cout << endl;  
         else  
             cout << '\t';  
     }

или

```
for (x = 1; x <= 20; x++)

 if (x % 5 == 0)

 cout << x << endl;

 else

 cout << x << '\t';
```

**2.13.** a) Ошибка: точка с запятой после заголовка **while** приводит к бесконечному циклу.

Исправление: заменить точку с запятой скобкой { или удалить ; и }.

b) Ошибка: использование числа с плавающей запятой для управления структурой повторения **for**.

Исправление: Использовать целое и осуществить соответствующие вычисления, чтобы получить желаемые значения

```
for (y = 1; y != 10; y++)

 cout << (float) y / 10 << endl;
```

c) Ошибка: отсутствие оператора **break** после операторов для первой метки **case**. Отметим, что это не обязательно является ошибкой, если программист хочет, чтобы операторы после **case 2:** выполнялись каждый раз, когда выполняются операторы после **case 1:**.

d) Ошибка: в условии продолжения повторения структуры **while** использована неправильная операция отношения.

Исправление: использовать <= вместо < или изменить 10 на 11.

## Упражнения

Упражнения 2.14–2.38 соответствуют разделам 2.1–2.12.

Упражнения 2.39–2.68 соответствуют разделам 2.13–2.21.

**2.14.** Найдите и исправьте ошибки в каждом из следующих фрагментов (в каждом фрагменте может быть более, чем одна ошибка):

- a) if (age >= 65) ;  
    cout << "Возраст более или равен 65" << endl;  
else  
    cout << "Возраст менее 65" << endl;
- b) int x = 1, total;  
  
while (x <= 10) {  
    total += x;  
    ++x;  
}  
  
c) while (x <= 100)  
    total += x;  
    ++x;
- d) while (y > 0) {  
    cout << y << endl;  
    ++y;  
}

**2.15.** Что напечатает следующая программа?

```
#include <iostream.h>
main()
{
 int y, x = 1, total = 0;
 while (x <= 10) {
 y = x * x;
 cout << y << endl;
 total += y;
 ++x;
 }
 cout << " Total равна " << total << endl;
}
```

Для упражнений 2.16–2.19 выполните следующие шаги:

1. Прочтите постановку задачи.
2. Сформулируйте алгоритм, используя псевдокод и нисходящую пошаговую детализацию.
3. Напишите программу на C++.
4. Проверьте, отладьте и выполните программу на C++.

**2.16.** Из-за высокой цены бензина водители озабочены затратами топлива своих автомобилей. Один водитель взял в рейс несколько емкостей бензина, записывая пройденные мили и бензин, использованный из каждой емкости. Разработайте программу на C++, которая вводила бы пробег в милях и бензин, использованный из каждой емкости. Программа должна рассчитывать и выводить на экран число миль на галлон для каждой емкости. После ввода исходных данных программа должна рассчитать и напечатать значение среднего числа миль на галлон, полученное для всех емкостей. Пример вывода:

Введите расход бензина (-1, если ввод закончен) : 12.8  
Введите пройденный путь: 287

Для этой емкости получено миль / галлон 22.421875  
 Введите расход бензина (-1, если ввод закончен) : 10.3  
 Введите пройденный путь: 200  
 Для этой емкости получено миль / галлон 19.417475

Введите расход бензина (-1, если ввод закончен) : 5  
 Введите пройденный путь: 120  
 Для этой емкости получено миль / галлон 24.000000

Введите расход бензина (-1, если ввод закончен) : -1

Средний расход бензина 21.601423

- 2.17. Разработайте программу на C++, которая будет определять, не превысили ли расходы клиента, имеющего депозитный счет, предела кредита. Для каждого клиента известны следующие данные:

1. Номер счета (целое).
2. Баланс с начала месяца.
3. Сумма всех расходов данного клиента в течение месяца.
4. Сумма всех приходов на счет данного клиента в течение месяца.
5. Допустимый размер кредита.

Программа должна ввести все эти данные, рассчитать новый баланс (равный начальному балансу + расход — приход) и определить, не превысил ли новый баланс предела кредита клиента. Для того клиента, чей кредит превышен, программа должна вывести на экран номер счета клиента, предел кредита, новый баланс и сообщение «Предел кредита превышен». Например:

Введите номер счета (-1, если ввод закончен) : 100  
 Введите начальный баланс: 5394.78  
 Введите сумму расходов: 1000.00  
 Введите сумму прихода: 500.00  
 Введите предел кредита: 5500.00  
 Счет: 100  
 Предел кредита: 5500.00  
 Баланс: 5894.78  
 Предел кредита превышен

Введите номер счета (-1, если ввод закончен) : 200  
 Введите начальный баланс: 1000.00  
 Введите сумму расходов: 123.45  
 Введите сумму прихода: 321.00  
 Введите предел кредита: 1500.00

Введите номер счета (-1, если ввод закончен) : 300  
 Введите начальный баланс: 500.00  
 Введите сумму расходов: 274.73  
 Введите сумму прихода: 100.00  
 Введите предел кредита: 800.00

Введите номер счета (-1, если ввод закончен) : -1

- 2.18. Одна большая химическая компания платит своим продавцам на основе комиссионных. Продавец получает \$200 в неделю плюс 9%

от объема продаж за неделю. Например, продавец, который продал за неделю химикалий на \$5000 получит \$200 плюс 9% от \$5000, то есть всего \$650. Разработайте программу на C++, которая должна вводить для каждого продавца объем его продаж за последнюю неделю, рассчитывать и выводить на экран его заработка. Данные вводятся поочередно для каждого продавца. Например:

Введите объем продаж в долларах (-1, если ввод закончен) :  
5000.00  
Заработка: \$650.00

Введите объем продаж в долларах (-1, если ввод закончен) :  
1234.56  
Заработка: \$311.11

Введите объем продаж в долларах (-1, если ввод закончен) :  
1088.89  
Заработка: \$298.00

Введите объем продаж в долларах (-1, если ввод закончен) : -1

- 2.19. Разработайте программу на C++, которая должна определять зарплатную плату для каждого из нескольких служащих. Компания выплачивает каждому служащему повременную зарплату за первые 40 часов работы и выплачивает в полуторном размере за все рабочие часы сверх 40. Вам дан список сотрудников компании, число часов, отработанных каждым за последнюю неделю, и почасовая ставка каждого сотрудника. Программа должна ввести эти данные для каждого сотрудника, рассчитать и вывести на экран его суммарную зарплату. Например:

Введите число рабочих часов (-1, если ввод закончен) : 39  
Введите почасовую ставку работника (\$00.00) : 10.00  
Зарплата: \$390.00

Введите число рабочих часов (-1, если ввод закончен) : 40  
Введите почасовую ставку работника (\$00.00) : 10.00  
Зарплата: \$400.00

Введите число рабочих часов (-1, если ввод закончен) : 41  
Введите почасовую ставку работника (\$00.00) : 10.00  
Зарплата: \$415.00

Введите число рабочих часов (-1, если ввод закончен) : -1

- 2.20. Во многих компьютерных приложениях часто используется поиск максимального числа, (т.е. максимального из заданной группы чисел). Например, программа, которая определяет победителя соревнования продавцов, должна вводить объемы продаж каждого продавца. Тот, у кого объем продаж выше, является победителем. Напишите псевдокод программы, а затем и саму программу на C++, которая вводит последовательно 10 чисел, определяет наибольшее из них и печатает его значение. Подсказка: ваша программа должна использовать следующие три переменные:

**counter:** счетчик для счета до 10 (для хранения количества введенных чисел и определения момента, когда введены все 10 чисел).

**number:** текущее введенное число.

**largest:** максимальное найденное число.

- 2.21.** Напишите программу на C++, использующую цикл и управляющую последовательность табуляции \t для печати следующей таблицы 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   |

- 2.22.** Используя подход упражнения 2.20, найдите *два* наибольших значения из 10 чисел. Указание: вы можете ввести каждое число только один раз.

- 2.23.** Модифицируйте программу на рис. 2.11 так, чтобы подтверждалась достоверность вводимой оценки. При любом вводе, даже если введено не 1 или 2, сохраняйте цикл, пока пользователь не введет правильное значение.

- 2.24.** Что напечатает следующая программа?

```
#include <iostream.h>
main ()
{
 int count = 1;
 while (count <= 10) {
 cout << (count % 2 ? "*****" : "++++++")
 << endl;
 ++count;
 }
 return 0;
}
```

- 2.25.** Что напечатает следующая программа?

```
#include <iostream.h>
main ()
{
 int row = 10, column;
 while (row >= 1) {
 column = 1;
 while (column <= 10) {
 cout << (row % 2 ? "<" : ">");
 ++column;
 }
 --row;
 cout << endl;
 }
 return 0;
}
```

**2.26. (Проблема обособленного *else*)** Определите напечатанный выходной результат для каждого из нижеприведенных фрагментов кода при  $x = 9$  и  $y = 11$ , и  $x = 11$  и  $y = 9$ . Отметим, что компилятор C++ игнорирует отступы в программе. Отметим также, что компилятор C++ всегда ассоциирует *else* с предшествующим *if*, пока ему не скажут об ином скобками *{ }* . Поскольку на первый взгляд программист может быть не уверен, какие *if* соответствуют каким *else*, это известно как проблема «обособленного *else*». Мы выбросили отступы из нижеследующих кодов, чтобы показать проблему более наглядно. (Совет: примените соглашение об отступах, которое вы изучили.)

- a) 

```
if (x < 10)
 if (y > 10)
 cout << "*****" << endl;
 else
 cout << "#####" << endl;
 cout << "$$$$$" << endl;
```
- b) 

```
if (x < 10) {
 if (y > 10)
 cout << "*****" << endl;
}
else {
 cout << "#####" << endl;
 cout << "$$$$$" << endl;
}
```

**2.27. (Другая проблема обособленного *else*)** Модифицируйте следующие фрагменты кода, чтобы получить указанный вывод на экран. Используйте соответствующую технику отступов. Вы можете не делать каких-либо других изменений, кроме вставки фигурных скобок. Компилятор C++ игнорирует отступы в программе. Мы выбросили отступы из нижеследующих кодов, чтобы показать проблему более наглядно. Примечание: возможно, что модификация не требуется.

```
if (y == 8)
if (x == 5)
cout << "|||||" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
cout << "&&&&&&" << endl;
```

- a) При  $x = 5$  и  $y = 8$  получите следующий результат

```
|||||
$$$$$
&&&&&
```

- b) При  $x = 5$  и  $y = 8$  получите следующий результат

```
|||||
```

- c) При  $x = 5$  и  $y = 8$  получите следующий результат

```
|||||
&&&&&
```

d) При  $x = 5$  и  $y = 7$  получите следующий результат. Подсказка: три последние оператора вывода после **else** являются частью составного оператора.

```
#####
$$$$
&&&&
```

**2.28.** Напишите программу, которая читает размер стороны квадрата и затем печатает звездочками и пробелами пустой квадрат заданного размера. Ваша программа должна работать для любых размеров, заданных в интервале 1–20. Например, если программа прочла размер 5, она должна напечатать:

```

* *
* *
* *

```

**2.29.** Палиндром — число или текст, который одинаково читается слева направо и справа налево. Например, каждое из следующих пятизначных целых чисел является палиндромом: 12321, 55555, 45554 и 11611. Напишите программу, которая читает пятизначные целые и определяет, являются ли они палиндромами. (Подсказка: используйте операции деление и вычисления остатка, чтобы выделить из числа отдельные разряды.)

**2.30.** Введите целые данные, содержащие только нули и единицы (т.е. «двоичные» целые), и напечатайте их десятичный эквивалент. (Подсказка. Используйте операции деление и вычисления остатка, чтобы «отрывать» разряды «двоичного» числа по одному справа налево. В десятичной системе самая правая цифра имеет позиционное значение 1, следующая цифра слева имеет позиционное значение 10, затем 100, затем 1000 и т.д.; в двоичной системе чисел самая правая цифра имеет позиционное значение 1, следующая цифра слева имеет позиционное значение 2, затем 4, затем 8 и т.д. Таким образом, десятичное число 234 может быть представлено как  $4 \cdot 1 + 3 \cdot 10 + 2 \cdot 100$ . Десятичным эквивалентом двоичного 1101 является  $1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8$  или  $1 + 0 + 4 + 8$  или 13.)

**2.31.** Напишите программу, которая выводит на экран следующий шаблон шахматной доски:

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

Ваша программа может использовать только три оператора вывода: один вида

```
cout << "*" ;
```

один вида

```
cout << ' ';
```

и один вида

```
cout << endl;
```

**2.32.** Напишите программу, которая постоянно печатает степени целого числа 2, соответственно 2, 4, 8, 16, 32, 64 и т.д. Число повторений вашего цикла не должно быть определено (т.е. вы должны создать бесконечный цикл). Что случилось, когда вы выполнили эту программу?

**2.33.** Напишите программу, которая читает радиус (как значение типа `float`), рассчитывает и печатает диаметр, длину окружности и площадь круга. Для р используйте значение 3.14159.

**2.34.** Чем ошибочен приведенный ниже оператор? Напишите правильный оператор, который бы выполнял то, что по всей вероятности пытался сделать программист.

```
cout << ++(x + y);
```

**2.35.** Напишите программу, которая читает три ненулевых значения типа `float`, определяет и печатает, могут ли они представлять стороны треугольника.

**2.36.** Напишите программу, которая читает три ненулевых целых числа, определяет и печатает, могут ли они представлять стороны прямоугольного треугольника.

**2.37.** Компания хочет передавать данные по телефону, но она обеспокоена возможностью телефонного перехвата. Все передаваемые данные являются четырехзначными целыми числами. Компания попросила вас написать программу, которая должна шифровать эти данные так, чтобы они могли передаваться с большей безопасностью. Ваша программа должна читать целые четырехзначные числа и шифровать их следующим образом: заменять каждую цифру значением *остатка от деления: (сумма этой цифры плюс 7) / 10*. Затем менять местами первую цифру с третьей и вторую с четвертой. Затем печатать полученное зашифрованное целое. Напишите отдельную программу, которая вводила бы зашифрованные четырехзначные целые и дешифровала их, получая исходные числа.

**2.38** Факториал неотрицательного целого *n* записывается как  $n!$  (произносится «эн факториал») и определяется следующим образом:

$n!=n\times(n-1)\times(n-2)\times\ldots\times1$  (для значений *n*, больших или равных 1)

и

$n!=1$  (для *n=0*).

Например,  $5!=54321=20$ .

a) Напишите программу, которая читает неотрицательное целое, рассчитывает и печатает его факториал.

b) Напишите программу, которая приблизительно рассчитывает значение математической константы *e*, используя формулу:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

c) Напишите программу, которая рассчитывает значение  $e^x$ , используя формулу:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

**2.39.** Найдите ошибки в следующих фрагментах (Замечание: ошибок может быть более одной):

a) `For (x = 100, x >= 1, x++)  
cout << x << endl;`

b) Следующий фрагмент должен печатать, является ли целая величина **value** нечетной, или четной:

```
switch (value % 2) {
 case 0:
 cout << "Четное целое" << endl;
 case 1:
 cout << "Нечетное целое" << endl;
}
```

c) Следующий код должен выводить нечетные целые от 19 до 1:

```
for (x = 19; x >= 1; x += 2)
 cout << x << endl;
```

d) Следующий код должен выводить четные целые от 2 до 100:

```
counter = 2;
do {
 cout << counter << endl;
 counter += 2;
} While (counter < 100);
```

**2.40.** Напишите программу, которая суммирует последовательность целых чисел. Полагайте, что первое прочитанное целое число указывает количество целых чисел, которые далее будут введены. Ваша программа должна читать только по одному значению в операторе ввода. Типичная входная последовательность может иметь вид

5 100 200 300 400 500

где 5 показывает, что будет введено последовательно 5 чисел, которые надо суммировать.

**2.41.** Напишите программу, которая подсчитывает и печатает среднее значение нескольких целых чисел. Полагайте, что последняя читаемая величина является меткой 9999. Типичная входная последовательность может иметь вид

10 8 11 7 9 9999

показывающая, что должно быть посчитано среднее значение чисел, предшествующих 9999.

**2.42** Что делает следующая программа?

```
#include <iostream.h>
main ()
```

```

{
 int x, y;
 cout << "Введите два целых числа в диапазоне 1-20:";
 cin >> x >> y;
 for (int i = 1; i <= y; i++) {
 for (int j = 1; j <= x; j++)
 cout << endl;
 }
 return 0;
}

```

- 2.43.** Напишите программу, которая находит наименьшее из нескольких целых. Полагайте, что первое прочитанное число задает количество последующих вводимых чисел.
- 2.44.** Напишите программу, которая считает и печатает произведение нечетных целых от 1 до 15.
- 2.45.** В теории вероятностей часто используется функция *факториал*. Факториал положительного целого *n* (*n!*) равен произведению положительных целых от 1 до *n*. Напишите программу вычисления факториалов целых чисел от 1 до 5. Напечатайте результаты в формате с табуляцией. Какие трудности могут препятствовать вам посчитать факториал 20?
- 2.46.** Модифицируйте программу расчета сложного процента в разделе 2.13 (рис. 2.21) так, чтобы расчет повторялся для ставок 5%, 6%, 7%, 8%, 9% и 10%. Используйте цикл *for* для варьирования ставки.
- 2.47.** Напишите программу, которая печатает следующие трафареты один под другим. Используйте цикл *for* для генерации трафаретов. Все звездочки (\*) должны печататься одним оператором вида *cout << '\*'*; (в результате звездочки будут печататься рядами). Подсказка: два последних трафарета требуют, чтобы каждая строка начиналась с соответствующего числа пробелов. Задача повышенной сложности: объедините ваши коды для решения четырех отдельных задач в единую программу, которая печатала бы все четыре трафарета рядом с помощью вложенных циклов *for*.

| (A)   | (B)   | (C)   | (D)   |
|-------|-------|-------|-------|
| *     | ***** | ***** | *     |
| **    | ***** | ***** | **    |
| ***   | ***** | ***** | ***   |
| ****  | ****  | ****  | ****  |
| ***** | ***   | ***   | ***** |
| ***** | **    | **    | ***** |
| ***** | *     | *     | ***** |

- 2.48.** Одно из интересных приложений компьютеров — рисование диаграмм и гистограмм. Напишите программу, которая читает пять чисел (каждое между 1 и 30). Для каждого просчитанного числа ваша программа должна напечатать строку, содержащую соответ-

ствующее число смежных звездочек. Например, если ваша программа прочла число 7, она должна напечатать \*\*\*\*\*.

- 2.49.** Торговый дом продает пять различных продуктов, розничная цена которых: продукт 1 — \$2.98, продукт 2 — \$4.50, продукт 3 — \$9.98, продукт 4 — \$4.49 и продукт 5 — \$6.87. Напишите программу, которая читает последовательность пар чисел, означающих:

1. номер продукта;
2. количество, проданное за день.

Ваша программа должна использовать оператор `switch`, который помогает определить розничную цену каждого продукта. Программа должна рассчитать и вывести на экран общую розничную стоимость всех проданных за неделю продуктов.

- 2.50.** Модифицируйте программу на рис. 2.22 так, чтобы она рассчитывала среднюю для класса оценку. Считайте, что вес оценки 'A' — 4 пункта, оценки 'B' — 3 пункта и т.д.

- 2.51.** Модифицируйте программу рис. 2.21 так, чтобы она использовала только целые для расчета сложного процента. (Подсказка: выразите все денежные суммы как целое число центов. Затем «разбейте» результат на две составляющие — доллары и центы, используя операции деления и вычисления остатка. Вставьте точку между dollarами и центами.)

- 2.52.** Положим, что  $i = 1$ ,  $j = 2$ ,  $k = 3$  и  $m = 2$ . Что напечатает каждый из приведенных операторов? Необходимы ли скобки в каждом случае?

- a) `cout << (i == 1) << endl;`
- b) `cout << (j == 3) << endl;`
- c) `cout << (i >= 1 && j < 4) << endl;`
- d) `cout << (m <= 99 && k < m) << endl;`
- e) `cout << (j >= i || k == m) << endl;`
- f) `cout << (k + m < j || 3 - j >= k) << endl;`
- g) `cout << (!m) << endl;`
- h) `cout << ( !(j - m) ) << endl;`
- i) `cout << ( !(r > m) ) << endl;`

- 2.53.** Напишите программу, которая печатает таблицу двоичных, восьмимерных и шестнадцатиричных эквивалентов десятичных чисел в диапазоне от 1 до 256. Если вы плохо знакомы с этими системами счисления, прочтите сначала приложение Г.

- 2.54.** Рассчитайте значение  $\pi$  на основании бесконечного ряда

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

Напечатайте таблицу, которая покажет, как значение  $\pi$  аппроксируется одним членом этого ряда, двумя членами, тремя и т.д. Сколько членов ряда потребовалось для получения значения 3.14? 3.141? 3.1415? 3.14159?

- 2.55. (Тройки Пифагора)** Прямоугольный треугольник может иметь все стороны, выраженные целыми числами. Множество троек целых значений сторон прямоугольного треугольника называется тройками Пифагора. Эти три стороны должны удовлетворять соотношению, по которому сумма квадратов двух сторон (катетов) равна квадрату третьей стороны (гипотенузы). Найдите все тройки Пифагора, в которых и катеты, и гипотенуза не больше 500. Используйте трижды вложенные циклы `for`, которые перебирают все возможности. Это пример вычисления «в лоб», сводящегося к перебору. Вы узнаете в более продвинутых курсах компьютерных вычислений, что есть много интересных проблем, для которых неизвестно других алгоритмов, кроме решения «в лоб».
- 2.56.** Компания платит своим сотрудникам или как менеджерам (которые получают фиксированный недельный оклад), или как почасовым работникам (которые получают фиксированную почасовую оплату за первые 40 рабочих часов и полуторную почасовую ставку за переработку сверх 40 часов), или как работникам на комиссионных началах (которые получают \$250 плюс 5.7% от суммы недельных продаж), или как сдельщикам (которые получают фиксированную сумму с каждой выработанной ими единицей продукции — каждый сдельщик в этой компании выпускает только один вид продукции). Напишите программу расчета еженедельной оплаты каждого сотрудника. Вы не знаете заранее числа сотрудников. Каждый тип сотрудника имеет свой код оплаты: менеджеры имеют код 1, почасовые работники — код 2, работающие на комиссионных началах имеют код 3, сдельщики — код 4. Используйте `switch`, основанный на этих кодах оплаты, для расчета выплат каждому сотруднику. В структуре `switch` предлагайте пользователю (клерку, составляющему расчетную ведомость) ввести соответствующие данные, необходимые для расчета выплаты сотруднику в соответствии с его кодом оплаты.
- 2.57. (Законы Моргана)** В данной главе мы рассмотрели логические операторы `&&`, и `!`. Законы Моргана помогают иногда выразить с их помощью логические выражения в более удобной форме. Эти законы гласят, что выражение `!(выражение1 && выражение2)` логически эквивалентно выражению `(! выражение1 || ! выражение2)`. Аналогично выражение `!(выражение1 || выражение2)` эквивалентно выражению `(! выражение1 && ! выражение2)`. Используйте законы Моргана для записи выражений, эквивалентных каждому из приведенных ниже, а затем напишите программу, которая показала бы в каждом случае эквивалентность первоначальных и новых выражений:
- `!(x < 5) && !(y >= 7)`
  - `!(a == b) || !(g != 5)`
  - `! ((x <= 8) && (y > 4))`
  - `! ((i > 4) || (j <= 6))`
- 2.58.** Напишите программу, которая напечатает следующий ромб. Вы можете использовать операторы вывода, которые печатают или одну звездочку (\*), или один пробел. Максимально используйте повторение.

ние (с вложенными структурами `for`) и минимизируйте число операторов вывода.

```

*

 *

```

**2.59.** Модифицируйте программу, которую вы написали в упражнении 2.58, чтобы она читала нечетное число в пределах от 1 до 19, определяющее число строк в ромбе. Ваша программа должна выводить на экран ромб соответствующего размера.

**2.60.** В адрес операторов `break` и `continue` раздается критика по поводу того, что они неструктурны. Действительно, операторы `break` и `continue` всегда могут быть заменены структурированными операторами, хотя часто это оказывается неудобным. Опишите, как в общем случае вы могли бы удалить из цикла оператор `break` и заменить его каким-то структурированным эквивалентом. (Подсказка. Оператор `break` осуществляет прерывание цикла в заданном месте его тела. Другим путем выхода из цикла является нарушение условия продолжения цикла. Рассмотрите использование в проверке условия продолжения цикла дополнительной проверки, устанавливающей, что «надо досрочно выйти из цикла, потому что выполнено условие его прерывания».) Используя такой прием, удалите оператор `break` из программы на рис. 2.26.

**2.61.** Что делает следующий фрагмент программы?

```

for (i = 1; i <= 5; i++) {
 for (j = 1; j <= 3; j++) {
 for (k = 1; k <= 4; k++)
 cout << '*';
 cout << endl;
 }
 cout << endl;
}

```

**2.62.** Опишите, как в общем случае вы могли бы удалить из цикла оператор `continue` и заменить его каким-то структурированным эквивалентом. Используя этот прием, удалите оператор `continue` из программы на рис. 2.27.

**2.63.** (*Песня «Двенадцать дней рождества»*) Напишите программу, использующую повторение и структуры `switch` для печати текста песни «Двенадцать дней рождества». Одна структура `switch` должна использоваться для печати дня («Первый», «Второй» и т.д.). Другая структура `switch` должна использоваться для печати остальной части каждого куплета (нашим читателям, возможно,

проще иметь дело с более знакомой им песней того же типа «Двадцать негритят». — *Прим. ред.*).

**Упражнение 2.64** относится к разделу 2.22 («Размышления об объектах»).

**2.64.** Опишите не более, чем в 200 словах, что такое автомобиль и что он делает. Составьте отдельные списки использованных имен существительных и глаголов. Мы полагаем, что каждое имя существительное соответствует объекту, который необходим для построения системы, в данном случае автомобиля. Выберите пять объектов из вашего списка и для каждого из них составьте список атрибутов и список вариантов поведения. Коротко опишите, как эти объекты взаимодействуют друг с другом и с другими объектами вашего описания. Тем самым вы осуществите несколько важных шагов типичного объектно-ориентированного проектирования.

# 3

## Функции



### Цели

- Понять, как можно конструировать программы по модульному принципу из небольших фрагментов, называемых функциями.
- Познакомиться с типовыми математическими функциями стандартной библиотеки С.
- Научиться создавать новые функции.
- Понять механизм обмена информацией между функциями.
- Познакомиться с техникой проведения расчетов с использованием генерации случайных чисел.
- Понять, как ограничивается область действия идентификаторов определенными частями программы.
- Понять, как можно писать и использовать функции, которые вызывают сами себя.

## План

- 3.1. Введение
- 3.2. Программные модули в C++
- 3.3. Математические библиотечные функции
- 3.4. Функции
- 3.5. Определение функций
- 3.6. Прототипы функций
- 3.7. Заголовочные файлы
- 3.8. Генерация случайных чисел
- 3.9. Пример: азартная игра
- 3.10. Классы памяти
- 3.11. Правила, определяющие область действия
- 3.12. Рекурсия
- 3.13. Пример использования рекурсии: последовательность чисел Фибоначчи
- 3.14. Рекурсии или итерации
- 3.15. Функции с пустыми списками параметров
- 3.16. Встраиваемые функции
- 3.17. Ссылки и ссылочные параметры
- 3.18. Аргументы по умолчанию
- 3.19. Унарная операция разрешения области действия
- 3.20. Перегрузка функций
- 3.21. Шаблоны функции
- 3.22. Размышления об объектах: идентификация атрибутов

*Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения*

### 3.1. Введение

Большинство компьютерных программ, решающих реальные практические задачи, намного превышают те программы, которые были представлены в первых главах. Экспериментально показано, что наилучшим способом создания и поддержки больших программ является их конструирование из малень-

ких фрагментов, или *модулей*, каждый из которых более управляем, чем сложная программа. Эта техника называется *разделяй и властвуй*. В этой главе описываются многие ключевые свойства языка C++, которые облегчают проектирование, реализацию, работу и сопровождение больших программ.

## 3.2. Программные модули в C++

Модули в C++ называются *функциями* и *классами*. Обычно программы на C++ пишутся путем объединения новых функций, которые пишет сам программист, с функциями, уже имеющимися в стандартной библиотеке C, и путем объединения новых классов, которые пишет сам программист, с классами, уже имеющимися в различных библиотеках классов. В этой главе мы сосредоточим внимание на функциях; классы мы будем детально обсуждать, начиная с главы 6.

Стандартная библиотека C обеспечивает широкий набор функций для выполнения типовых математических расчетов, операций со строками, с символами, ввода-вывода, проверки ошибок и многих других полезных операций. Это облегчает работу программиста, поскольку эти функции обладают многими из необходимых программисту свойств. Функции стандартной библиотеки C являются частью среды программирования C++.

### Хороший стиль программирования 3.1

Внимательно изучайте широкий набор функций в стандартной библиотеке ANSI C и классов в различных библиотеках классов.

### Замечание по технике программирования 3.1

Избегайте повторного изобретения колеса. Если возможно, используйте стандартную библиотеку ANSI C вместо того, чтобы писать новые функции. Это сокращает затраты времени на создание программы.

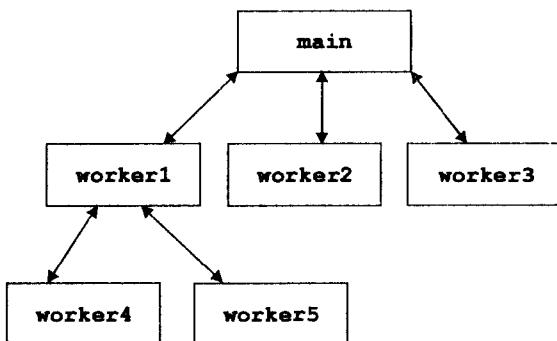
### Замечание по мобильности 3.1

Использование стандартной библиотеки ANSI C увеличивает мобильность программы.

Программист может написать функции, чтобы определить какие-то специфические задачи, которые можно использовать в разных местах программы. Эти функции иногда называют *функциями, определенными пользователем*. Операторы, реализующие данную функцию, пишутся только один раз и скрыты от других функций.

Функция *активизируется* (т. е. начинает выполнять запроектированную для нее задачу) путем *вызыва функции*. В вызове функции указывается ее имя и дается информация (в виде *аргументов*), необходимая вызываемой функции для ее работы. Это аналогично иерархической форме управления. Начальник (*вызывающая функция* или *вызывающий оператор*) просит подчиненного (*вызывающую функцию*) выполнить задание и *возвратить* (т.е. сообщить) результаты после того, как задание выполнено. Функция-начальник не знает, как функция-подчиненный выполняет порученное ей задание. Подчиненный может вызвать другие подчиненные функции, причем началь-

ник не будет осведомлен об этом. Далее мы увидим, как это «скрытие» деталей выполнения способствует разработке хорошего программного обеспечения. На рис. 3.1 показана функция `main`, иерархически связанная с некоторыми рабочими функциями. Заметим, что `worker1` действует как функция начальника по отношению к `worker4` и `worker5`. Взаимоотношения между функциями могут отличаться от показанных на этом рисунке иерархических отношений.



**Рис 3.1.** Иерархическое взаимоотношение функция-начальник / функция-подчиненный

### 3.3. Математические библиотечные функции

Математические библиотечные функции позволяют программисту выполнять определенные типовые математические вычисления. Мы будем использовать далее разнообразные математические библиотечные функции для иллюстрации самой концепции функций. Позже в этой книге мы обсудим многие другие функции из стандартной библиотеки С. Полный перечень функций стандартной библиотеки С приведен в приложении А.

Обычно функция вызывается путем записи имени функции, после которого записывается левая круглая скобка, затем *аргумент* функции (или список аргументов, разделенных запятыми), а завершает запись правая круглая скобка. Например, программист, желающий вычислить и напечатать квадратный корень из 900.0, мог бы написать

```
cout << sqrt(900.0);
```

При выполнении этого оператора вызывается математическая библиотечная функция `sqrt`, которая вычисляет квадратный корень из числа, заключенного в круглые скобки (900.0). Число 900.0 является аргументом функции `sqrt`. Приведенный выше оператор должен был бы напечатать 30. Функция `sqrt` получает аргумент типа `double` и возвращает результат типа `double`. Вообще все функции в математической библиотеке возвращают данные типа `double`.

#### Хороший стиль программирования 3.2

При использовании функций математической библиотеки включайте соответствующий заголовочный файл с помощью директивы препроцессора `#include <math.h>`.

### Типичная ошибка программирования 3.1

При использовании функций математической библиотеки забывают включать ее заголовочный файл, что приводит к ошибке компиляции. Стандартный заголовочный файл должен быть включен для любой стандартной библиотечной функции, используемой в программе.

Аргументами функции могут быть константы, переменные или выражения. Если  $c1 = 13.0$ ,  $d = 3.0$  и  $f = 4.0$ , то оператор

```
cout << sqrt(c1 + d*f)
```

вычислит и напечатает квадратный корень из  $13.0 + 3.0 * 4.0 = 25.0$ , а именно, напечатает 5, так как C++ обыкновенно не печатает в конце нуль или десятичную точку в числах с плавающей точкой, не имеющих дробной части).

Некоторые математические библиотечные функции приведены на рис. 3.2, где переменные  $x$  и  $y$  имеют тип **double**.

## 3.4. Функции

Функции позволяют пользователю использовать модульное программирование (составлять программу из модулей). Все переменные объявляются в описаниях функций *локальными переменными* — они известны только для функции, в которой они описаны. Большинство функций имеют список *параметров*, который обеспечивает значения для связующей информации между функциями. Параметры тоже являются локальными переменными.

### Замечание по технике программирования 3.2

В программах, содержащих много функций, **main** должна быть построена как группа вызовов функций, которые и выполняют основную часть работы.

Существует несколько причин для построения программ на основе функций. Подход «разделяй и властвуй» делает разработку программ более управляемой. Другая причина — *повторное использование программных кодов*, т.е. использование существующих функций как стандартных блоков для создания новых программ. Повторное использование — основной фактор развития объектно-ориентированного программирования. При продуманном присвоении имен функций и хорошем их описании программа может быть создана быстрее из стандартизованных функций, соответствующих определенным задачам. Третья причина — желание избежать в программе повторения каких-то фрагментов. Код, оформленный в виде функции, может быть выполнен в разных местах программы простым вызовом этой функции.

### Замечание по технике программирования 3.3

Каждая функция должна выполнять одну хорошо определенную задачу и имя функции должно наглядно отражать данную задачу. Это способствует успешному повторному использованию программного обеспечения.

| Функция                 | Описание                                              | Пример                                                                                                                                    |
|-------------------------|-------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sqrt(x)</code>    | корень квадратный из x                                | <code>sqrt(900.0) = 30.0</code><br><code>sqrt(9.0) = 3.0</code>                                                                           |
| <code>exp(x)</code>     | экспоненциальная функция $e^x$                        | <code>exp(1.0) = 2.718282</code><br><code>exp(2.0) = 7.389056</code>                                                                      |
| <code>log(x)</code>     | логарифм натуральный x (по основанию e)               | <code>log(2.718282) = 1.0</code><br><code>log(7.389056) = 2.0</code>                                                                      |
| <code>log10(x)</code>   | логарифм десятичный x (по основанию 10)               | <code>log10(1.0) = 0.0</code><br><code>log10(10.0) = 1.0</code><br><code>log10(100.0) = 2.0</code>                                        |
| <code>fabs(x)</code>    | абсолютное значение x                                 | если $x > 0$ , то <code>fabs(x) = x</code><br>если $x = 0$ , то <code>fabs(x) = 0.0</code><br>если $x < 0$ , то <code>fabs(x) = -x</code> |
| <code>ceil(x)</code>    | округление x до наименьшего целого, не меньшего чем x | <code>ceil(9.2) = 10.0</code><br><code>ceil(-9.8) = -9.0</code>                                                                           |
| <code>floor(x)</code>   | округление x до наибольшего целого, не большего чем x | <code>floor(9.2) = 9.0</code><br><code>floor(-9.8) = -10.0</code>                                                                         |
| <code>pow(x, y)</code>  | x в степени y                                         | <code>pow(2, 7) = 128.0</code><br><code>pow(9, 0.5) = 3.0</code>                                                                          |
| <code>fmod(x, y)</code> | остаток от x/y, как число с плавающей точкой          | <code>fmod(13.657, 2.333) = 1.992</code>                                                                                                  |
| <code>sin(x)</code>     | синус x (x в радианах)                                | <code>sin(0.0) = 0.0</code>                                                                                                               |
| <code>cos(x)</code>     | косинус x (x в радианах)                              | <code>cos(0.0) = 1.0</code>                                                                                                               |
| <code>tan(x)</code>     | тангенс x (x в радианах)                              | <code>tan(0.0) = 0.0</code>                                                                                                               |

Рис. 3.2. Наиболее употребительные математические библиотечные функции

#### **Замечание по технике программирования 3.4**

Если вы не можете выбрать подходящее имя, которое бы выражало суть работы функции, то может быть на нее возложено слишком много различных задач. Обычно лучше разбить такую функцию на несколько более мелких.

## **3.5 Определения функций**

Каждая программа, которую мы рассматривали, содержала функцию, называемую `main`, которая вызывает стандартные библиотечные функции для выполнения соответствующих им задач. Теперь мы рассмотрим, как программисты пишут свои собственные необходимые им функции. Рассмотрим программу, которая использует функцию `square` для вычисления квадратов целых чисел от 1 до 10 (рис. 3.3).

#### **Хороший стиль программирования 3.3**

Размещайте пустую строку между описаниями функций, чтобы отделить функции и облегчить чтение программы.

**Функция `square` активизируется или вызывается в `main` вызовом `square(x)`**

Функция создает копию значения *x* в *параметре у*. Затем *square* вычисляет *y \* y*. Результат передается в ту точку *main*, из которой была вызвана *square*, и затем этот результат выводится на экран. Благодаря структуре повторения *for* этот процесс повторяется десять раз.

Описание *square* показывает, что эта функция ожидает передачи в нее целого параметра *y*. Ключевое слово *int*, предшествующее имени функции, указывает, что *square* возвращает целый результат. Оператор *return* в *square* передает результат вычислений обратно в вызывающую функцию.

```
// функция square, определяемая программистом

#include <iostream.h>

int square(int); // прототип функции

main()
{
 for (int x = 1; x <= 10; x++)
 cout << square(x) << " ";
 cout << endl;

 return 0;
}

// описание функции
int square(int y)
{
 return y * y;
}
```

---

1    4    9    16    25    36    49    64    81    100

**Рис. 3.3.** Использование функции, определенной программистом

## Строка

```
int square(int);
```

является *прототипом функции*. Тип данных *int* в круглых скобках указывает компилятору, что функция *square* ожидает в операторе вызова целое значение аргумента. Тип данных *int* слева от имени функции *square* указывает компилятору, что *square* возвращает оператору вызова целый результат. Компилятор обращается к прототипу функции для проверки того, что вызовы функции *square* содержат правильный возвращаемый тип, правильное количество аргументов, правильный тип аргументов и правильный порядок перечисления аргументов. Прототипы функций детально обсуждаются в разделе 3.6.

**Формат описания функции имеет вид**

```
типа-возвращаемого-значения имя-функции (список-параметров)
(
 объявления и операторы
)
```

**Имя-функции** — это любой правильно написанный идентификатор. **Тип-возвращаемого-значения** — это тип данных результата, возвращаемого из функции оператору ее вызова. Тип возвращаемого значения **void** указывает, что функция не возвращает никакого значения. Компилятор предполагает тип **int** для неопределенного типа возвращаемого значения.

### Типичная ошибка программирования 3.2

Пропуск типа возвращаемого значения в описании функции вызывает синтаксическую ошибку, если прототип функции определяет возвращаемый тип иначе, чем **int**.

### Типичная ошибка программирования 3.3

Если забыть вернуть значение из функции, в которой предполагается возвращение результата, это может привести к неожиданным ошибкам. Описание языка C++ указывает, что результат такой оплошности не определен. В этом случае обычно компиляторы C++ выдают предупреждающее сообщение.

### Типичная ошибка программирования 3.4

Возвращение какого-то значения из функции, для которой тип возвращаемого значения объявлен как **void**, вызывает синтаксическую ошибку.

### Хороший стиль программирования 3.4

Несмотря на то, что пропущенный тип возвращаемого значения по умолчанию **int**, всегда задавайте тип возвращаемого значения явным образом. Исключением является функция **main**, для которой тип возвращаемого значения обычно не указывается.

**Список-параметров** — это список разделенных запятыми объявлений тех параметров, которые получает функция при ее вызове. Если функция не получает никаких значений, **список-параметров** задается как **void**. Тип должен быть указан явно для каждого параметра в списке параметров.

### Типичная ошибка программирования 3.5

Объявление параметров функции, имеющих одинаковый тип, в виде **float x, y** вместо **float x, float y**. Объявление параметра **float x, y** вызовет ошибку компиляции, так как типы надо указывать для каждого параметра в списке.

### Типичная ошибка программирования 3.6

Точка с запятой после правой круглой скобки, закрывающей список параметров в описании функции, является синтаксической ошибкой.

### Типичная ошибка программирования 3.7

Повторное определение параметра функции как локальной переменной этой функции является синтаксической ошибкой.

### **Хороший стиль программирования 3.5**

Не используйте одинаковые имена для аргументов, передаваемых в функцию, и соответствующих параметров в описании функции, хотя это и не является ошибкой. Использование разных имен помогает избежать двусмыслинности.

*Объявления и операторы внутри фигурных скобок образуют тело функции.* Тело функции рассматривается как блок. Блок — это просто составной оператор, который включает объявления. Переменные могут быть объявлены в любом блоке, а блоки могут быть вложенными. *При любых обстоятельствах функция не может быть описана внутри другой функции.*

### **Типичная ошибка программирования 3.8**

Описание функции внутри другой функции является синтаксической ошибкой.

### **Хороший стиль программирования 3.6**

Выбор осмысленных имен функций и осмысленных имен параметров делает программу более легко читаемой и помогает избежать излишних комментариев.

### **Замечание по технике программирования 3.5**

Обычно функция должна быть не длиннее одной страницы. Еще лучше, если она будет не длиннее половины страницы. Безотносительно к длине функции она должна хорошо определять только одну задачу. Небольшие функции способствуют повторному использованию программных кодов.

### **Замечание по технике программирования 3.6**

Программа должна быть написана как совокупность небольших функций. Это облегчает написание, отладку, сопровождение и модификацию программы.

### **Замечание по технике программирования 3.7**

Функция, требующая большого количества параметров, возможно, выполняет слишком много задач. Попробуйте разделить такую функцию на небольшие функции, которые выполняют отдельные задачи. Заголовок функции по возможности не должен занимать более одной строки.

### **Замечание по технике программирования 3.8**

Прототип функции, заголовок функции и вызовы функции должны быть согласованы между собой по количеству, типу и порядку следования аргументов и параметров и по типу возвращаемых результатов.

Существует три способа возврата управления к точке, из которой была вызвана функция. Если функция не должна возвращать результат, управление возвращается или просто при достижении правой фигурной скобки, завершающей функцию, или при выполнении оператора

```
return;
```

Если функция должна возвращать результат, то оператор  
`return выражение;`  
возвращает значение *выражения* в обращение к функции.

Наш второй пример использует определенную программистом функцию `maximum` для определения и возвращения наибольшего из трех целых чисел (рис. 3.4). Вводятся три целых числа. Затем эти целые числа передаются функции `maximum`, которая определяет наибольшее из чисел. Это значение возвращается функции `main` с помощью оператора `return` в `maximum` и печатается.

```
// Определение максимального из трех целых чисел
#include <iostream.h>

int maximum(int, int, int); // прототип функции

main()
{
 int a, b, c;

 cout << "Введите три целых числа: ";
 cin >> a >> b >> c;
 cout << "Максимум равен " << maximum(a, b, c) << endl;

 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;
}
```

Введите три целых числа: 22 85 17

Максимум равен 85

Введите три целых числа: 92 35 14

Максимум равен: 92

Введите три целых числа: 45 19 98

Максимум равен: 98

Рис. 3.4. Определенная программистом функция `maximum`

## 3.6. Прототипы функций

*Прототип функции* является одной из наиболее важных особенностей C++. Прототип функции указывает компилятору тип данных, возвращаемых функцией, количество параметров, которое ожидает функция, тип параметров и ожидаемый порядок их следования. Компилятор использует прототип функции для проверки правильности вызовов функции. Ранние версии С не выполняли такого рода проверку, поэтому был возможен неправильный вызов функции без обнаружения ошибок компилятором. Подобные вызовы приводили к неисправимым ошибкам выполнения или к хитроумным исправимым логическим ошибкам, которые было очень трудно обнаружить. Прототипы функции устранили этот недостаток.

### Замечание по технике программирования 3.9

В C++ требуются прототипы функций. Используйте директиву препроцессора `#include`, чтобы получить прототипы стандартных библиотечных функций из заголовочных файлов соответствующих библиотек. Используйте также `#include` для заголовочных файлов, содержащих прототипы функций, используемых вами или членами вашей группы.

Прототип функции `maximum` на рис. 3.4 имеет вид

```
int maximum(int, int, int);
```

Этот прототип указывает, что `maximum` имеет три аргумента типа `int` и возвращает результат типа `int`. Заметим, что прототип функции такой же, как заголовок описания функции `maximum`, за исключением того, что в него не включены имена параметров (`x`, `y` и `z`).

### Хороший стиль программирования 3.7

Имена параметров могут быть включены в прототипы функции с целью документирования. Компилятор эти имена игнорирует.

### Типичная ошибка программирования 3.9

Отсутствие точки с запятой в конце прототипа функции является синтаксической ошибкой.

Вызов функции, который не соответствует прототипу функции, ведет к синтаксической ошибке. Синтаксическая ошибка возникает также в случае отсутствия согласования между прототипом и описанием функции. Например, если бы в программе на рис. 3.4 прототип функции был бы написан так:

```
void maximum(int, int, int);
```

компилятор сообщил бы об ошибке, потому что возвращаемый тип `void` в прототипе функции отличался бы от возвращаемого типа `int` в заголовке функции.

Другой важной особенностью прототипов функций является *приведение типов аргументов*, т.е. задание аргументам подходящего типа. Например,

математическая библиотечная функция `sqrt` может быть вызвана с аргументом целого типа, даже если функция прототип в `math.h` определяет аргумент типа `double`, и при этом функция будет работать правильно. Оператор

```
cout << sqrt(4);
```

правильно вычисляет `sqrt(4)`, и печатает значение 2. Прототип функции заставляет компилятор преобразовать целое значение 4 в значение 4.0 типа `double` прежде, чем значение будет передано в `sqrt`. Вообще, значения аргументов, которые первоначально не соответствуют типам параметров в прототипе функции, преобразуются в подходящий тип перед вызовом функции. Эти преобразования могут привести к неверным результатам, если не руководствоваться *правилами приведения типов C++*. Правила приведения определяют, как типы могут быть преобразованы в другие типы без потерь. В приведенном выше примере `sqrt` тип `int` автоматически преобразуется в `double` без изменения его значений. Однако `double` преобразуется в `int` с отбрасыванием дробной части значения `double`. Преобразование больших целых типов в малые целые типы (например, `long` в `short`) может также привести к изменению значений.

Правила приведения типов применяются к выражениям, содержащим значения двух или более типов данных; такие выражения относятся к *выражениям смешанного типа*. Тип каждого значения в выражении смешанного типа приводится к «наивысшему» типу, имеющемуся в выражении (на самом деле создается и используется временная копия выражения — истинные значения остаются неизменными). На рис. 3.5 перечислены встроенные типы данных в порядке следования от высших типов к низшим.

Преобразование значений к низшему типу может привести к неверным значениям. Поэтому значения могут преобразовываться к низшему типу только путем явного присваивания значения переменной низшего типа, либо с помощью операции приведения к типу. Значения аргументов функции преобразуются к типам параметров в прототипе функции так, как если бы они были непосредственно присвоены переменным этих типов. Если наша функция `square`, которая использует параметр целого типа (рис. 3.3), вызывается с аргументом с плавающей точкой, аргумент преобразуется к типу `int` (низшему типу) и `square` обычно возвращает неверное значение. Например, `square(4.5)` возвратит 16 вместо 20.25.

### Типичная ошибка программирования 3.10

Преобразование от высшего типа в иерархии типов к низшему может изменить значение данных.

### Типичная ошибка программирования 3.11

Отсутствие прототипа функции, когда функция не определена перед ее первым вызовом, приводит к синтаксической ошибке.

### Замечание по технике программирования 3.10

Прототип функции, размещенный вне описания какой-то другой функции, относится ко всем вызовам данной функции, появляющимся после этого прототипа в данном файле. Прототип функции, размещенный внутри описания некоторой функции, относится только к вызовам внутри этой функции.

| Типы данных        |                          |
|--------------------|--------------------------|
| long double        |                          |
| double             |                          |
| float              |                          |
| unsigned long int  | (синоним unsigned long)  |
| long int           | (синоним long)           |
| unsigned int       | (синоним unsigned)       |
| int                |                          |
| unsigned short int | (синоним unsigned short) |
| short int          | (синоним short)          |
| unsigned char      |                          |
| char               |                          |

Рис. 3.5. Иерархия преобразований встроенных типов данных

## 3.7. Заголовочные файлы

Каждая стандартная библиотека имеет соответствующий заголовочный файл, содержащий прототипы всех функций библиотеки и объявления различных типов данных и констант, которые используются этими функциями. На рис. 3.6 в алфавитном порядке перечислены заголовочные файлы стандартной библиотеки C ANSI/ISO, которые можно включать в программы на C++. Термин «макрос», который несколько раз использован на рис. 3.6, подробно обсуждается в главе 17 «Препроцессор». Различные специальные заголовочные файлы C++ мы обсудим в этой книге позже.

Программист может сам создавать требующиеся ему заголовочные файлы. Заголовочные файлы, определяемые программистом, также должны иметь расширение .h. Заголовочные файлы, определяемые программистом, могут быть включены с помощью директивы препроцессора #include. Например, заголовочный файл square.h может быть включен в нашу программу директивой

```
#include "square.h"
```

в начале программы. В разделе 17.2 представлена дополнительная информация о включении заголовочных файлов.

## 3.8. Генерация случайных чисел

Теперь мы предпримем краткую, но, надеемся, увлекательную экскурсию в популярную область программирования, а именно в моделирование и игры. В этом и следующем разделах мы как следует разберемся в структурированных игровых программах, которые включают сложные функции. Эти программы используют большинство из изученных нами управляющих структур.

| Заголовочный файл стандартной библиотеки | Объяснение                                                                                                                                                                                                                                                                                                                                                                               |
|------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <assert.h>                               | Содержит макросы и информацию для дополнительной диагностики, помогающей при отладке программы.                                                                                                                                                                                                                                                                                          |
| <ctype.h>                                | Содержит прототипы функций, проверяющих некоторые свойства символов, и прототипы функций, которые могут быть использованы для преобразования строчных букв в заглавные и обратно.                                                                                                                                                                                                        |
| <errno.h>                                | Определяет макросы, полезные для получения сообщений об ошибках.                                                                                                                                                                                                                                                                                                                         |
| <float.h>                                | Содержит ограничения на числа с плавающей запятой в системе.                                                                                                                                                                                                                                                                                                                             |
| <limits.h>                               | Содержит общие ограничения системы.                                                                                                                                                                                                                                                                                                                                                      |
| <locale.h>                               | Содержит прототипы функций и другую информацию, которая позволяет модифицировать программу в зависимости от места ее выполнения. Позволяет компьютерной системе в зависимости от географического места выполнения программы применять различные соглашения относительно формы представления таких данных, как даты, время, долларовые расчеты и большие числа в различных регионах мира. |
| <math.h>                                 | Содержит прототипы математических библиотечных функций.                                                                                                                                                                                                                                                                                                                                  |
| <setjmp.h>                               | Содержит прототипы функций, которые позволяют обходить обычную последовательность вызова функции и возврата из нее.                                                                                                                                                                                                                                                                      |
| <signal.h>                               | Содержит прототипы функций и макросы для обработки различных ситуаций, которые могут возникнуть во время выполнения программы.                                                                                                                                                                                                                                                           |
| <stdarg.h>                               | Определяет макросы для работы со списком аргументов функции при неизвестном количестве и типе аргументов.                                                                                                                                                                                                                                                                                |
| <stddef.h>                               | Содержит объявления типов, используемых для выполнения некоторых вычислений.                                                                                                                                                                                                                                                                                                             |
| <stdio.h>                                | Содержит прототипы стандартных библиотечных функций ввода-вывода и используемую ими информацию.                                                                                                                                                                                                                                                                                          |
| <stdlib.h>                               | Содержит прототипы функций, преобразующих числа в текст и текст в числа, ведающих выделением памяти, генерирующих случайные числа и осуществляющих другие полезные операции.                                                                                                                                                                                                             |
| <string.h>                               | Содержит прототипы функций, обрабатывающих строки.                                                                                                                                                                                                                                                                                                                                       |
| <time.h>                                 | Содержит прототипы функций и типы для работы с временем и датами.                                                                                                                                                                                                                                                                                                                        |

Рис. 3.6. Заголовочные файлы стандартной библиотеки

Есть нечто такое в атмосфере казино с азартными играми, что подвигает людей любого склада проматывать деньги на плюшевых столах красного дерева или у одноруких бандитов. Это — элемент случайности, возможность превратить карманные деньги в истинное богатство. Элемент случайности может быть введен в компьютерные приложения с помощью функции `rand` из стандартной библиотеки С.

Рассмотрим следующий оператор:

```
i = rand();
```

Функция `rand` генерирует целое число в диапазоне между 0 и `RAND_MAX` (символическая константа, определенная в заголовочном файле `<stdlib.h>`). Значение `RAND_MAX` должно быть по меньшей мере равно 32767 — максимальное положительное значение двухбайтового (т.е. 16-би-

тогого) целого числа. Программы, представленные в этом разделе, были проверены на системе с максимальным значением **RAND\_MAX**, равным 32767. Если **rand** действительно вырабатывает случайные целые число, то при каждом вызове **rand** результирующее число имеет равную вероятность оказаться любым целым, лежащим между 0 и **RAND\_MAX**.

Диапазон значений, которые вырабатываются непосредственно функцией **rand**, отличается от диапазонов, которые требуются в специальных приложениях. Например, программа, моделирующая бросание монеты, требует только двух значений: 0 для «орла» и 1 для «решки». Программе, моделирующей метание кости с шестью гранями, должны бы потребоваться случайные целые числа в диапазоне от 1 до 6. Программа, которая случайным образом определяет тип следующего космического корабля (из четырех возможных), пересекающего горизонт в видеоигре, должна требовать случайные целые числа в диапазоне от 1 до 4.

Чтобы продемонстрировать **rand**, давайте разработаем программу моделирования 20 бросаний шестигранной игральной кости с печатью результата каждого бросания. Прототип функции **rand** можно найти в **<stdlib.h>**. Для того, чтобы выработать целые числа в диапазоне от 0 до 5, используем операцию вычисления остатка % в сочетании с **rand**:

```
rand() % 6
```

Это называется *масштабированием*. Число 6 называется *масштабирующим коэффициентом*. Затем мы *сдвигаем* диапазон чисел, добавляя 1 к полученному результату. Рис. 3.7 подтверждает, что результаты находятся в диапазоне от 1 до 6.

```
// Сдвинутые, масштабированные целые числа,
// полученные по закону 1 + rand() % 6

#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

main()
{
 for (int i = 1; i <= 20; i++)
 {
 cout << setw(10) << 1 + rand() % 6;

 if (i % 5 == 0)
 cout << endl;
 }
 return 0;
}
```

|   |   |   |   |   |
|---|---|---|---|---|
| 5 | 5 | 3 | 5 | 5 |
| 2 | 4 | 2 | 5 | 5 |
| 5 | 3 | 2 | 2 | 1 |
| 5 | 1 | 4 | 6 | 4 |

Рис. 3.7. Сдвинутые масштабированные целые числа, полученные по закону  $1 + \text{rand()} \% 6$

Чтобы показать, что эти числа имеют примерно одинаковую вероятность появления, давайте промоделируем с помощью программы, представленной на рис. 3.8, 6000 бросаний игральной кости. Каждое целое число от 1 до 6 должно появиться примерно 1000 раз.

Как показывает результат работы этой программы, с помощью функции `rand` и с применением масштабирования и сдвига мы действительно промоделировать бросание шестигранной игральной кости. Заметим, что в структуре `switch` не предусмотрен раздел `default`. После того как мы изучим массивы в главе 4, мы покажем, как можно изящно заменить всю структуру `switch` всего одним односторонним оператором.

Повторное выполнение программы на рис. 3.7 приводит к результату:

|   |   |   |   |   |
|---|---|---|---|---|
| 5 | 5 | 3 | 5 | 5 |
| 2 | 4 | 2 | 5 | 5 |
| 5 | 3 | 2 | 2 | 1 |
| 5 | 1 | 4 | 6 | 4 |

Заметьте, что печатается точно та же последовательность чисел, которая уже была раньше. Какие же это случайные числа? Как ни смешно, эта повторяемость является важной характеристикой функции `rand`. При отладке программы такая повторяемость имеет важное значение для доказательства того, что программа работает должным образом.

Функция `rand` на самом деле генерирует *псевдослучайные числа*. Повторный вызов `rand` производит последовательность чисел, которые кажутся случайными. Но та же самая последовательность повторяется при каждом повторении программы. Когда программа тщательно отлажена, она может быть использована для получения разных последовательностей случайных чисел при каждом выполнении.

Это называется *рандомизацией* и реализуется в законченном виде с помощью стандартной библиотечной функции `srand`. Функция `srand` получает целый аргумент `unsigned` и при каждом выполнении программы *задает начальное число*, которое функция `rand` использует для генерации последовательности квазислучайных чисел.

Использование `srand` демонстрируется программой, приведенной на рис. 3.9. В программе использован тип данных `unsigned`, что является краткой записью `unsigned int`. Число типа `int` занимает при хранении по меньшей мере два байта памяти и может иметь как положительное, так и отрицательное значение. Переменная типа `unsigned int` также хранится по меньшей мере в двух байтах памяти. Двухбайтовая `unsigned int` может иметь только положительные значения в диапазоне от 0 до 65535. Четырехбайтовая `unsigned int` может иметь только положительные значения в диапазоне от 0 до 4294967295. Функция `srand` получает значение `unsigned int` в качестве аргумента. Прототип функции `srand` находится в заголовочном файле `<stdlib.h>`.

Прогоните программу несколько раз и понаблюдайте результаты. Заметьте, что при прогоне программы с разными начальными значениями каждый раз получаются разные последовательности случайных чисел.

```

// Бросание шестигранной игральной кости 6000 раз
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

main()
{
 int frequency1 = 0, frequency2 = 0,
 frequency3 = 0, frequency4 = 0,
 frequency5 = 0, frequency6 = 0;

 for (int roll = 1; roll <= 6000; roll++) {
 int 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;
 case 6:
 ++frequency6;
 break;
 }
 }

 cout << "Грань" << setw(13) << "Частота"
 << endl << " 1" << setw(13) << frequency1
 << endl << " 2" << setw(13) << frequency2
 << endl << " 3" << setw(13) << frequency3
 << endl << " 4" << setw(13) << frequency4
 << endl << " 5" << setw(13) << frequency5
 << endl << " 6" << setw(13) << frequency6 << endl;

 return 0;
}

```

| Грань | Частота |
|-------|---------|
| 1     | 987     |
| 2     | 984     |
| 3     | 1029    |
| 4     | 974     |
| 5     | 1004    |
| 6     | 1022    |

Рис. 3.8. Бросание шестигранной игральной кости 6000 раз

```
// Программа randomизации бросания игральной кости
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

main()
{
 unsigned seed;

 cout << "Введите число: ";
 cin >> seed;
 srand(seed);

 for (int i = 1; i <= 10; i++) {
 cout << setw(10) << 1 + rand() % 6;

 if (i % 5 == 0)
 cout << endl;
 }
 return 0;
}
```

---

**Введите число: 67**

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 6 | 5 | 1 | 4 |
| 5 | 6 | 3 | 1 | 2 |

**Введите число: 432**

|   |   |   |   |   |
|---|---|---|---|---|
| 4 | 2 | 6 | 4 | 3 |
| 2 | 5 | 1 | 4 | 4 |

**Введите число: 67**

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 6 | 5 | 1 | 4 |
| 5 | 6 | 3 | 1 | 2 |

**Рис. 3.9.** Программа randomизации бросания игральной кости

Если мы хотим randomизировать не вводя каждый раз начальное число, можно использовать оператор, подобный следующему

```
srand(time(NULL));
```

При этом для автоматического получения начального числа компьютер считывает показания своих часов. Функция `time` (с аргументом `NULL`, как записано в указанном выше операторе) возвращает текущее «календарное время» в секундах. Это значение преобразуется в беззнаковое целое число и используется как начальное значение в генераторе случайных чисел. Прототип функции для `time` находится в `<time.h>`.

Значение, генерируемое функцией `rand`, всегда находится в диапазоне:

$0 \leq \text{rand}() \leq \text{RAND\_MAX}$

Выше мы показали, что смоделировать бросание шестигранной игральной кости можно с помощью всего одного оператора:

```
face = 1 + rand() % 6;
```

который присваивает переменной `face` целое (и случайное) значение в диапазоне  $1 < \text{face} < 6$ . Заметим, что ширина этого диапазона (т.е. число последовательных целых значений, расположенных в нем) равна 6, а начало диапазона равно 1. Обращаясь к предыдущему оператору, мы видим, что ширина диапазона определяется числом, используемым для масштабирования `rand` с помощью операции вычисления остатка (т. е. 6), а начало диапазона равно числу, которое прибавляется к `rand % 6` (т. е. 1). Мы можем записать этот результат в общем виде:

```
n = a + rand() % b;
```

где `a` — величина сдвига (которая равна первому числу в требуемом диапазоне последовательных целых чисел), а `b` — масштабирующий коэффициент (который равен ширине требуемого диапазона целых чисел). В приведенных далее упражнениях мы увидим, что можно выбирать целые числа наугад из набора значений, отличных от диапазона последовательных целых чисел.

### Типичная ошибка программирования 3.12

Использование `srand` вместо `rand` для генерации случайных чисел.

## 3.9. Пример: азартная игра

Одной из наиболее популярных азартных игр считается игра в кости, известная как «крепс», в которую играют и в казино, и в глухих закоулках по всему миру. Правила игры просты:

*Игрок бросает две кости. Каждая кость имеет шесть граней. Эти грани помечены как 1, 2, 3, 4, 5 и 6. После броска вычисляется сумма цифр двух верхних граней. Если сумма после первого броска равна 7 или 11, игрок выиграл. Если после первого броска сумма равна 2, 3 или 12 (это называется «крепс»), игрок проигрывает (т. е. выигрывает «банк»). Если после первого броска сумма равна 4, 5, 6, 8, 9 или 10, то эта сумма становится «очками» игрока. Чтобы выиграть, игрок должен продолжать бросать кости до тех пор, пока не выпадет сумма, равная его очкам. Игрок проигрывает, если во время этих бросков ему выпадет сумма 7.*

Программа на рис. 3.10 моделирует игру крепс. На рис 3.11 показано несколько примеров ее выполнения.

Заметим, что игрок должен бросать каждый раз по две кости. Мы описали функцию `rollDice`, которая имитирует бросок костей, подсчитывает выпавшую сумму и печатает ее. Функция `rollDice` описана один раз, но она вызывается из двух мест программы. Интересно, что `rollDice` не требует аргументов, поэтому в списке параметров мы указали `void`. Функция `rollDice` возвращает сумму двух костей, возвращаемый тип `int` указывается в заголовочном файле.

Игра разумно запутана. Игрок может выиграть или проиграть после первого же броска или после серии бросков. Переменная `gameStatus` используется для запоминания состояния игры. Переменная `gameStatus` объявлена как имеющая тип с именем `Status`. Стока

```
enum Status { CONTINUE, WON, LOST };
```

объявляет определенный пользователем тип, называемый *перечислимым* (enumeration). Перечислимый тип, вводимый ключевым словом `enum` перед именем

типа (в нашем случае **Status**), является набором целых именованных констант, представленных своими идентификаторами. Значения констант этого списка перечисления начинаются, если не указано иное, с 0 и увеличиваются последовательно на 1. В предыдущем списке перечисления имя **CONTINUE** присвоено значению 0, **WON** присвоено значению 1, а **LOST** — значению 2. Идентификаторы в enum должны быть уникальными, но отдельные константы перечисления могут иметь одинаковые целые значения.

```
// Крепс
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

int rollDice(void);

main()
{
 enum Status { CONTINUE, WON, LOST };

 int sum, myPoint;
 Status gameStatus;

 srand(time(NULL));
 sum = rollDice(); // первый бросок костей

 switch(sum) {
 case 7: case 11: // выигрыш после первого броска
 gameStatus = WON;
 break;
 case 2: case 3: case 12: // проигрыш после первого броска
 gameStatus = LOST;
 break;
 default: // запоминание очков
 gameStatus = CONTINUE;
 myPoint = sum;
 cout << "Очки: " << myPoint << endl;
 break;
 }

 while (gameStatus == CONTINUE) { // бросать дальше
 sum = rollDice();

 if (sum == myPoint) // выигрыш по очкам
 gameStatus = WON;
 else
 if (sum == 7) // проигрыш после суммы 7
 gameStatus = LOST;
 }

 if (gameStatus == WON)
 cout << "Игрок выиграл" << endl;
 else
 cout << "Игрок проиграл" << endl;
}

}
```

**Рис. 3.10.** Программа моделирования игры в крепс (часть 1 из 2)

```

int rollDice(void)
{
 int die1, die2, workSum;

 die1 = 1 + rand() % 6;
 die2 = 1 + rand() % 6;
 workSum = die1 + die2;
 cout << "Бросок игрока " << die1 << " + " << die2
 << " = " << workSum << endl;

 return workSum;
}

```

Рис. 3.10. Программа моделирования игры в крепс (часть 2 из 2)

```

Бросок игрока 6 + 5 = 11
Игрок выиграл
...
Бросок игрока 6 + 6 = 12
Игрок проиграл
...
Бросок игрока 4 + 6 = 10
Очки: 10
Бросок игрока 2 + 4 = 6
Бросок игрока 6 + 5 = 11
Бросок игрока 3 + 3 = 6
Бросок игрока 6 + 4 = 10
Игрок выиграл

Бросок игрока 1 + 3 = 4
Очки: 4
Бросок игрока 1 + 4 = 5
Бросок игрока 5 + 4 = 9
Бросок игрока 4 + 6 = 10
Бросок игрока 6 + 5 = 11
Бросок игрока 1 + 2 = 3
Бросок игрока 5 + 2 = 7
Игрок проиграл

```

Рис. 3.11. Примеры прогонов игры в крепс

### Хороший стиль программирования 3.8

Делайте заглавной первую букву идентификатора, используемого как имя типа, определенного пользователем.

Переменным типа **Status**, определенного пользователем, может быть присвоено только одно из трех значений, объявленных в перечислении. Если игра выиграна, **gameStatus** принимает значение **WON**. Если игра проиграна, **gameStatus** устанавливается равной **LOST**. В ином случае **gameStatus** принимает значение **CONTINUE**, так что кости могут быть брошены снова.

### Типичная ошибка программирования 3.13

Присвоение целого эквивалента константы перечисления переменной перечислимого типа приводит к замечанию (предупреждению) компилятора.

Другой популярный пример перечислимого типа:

```
enum Months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

Это объявление создает определенный пользователем тип **Months** с константами перечисления, представляющими месяцы года. Поскольку первое значение приведенного перечисления установлено равным 1, оставшиеся значения увеличиваются на 1 от 1 до 12. В объявлении перечислимого типа любой константе перечисления можно присвоить целое значение.

### Типичная ошибка программирования 3.14

После того, как константа перечисления определена, попытка присвоить ей другое значение является синтаксической ошибкой.

### Хороший стиль программирования 3.9

Используйте в именах констант перечисления только заглавные буквы. Это выделяет константы в тексте программы и напоминает программисту о том, что константы перечисления не являются переменными.

### Хороший стиль программирования 3.10

Использование перечислений вместо целых констант облегчает чтение программы.

После первого бросания, если игра выиграна, структура **while** пропускается, поскольку **gameStatus** не равна **CONTINUE**. Программа передает управление в структуру **if/else**, которая печатает «Игрок выиграл», если **gameStatus** равна **WON**, и «Игрок проиграл», если **gameStatus** равна **LOST**.

После первого бросания, если игра не окончена, **sum** сохраняется в **myPoint**. Управление передается структуре **while**, потому что **gameStatus** равна **CONTINUE**. При каждом выполнении **while** вызывается **rollDice** для вычисления нового значения переменной **sum**. Если **sum** совпадает с **myPoint**, **gameStatus** устанавливается равной **WON**, проверка **while** терпит неудачу, структура **if/else** печатает «Игрок выиграл» и выполнение завершается. Если **sum** равна 7, **gameStatus** устанавливается в **LOST**, проверка **while** терпит неудачу, структура **if/else** печатает «Игрок проиграл» и выполнение завершается.

Обратите внимание на интересные применения различных механизмов управления вычислениями, которые мы обсудили. Программа «крепс» использует две функции — **main** и **rollDice**, структуры **switch**, **while**, **if/else** иложенную структуру **if**. В упражнениях мы исследуем различные интересные характеристики игры крепс.

## 3.10. Классы памяти

В главах с 1 по 3 мы использовали идентификаторы для имен переменных. Атрибутами переменных являются имя, тип, размер и значение. В этой главе мы используем также идентификаторы в качестве имен функций, определенных пользователем. В действительности каждый идентификатор в

программе имеет другие атрибуты, включающие *класс памяти*, *область действия* и *компоновка*.

В C++ имеется четыре спецификации класса памяти: *auto*, *register*, *extern* и *static*. Спецификация класса памяти идентификатора помогает определить его класс памяти, область действия и пространство имен.

*Класс памяти* идентификатора определяет его *время жизни* — период, в течение которого этот идентификатор существует в памяти. Одни идентификаторы существуют недолго, другие — неоднократно создаются и уничтожаются, третий — существуют на протяжении всего времени выполнения программы.

*Областью действия* (*областью видимости*) идентификатора называется область программы, в которой на данный идентификатор можно сослаться. На некоторые идентификаторы можно сослаться в любом месте программы, тогда как на другие — только в определенных частях программы.

*Компоновка* идентификатора определяет для программ с многими исходными файлами (этую тему мы начнем изучать в главе 6), известен ли идентификатор только в одном текущем исходном файле или же в любом исходном файле с соответствующим объявлением.

В этом разделе обсуждаются четыре спецификации класса памяти и два класса памяти. В разделе 3.11 обсуждается область действия идентификаторов.

Спецификации класса памяти могут быть разбиты на два класса: *автоматический класс памяти с локальным временем жизни* и *статический класс памяти с глобальным временем жизни*. Ключевые слова *auto* и *register* используются для объявления переменных с локальным временем жизни. Такие переменные создаются при входе в блок, в котором они объявлены, они существуют лишь во время активности блока и исчезают при выходе из блока.

К классу с локальным временем жизни могут относиться только переменные. К этому классу относятся локальные переменные функций и параметры функций. Спецификация *auto* явно объявляет переменные автоматического класса памяти, т.е. с локальным временем жизни. Например, следующее объявление указывает, что переменные *x* и *y* типа *float* являются переменными с локальным временем жизни, т.е. они существуют только в теле функции, в которой появляется объявление:

```
auto float x, y;
```

Локальные переменные являются переменными с локальным временем жизни по умолчанию, так что ключевое слово *auto* используется редко. Далее мы будем ссылаться на переменные автоматического класса памяти просто как на автоматические переменные.

### **Совет по повышению эффективности 3.1**

Автоматический класс памяти — это средство экономии памяти, так как переменные этого класса создаются при входе в блок, в котором они объявлены, и уничтожаются при выходе из блока.

### **Замечание по технике программирования 3.11**

Автоматическое хранение — еще один пример принципа наименьших привилегий. Зачем хранить в памяти и делать доступными переменные, когда они не нужны?

Обычно данные в версии программы на машинном языке, загружаются для расчетов и другой обработки в регистры.

### **Совет по повышению эффективности 3.2**

Спецификация класса памяти **register** может быть помещена перед объявлением автоматической переменной, чтобы компилятор сохранял переменную не в памяти, а в одном из высокоскоростных аппаратных регистров компьютера. Если интенсивно используемые переменные, такие как счетчики или суммы могут сохраняться в аппаратных регистрах, накладные расходы на повторную загрузку переменных из памяти в регистр и обратную загрузку результата в память могут быть исключены.

### **Типичная ошибка программирования 3.15**

Использование нескольких спецификаций класса памяти для одного идентификатора. Для идентификатора может быть указана только одна спецификация класса памяти. Например, если вы указали **register**, нельзя указать также и **auto**.

Компилятор может проигнорировать объявления **register**. Например, может оказаться недостаточным количество регистров, доступных компилятору для использования. Приведенное ниже объявление определяет, что целая переменная **counter** должна быть помещена в один из регистров компьютера; независимо от того, сделает это компилятор или нет, **counter** получит начальное значение 1:

```
register int counter = 1;
```

Ключевое слово **register** может применяться только к локальным переменным и параметрам функций.

### **Совет по повышению эффективности 3.3**

Часто объявления **register** не являются необходимыми. Современные оптимизирующие компиляторы способны распознавать часто используемые переменные и решать, помещать их в регистры или нет, не требуя от программиста объявления **register**.

Ключевые слова **extern** и **static** используются, чтобы объявить идентификаторы переменных и функций как идентификаторы статического класса памяти с глобальным временем жизни. Такие переменные существуют с момента начала выполнения программы. Для таких переменных память выделяется и инициализируется сразу после начала выполнения программы. Имена функций тоже существуют с самого начала выполнения программы. Однако это не означает, что эти идентификаторы могут быть использованы по всей программе. Как будет видно из раздела 3.11, класс памяти и область действия (где можно использовать имя) — это разные вещи.

Существует два типа идентификаторов статического класса памяти: внешние идентификаторы (такие, как глобальные переменные и имена функций), и локальные переменные, объявленные спецификацией класса памяти **static**. Глобальные переменные и имена функций по умолчанию относятся к классу памяти **extern**. Глобальные переменные создаются путем размещения их объявлений вне описания какой-либо функции. Глобальные переменные сохраняют свои значения в течение всего времени выполнения программы. На глобальные переменные и функции может ссылаться любая функция, которая расположена после их объявления или описания в файле.

### **Замечание по технике программирования 3.12**

Объявление переменной глобальной, а не локальной, приводит к неожиданным побочным эффектам, когда функции, не нуждающиеся в доступе к этой переменной, случайно или намеренно изменяют ее. Вообще лучше избегать использования глобальных переменных за исключением особых случаев, когда требуется уникальная производительность (это будет рассмотрено в главе 18).

### **Хороший стиль программирования 3.11**

Переменные, используемые только в отдельной функции, предпочтительнее объявлять как локальные переменные этой функции, а не как глобальные переменные.

Локальные переменные, объявленные с ключевым словом **static**, известны только в той функции, в которой они определены, но в отличие от автоматических переменных локальные переменные **static** сохраняют свои значения в течение всего времени существования функции. При каждом следующем вызове функции локальные переменные содержат те значения, которые они имели при предыдущем вызове. Следующий оператор объявляет локальную переменную **count** как **static** и присваивает начальное значение 1

```
static int count = 1;
```

Все числовые переменные статического класса памяти принимают нулевые начальные значения, если программист явно не указал другие начальные значения. (Статические переменные — указатели, обсуждаемые в главе 5, тоже имеют нулевые начальные значения).

Спецификации класса памяти **extern** и **static** имеют специальное значение, когда они применяются явно к внешним идентификаторам. В главе 18, «Другие темы», мы обсудим явное использование **extern** и **static** с внешними идентификаторами и программами с множеством исходных файлов.

## **3.11. Правила, определяющие область действия**

*Область действия идентификатора* — это часть программы, в которой на идентификатор можно ссылаться. Например, когда мы объявляем локальную переменную в блоке, на нее можно ссылаться только в этом блоке или в блоке, вложенном в этот блок. Существуют четыре области действия идентификатора — *область действия функция*, *область действия файл*, *область действия блок* и *область действия прототип функции*.

Идентификатор, объявленный вне любой функции (на внешнем уровне), имеет *область действия файл*. Такой идентификатор «известен» всем функциям от точки его объявления до конца файла. Глобальные переменные, описания функций и прототипы функций, находящиеся вне функции — все они имеют областью действия файл.

Метки (идентификаторы с последующим двоеточием, например, **start:**) — единственные идентификаторы, имеющие *областью действия функцию*. Метки можно использовать всюду в функции, в которой они появились, но на них нельзя ссылаться вне тела функции. Метки используются в структурах **switch** (как метки **case**) и в операторах **goto** (смотри главу 18 «Другие темы»). Метки — относятся к тем деталям реализации, которые функции «прятут» друг от друга. Это скрытие — более формально называемое *скрытие (утилизация)*.

вание) информации — один из наиболее фундаментальных принципов разработки хорошего программного обеспечения.

Идентификаторы, объявленные внутри блока (на внутреннем уровне), имеют область действия блок. Область действия блок начинается объявлением идентификатора и заканчивается конечной правой фигурной скобкой блока. Локальные переменные, объявленные в начале функции, имеют область действия блок так же, как и параметры функции, являющиеся локальными переменными. Любой блок может содержать объявления переменных. Если блоки вложены и идентификатор во внешнем блоке имеет такое же имя, как идентификатор во внутреннем блоке, идентификатор внешнего блока «невидим» (скрыт) до момента завершения работы внутреннего блока. Это означает, что пока выполняется внутренний блок, он видит значение своих собственных локальных идентификаторов, а не значения идентификаторов с идентичными именами в охватывающем блоке. Локальные переменные, объявленные как `static`, имеют область действия блок, несмотря на то, что они существуют с самого начала выполнения программы.

Единственными идентификаторами с областью действия прототипа функции являются те, которые используются в списке параметров прототипа функции. Как ясно из предыдущего, прототипы функций не требуют имен в списке параметров — требуются только типы. Если в списке параметров прототипа функции используется имя, компилятор это имя игнорирует. Идентификаторы, используемые в прототипе функции, можно повторно использовать где угодно в программе, не опасаясь двусмыслиности.

### **Типичная ошибка программирования 3.16**

Непредумышленное использование одинаковых имен идентификаторов во внутреннем и внешнем блоках, когда на самом деле программист хочет, чтобы идентификатор во внешнем блоке был активным во время работы внутреннего блока.

### **Хороший стиль программирования 3.12**

Избегайте применения имен переменных, которые незримо уже используются во внешних областях действия. Этого можно достигнуть, вообще избегая использования в программе одинаковых идентификаторов.

Программа на рис. 3.12 демонстрирует области действия глобальных переменных, автоматических локальных переменных, и локальных переменных типа `static`.

Объявленной глобальной переменной `x` присвоено начальное значение 1. Эта глобальная переменная невидима в любом блоке (функции), в котором объявляется переменная с именем `x`. В `main` объявляется локальная переменная `x`, которой присваивается начальное значение 5. Переменная печатается, чтобы показать, что в `main` невидима глобальная `x`. Далее, в `main` определяется новый блок, с другой локальной переменной `x`, которой присваивается начальное значение 7. Эта переменная печатается, чтобы показать, что она делает невидимой `x` внешнего блока `main`. Переменная `x` со значением 7 автоматически разрушается при выходе из блока, после чего печатается локальная переменная `x` внешнего блока функции `main`, чтобы показать, что она теперь не скрыта. Программа определяет три функции — каждая из них не требует аргументов и ничего не возвращает. Функция `a` определяет автоматическую переменную `x` и присваивает ей на-

чальное значение 25. При вызове **a** переменная печатается, получает приращение и печатается снова перед выходом из функции. Автоматическая переменная **x** создается заново и ей присваивается начальное значение 25 при каждом вызове этой функции. Функция **b** объявляет переменную **x** как **static** и присваивает ей начальное значение 50. Локальные переменные, объявленные как **static**, сохраняют свои значения, даже когда они находятся за пределами области действия. При вызове **b** переменная **x** печатается, получает приращение и печатается снова перед выходом из функции. При следующем вызове этой функции локальная переменная **x** типа **static** будет содержать значение 51. Функция **c** не объявляет никаких переменных. Поэтому, когда она ссылается на переменную **x**, то использует глобальную переменную **x**. При вызове **c** глобальная переменная **x** печатается, умножается на 10 и печатается снова перед выходом из функции. При следующем вызове этой функции глобальная переменная **x** будет содержать это модифицированное значение 10. Окончательно, программа снова печатает локальную переменную **x** в **main**, чтобы показать, что ни одна из функций не вызывает изменения значения **x**, так как все функции ссылаются на переменные в другой области действия.

```
// Пример областей действия
#include <iostream.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 main

 cout << "локальная x во внешней области действия main = "
 << x << endl;
 { // начало новой области действия
 int x = 7;

 cout <<"локальная x во внутренней области действия main = "
 << x << endl;
 } // конец новой области действия

 cout << "локальная x во внешней области действия main = "
 << x << endl;

 a(); // а имеет автоматическую локальную переменную x
 b(); // б имеет статическую локальную переменную x
 c(); // с использует глобальную переменную x
 a(); // а заново дает начальное значение x
 b(); // статическая локальная x сохраняет предыдущее
 // значение
 c(); // глобальная x также сохраняет свое значение

 cout << "локальная x в main = " << x << endl;
 return 0;
}
```

Рис. 3.12. Пример областей действия (часть 1 из 2)

```

void a(void)
{
 int x = 25; // каждый раз а присваивается начальное значение
 cout << endl << "локальная переменная x в а = " << x
 << " после входа в а" << endl;
 ++x;
 cout << "локальная переменная x в а = " << x
 << " перед выходом из а" << endl;
}

void b(void)
{
 static int x = 50; // Начальное значение присваивается только
 // при первом вызове b
 cout << endl << "локальная статическая переменная x = " << x
 << " при входе в b" << endl;
 ++x;
 cout << "локальная статическая переменная x = " << x
 << " при выходе из b" << endl;
}
void c(void)
{
 cout << endl << "глобальная переменная x = " << x
 << " при входе в с" << endl;
 x *= 10;
 cout << "глобальная переменная x = " << x << " при выходе из с"
<< endl;
}

```

локальная x во внешней области действия main = 5  
 локальная x во внутренней области действия main = 7  
 локальная x во внешней области действия main = 5

локальная переменная x в а = 25 после входа в а  
 локальная переменная x в а = 26 перед выходом из а

локальная статическая переменная x = 50 при входе в б  
 локальная статическая переменная x = 51 при выходе из б

глобальная переменная x = 1 при входе в с  
 глобальная переменная x = 10 при выходе из с

локальная переменная x в а = 25 после входа в а  
 локальная переменная x в а = 26 перед выходом из а

локальная статическая переменная x = 51 при входе в б  
 локальная статическая переменная x = 52 при выходе из б

глобальная переменная x = 10 при входе в с  
 глобальная переменная x = 100 при выходе из с

локальная x в main = 5

Рис. 3.12. Пример областей действия (часть 2 из 2)

## 3.12 Рекурсия

Программы, которые мы обсуждали до сих пор, в общем случае состояли из функций, которые вызывали какие-либо другие функции в строгой иерархической манере. В некоторых случаях полезно иметь функции, которые вызывают сами себя. *Рекурсивная функция* — это функция, которая вызывает сама себя либо непосредственно, либо косвенно с помощью другой функции. Важная тема рекурсии подробно обсуждается в курсах, составляющих вершину компьютерной науки. В этом и последующем разделах представлены простые примеры рекурсии. Вообще в этой книге вопросы рекурсии рассматриваются достаточно широко. Рисунок 3.17 (в конце раздела 3.14) обобщает примеры и упражнения на рекурсию, приведенные в этой книге.

Сначала мы рассмотрим понятие рекурсии, а затем проанализируем несколько программ, содержащих рекурсивные функции. Рекурсивная задача в общем случае разбивается на ряд этапов. Для решения задачи вызывается рекурсивная функция. Эта функция знает, как решать только простейшую часть задачи — так называемую *базовую задачу* (или несколько таких задач). Если эта функция вызывается для решения базовой задачи, она просто возвращает результат. Если функция вызывается для решения более сложной задачи, она делит эту задачу на две части: одну часть, которую функция умеет решать, и другую, которую функция решать не умеет. Чтобы сделать рекурсию выполнимой, последняя часть должна быть похожа на исходную задачу, но быть по сравнению с ней несколько проще или несколько меньше. Поскольку эта новая задача подобна исходной, функция вызывает новую копию самой себя, чтобы начать работать над меньшей проблемой — это называется *рекурсивным вызовом*, или *шагом рекурсии*. Шаг рекурсии включает ключевое слово `return`, так как в дальнейшем его результат будет объединен с той частью задачи, которую функция умеет решать, и сформируется конечный результат, который будет передан обратно в исходное место вызова, возможно, в `main`.

Шаг рекурсии выполняется до тех пор, пока исходное обращение к функции не закрыто, т.е. пока еще не закончено выполнение функции. Шаг рекурсии может приводить к большому числу таких рекурсивных вызовов, поскольку функция продолжает деление каждой новой подзадачи на две части. Чтобы завершить процесс рекурсии, каждый раз, как функции вызывает саму себя с несколько упрощенной версией исходной задачи, должна формироваться последовательность все меньших и меньших задач, в конце концов сходящаяся к базовой задаче. В этот момент функция распознает базовую задачу, возвращает результат предыдущей копии функции и последовательность возвратов повторяет весь путь назад, пока не дойдет до первоначального вызова и не возвратит конечный результат в функцию `main`. Все это звучит довольно экзотично по сравнению с традиционным решением задач, которое мы рассматривали до сих пор. Как пример работы данной концепции, рассмотрим рекурсивную программу для выполнения одного распространенного математического расчета.

Факториал неотрицательного целого числа  $n$ , записываемый как  $n!$ , равен  $n \times (n-1) \times (n-2) \times \dots \times 1$  причем считается, что  $1! = 1$  и  $0! = 1$ . Например,  $5!$  вычисляется как  $5 \times 4 \times 3 \times 2 \times 1$  и равен 120

Факториал целого числа, `number`, большего или равного 0, может быть вычислен *итеративно* (нерекурсивно) с помощью оператора `for` следующим образом:

```
factorial = 1;
for (int counter = number; counter >= 1; counter--)
 factorial *= counter;
```

Рекурсивное определение функции факториал дается следующим соотношением:

$$n! = n \times (n-1)!$$

Например, факториал  $5!$  очевидно равен  $5 \times 4!$ . В самом деле:

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

$$5! = 5 \times (4 \times 3 \times 2 \times 1)$$

$$5! = 5 \times 4!$$

Вычисление  $5!$  должно происходить в соответствии с рис. 3.13. Рис. 3.13а показывает, как протекает последовательность рекурсивных вызовов до тех пор, пока не будет вычислен  $1! = 1$ , что приведет к завершению рекурсии. Рис. 3.13б показывает значения, возвращаемые из каждого рекурсивного вызова оператору вызова до тех пор, пока не будет вычислено и возвращено окончательное значение.

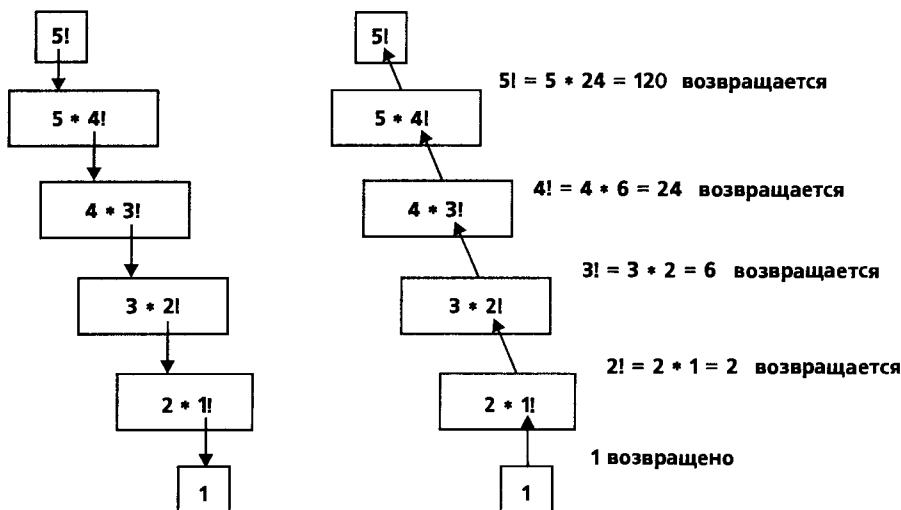


Рис. 3.13. Рекурсивное вычисление  $5!$

Программа на рис. 3.14 использует рекурсию для вычисления и печати факториалов целых чисел от 0 до 10 (выбор типа данных `unsigned long` будет вскоре объяснен). Рекурсивная функция `factorial` сначала проверяет, истинно ли условие завершения рекурсии, т.е. меньше или равно 1 значение `number`. Если действительно `number` меньше или равно 1, `factorial` возвращает 1, никаких дальнейших рекурсий не нужно и программа завершает свою работу. Если `number` больше 1, оператор

```
return number * factorial (number - 1);
```

представляет задачу как произведение `number` и рекурсивного вызова `factorial`, вычисляющего факториал величины `number - 1`. Отметим, что `factorial(number - 1)` является упрощенной задачей по сравнению с исходным вычислением `factorial(number)`.

В объявлении функции `factorial` указано, что она получает параметр типа `unsigned long` и возвращает результат типа `unsigned long`. Это является краткой записью типа `unsigned long int`. Описание языка C++ требует, чтобы переменная типа `unsigned long int` хранилась по крайней мере в 4 байтах (32 битах), и таким образом могла бы содержать значения в диапазоне по крайней мере от 0 до 4294967295. (Тип данных `long int` также хранится по крайней мере в 4 байтах и может содержать значения по крайней мере в диапазоне от 2147483647). Как можно видеть из рис. 3.14, значение факториала растет очень быстро. Мы выбрали тип данных `unsigned long` для того, чтобы программа могла рассчитать числа, большие чем  $7!$ , на компьютерах с маленькими целыми числами (такими, как 2-байтовые). К несчастью, функция `factorial` начинает вырабатывать большие значения так быстро, что даже `unsigned long` не позволяет нам напечатать много значений факториала до того, как будет превышен допустимый предел переменной `unsigned long`.

```
// Рекурсивная функция факториала
#include <iostream.h>
#include <iomanip.h>

unsigned long factorial(unsigned long);

main()
{
 for (int i = 0; i <= 10; i++)
 cout << setw(2) << i << "!" = " << factorial(i) << endl;

 return 0;
}

// Рекурсивное описание функции факториала
unsigned long factorial (unsigned long number)
{
 if (number <= 1)
 return 1;
 else
 return number * factorial(number - 1);
}

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Рис. 3.14. Вычисление факториалов с помощью рекурсивной функции

Как будет показано в упражнениях, пользователь, желающий вычислить факториалы больших чисел, может оказаться вынужденным использовать типы `float` и `double`. Это указывает на слабость большинства языков программирования, а именно, что эти языки нелегко расширить для удовлетворения уникальных требований различных приложений. Как мы увидим в разделе данной книги, посвященном объектно-ориентированному программированию, C++ является расширяемым языком, позволяющим, если мы пожелаем, создавать произвольно большие целые числа.

### Типичная ошибка программирования 3.17

Забывают возвращать значение из рекурсивной функции, когда оно необходимо. Большинство компиляторов вырабатывает при этом предупреждающее сообщение.

### Типичная ошибка программирования 3.18

Пропуск базовой задачи или неправильная запись шага рекурсии, из-за чего процесс не сходится к базовой задаче, приводят к бесконечной рекурсии и существенным затратам памяти. Это аналог бесконечного цикла в итеративном (нерекурсивном) процессе. Бесконечная рекурсия может быть также вызвана вводом неправильной величины.

## 3.13. Пример использования рекурсии: последовательность чисел Фибоначчи

Последовательность чисел Фибоначчи

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

начинается с 0 и 1 и обладает тем свойством, что каждый последующий член последовательности представляет собой сумму двух предыдущих членов. Эта последовательность часто встречается в природе, в частности она описывает форму спирали. Отношение последовательных чисел Фибоначчи сходится к постоянному числу 1,618... Это число также часто встречается в природе и называется золотым сечением. Люди склонны рассматривать золотое сечение как источник эстетического наслаждения. Архитекторы часто проектируют окна, комнаты и здания так, что их длина и ширина находятся в отношении золотого сечения. В таком же отношении часто находятся длина и ширина почтовых ящиков.

Последовательность Фибоначчи можно определить рекурсивно следующим образом:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

В программе, приведенной на рис. 3.15, i-е число Фибоначчи вычисляется рекурсивно с помощью функции `fibonacci`. Заметим, что числа Фибоначчи имеют тенденцию к быстрому росту. Поэтому в функции `fibonacci` мы выбрали тип `unsigned long` и для параметра, и для возвращаемого результата. На рис. 3.15 в примере вывода каждая пара строк соответствует одному прогону программы.

```
// Рекурсивная функция вычисления числа Фибоначчи
#include <iostream.h>
unsigned long fibonacci(unsigned long);

main()
{
 unsigned long result, number;

 cout << "Введите целое число: ";
 cin >> number;
 result = fibonacci(number);
 cout << "Число Фибоначчи(" << number << ") = " << result << endl;
 return 0;
}

// Рекурсивное описание функции fibonacci
unsigned long fibonacci(unsigned long n)
{
 if (n == 0 || n == 1)
 return n;
 else
 return fibonacci(n - 1) + fibonacci(n - 2);
}
```

---

Введите целое число: 0  
Число Фибоначчи (0) = 0

Введите целое число: 1  
Число Фибоначчи (1) = 1

Введите целое число: 2  
Число Фибоначчи (2) = 1

Введите целое число: 3  
Число Фибоначчи (3) = 2

Введите целое число: 4  
Число Фибоначчи (4) = 3

Введите целое число: 5  
Число Фибоначчи (5) = 5

Введите целое число: 6  
Число Фибоначчи (6) = 8

Введите целое число: 10  
Число Фибоначчи (10) = 55

Введите целое число: 20  
Число Фибоначчи (20) = 6765

Введите целое число: 30  
Число Фибоначчи (30) = 832040

Введите целое число: 35  
Число Фибоначчи (35) = 9227465

Рис. 3.15. Рекурсивная генерация чисел Фибоначчи

Вызов `fibonacci` из `main` не рекурсивный, но все последующие вызовы `fibonacci` рекурсивны. При каждом вызове `fibonacci` она сначала проверяет, не является ли задача базовой, для которой  $n$  равно 0 или 1. Если да, то возвращается  $n$ . Интересно, что при  $n$  большем 1, шаг рекурсии генерирует два рекурсивных вызова, каждый из которых представляет собой несколько упрощенную задачу по сравнению с исходным вызовом `fibonacci`. На рис. 3.16 показано, как функция `fibonacci` вычисляет `fibonacci(3)` — мы обозначаем `fibonacci` просто как `f`, чтобы рисунок легче читался.

Этот рисунок ставит некоторые интересные вопросы о последовательности, в которой компилятор C++ будет вычислять операнды заданных операций. Это отличается от вопроса о последовательности выполнения операций, диктуемой правилами старшинства операций. Из рис. 3.16 видно, что для вычисления  $f(3)$  нужно выполнить два рекурсивных вызова, а именно,  $f(2)$  и  $f(1)$ . Но в каком порядке должны быть сделаны эти вызовы?

Большинство программистов просто полагает, что операнды будут вычисляться слева направо. Странно, но C++ не оговаривает порядок, в котором должны вычисляться операнды большинства операций, включая `+`. Поэтому программист не может делать каких-либо предположений о последовательности, в которой будут выполняться эти вызовы. Эти вызовы в действительности могут выполняться в любом порядке: сначала  $f(2)$ , а потом  $f(1)$ , или наоборот — сначала  $f(1)$ , а потом  $f(2)$ . В этой и большинстве других программ результат не зависит от последовательности вычислений. Но в некоторых программах вычисления операндов могут иметь побочный эффект, который может повлиять на окончательный результат выражения. Язык C++ определяет порядок вычисления операндов только для четырех операций — а именно, `&&`, `||`, последования `(, )` и `?:`. Для первых трех бинарных операций операнды гарантированно вычисляются слева направо. Последняя операция — единственная тернарная операция в C++. Ее самый левый operand всегда выполняется первым; если результат его вычисления отличен от нуля, то следующим вычисляется средний operand, а последний operand игнорируется; если же результат вычисления самого левого operandа равен нулю, то следующим вычисляется третий operand, а средний operand игнорируется.

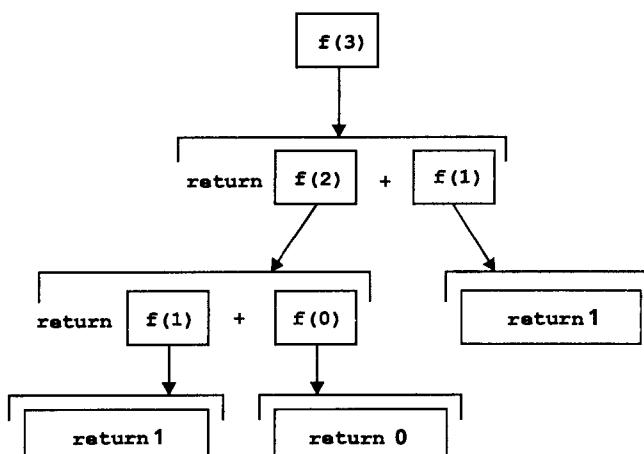


Рис. 3.16. Последовательность рекурсивных вызовов функции `fibonacci`

### Типичная ошибка программирования 3.19

Написание программ, которые зависят от последовательности вычисления операндов каких-то операций, отличных от **&&**, **||**, последования **( , )** и **?:**, может привести к ошибкам, потому что компиляторы могут вычислять операнды не в той последовательности, которую ожидает программист.

### Замечание по мобильности 3.2

Программы, которые зависят от последовательности вычисления операндов операций, отличных от **&&**, **||**, последования **( , )** и **?:**, могут по-разному работать в системах с различными компиляторами.

Нужно учитывать одну особенность, характерную для рекурсивных программ, подобных той, которую мы использовали для генерации чисел Фибоначчи. Каждый уровень рекурсии в функции **fibonacci** удваивает количество вызовов, так что количество рекурсивных вызовов, которое должно быть выполнено для вычисления  $n$ -го числа Фибоначчи, оказывается порядка  $2^n$ . Объем вычислений резко нарастает с увеличением  $n$ . Вычисление только 20-го числа Фибоначчи потребовало бы порядка  $2^{20}$  или около миллиона вызовов, вычисление 30-го числа Фибоначчи потребовало бы порядка  $2^{30}$  или около миллиарда вызовов и так далее. В методах вычислений это называется **экспоненциальной сложностью**. Проблемы такого рода не по плечу даже самым мощным компьютерам в мире! Вопросы сложности в целом и экспоненциальной сложности в частности детально рассматриваются на верхнем уровне обучения в курсах, обычно называемых «Алгоритмы».

### Совет по повышению эффективности 3.4

Избегайте рекурсивных программ, подобных программе для вычисления чисел Фибоначчи, которые приводят к экспоненциальному нарастанию количества вызовов.

## 3.14. Рекурсии или итерации

В предыдущих разделах мы рассмотрели две функции, которые можно легко реализовать рекурсивно или итеративно. В этом разделе мы сравним эти два подхода и обсудим основания, по которым программист может предпочесть в конкретной ситуации то или иной подход.

Как итерации, так и рекурсии основаны на управляющей структуре: итерации используют структуру повторения, рекурсии используют структуру выбора.

Как итерации, так и рекурсии включают повторение: итерации используют структуру повторения явным образом, рекурсии реализуют повторение посредством повторных вызовов функции. Как итерации, так и рекурсии включают проверку условия окончания: итерации заканчиваются после нарушения условия продолжения цикла, рекурсии заканчиваются после распознавания базовой задачи. Как итерации с повторением, управляемым счетчиком, так и рекурсии постепенно приближаются к моменту своего завершения: в итерациях выполняется изменение счетчика до тех пор, пока он не примет значение, при котором перестает выполняться условие продолжения цикла; в рекурсиях поддерживается процесс создания упрощенной

версии исходной задачи до тех пор, пока не будут достигнута базовая задача. Как итерации, так и рекурсии могут оказаться бесконечными: бесконечный итеративный цикл возникает, если условие продолжения цикла никогда не становится ложным; бесконечная рекурсия возникает, когда шаг рекурсии не упрощает исходную задачу таким образом, чтобы она сходилась к базовой.

Рекурсия имеет много недостатков. Повторный запуск рекурсивного механизма вызовов функции приводит к росту накладных расходов: к нарастающим затратам процессорного времени или требуемого объема памяти. Каждый рекурсивный вызов приводит к созданию новой копии функции (на самом деле копируются только переменные данной функции); для этого может потребоваться значительная память. Итерации обычно не связаны с функциями, так что в них отсутствуют накладные расходы на повторные вызовы функции и дополнительные затраты памяти. Тогда для чего же применять рекурсию?

### **Замечание по технике программирования 3.13**

Любые задачи, которые можно решить рекурсивно, могут быть решены также и итеративно (нерекурсивно). Обычно рекурсивный подход предпочитают итеративному, если он более естественно отражает задачу и ее результаты, то есть более нагляден и легче отлаживается. Другая причина предпочтения рекурсивного решения состоит в том, что итеративное решение может не быть очевидным.

### **Совет по повышению эффективности 3.5**

Избегайте использования рекурсий в случаях, когда требуется высокая эффективность. Рекурсивные вызовы требуют времени и дополнительных затрат памяти.

### **Типичная ошибка программирования 3.20**

Случайный вызов нерекурсивной функции самой себя либо непосредственно, либо косвенно, через другую функцию.

Большинство учебников по программированию знакомят с рекурсией гораздо позже, чем это сделано в этой книге. Мы считаем, что рекурсия — весьма обширная и сложная тема, так что лучше познакомиться с ней раньше и распределить ее примеры по всему остальному тексту книги. На рис. 3.17 приведена таблица, содержащая список примеров и упражнений по рекурсии в данной книге.

Давайте посмотрим еще раз на многочисленные советы, которые мы даем в этой книге. Хорошая техника программирования важна. Часто важна и высокая производительность. К несчастью, эти цели часто не совместимы друг с другом. Хорошая техника программирования — это ключевой момент в вопросе управления созданием все более сложных и крупных программных систем. Высокая производительность этих систем — ключ к созданию систем будущего, которые предъявят еще большие требования к аппаратным средствам. Что же важнее?

### **Замечание по технике программирования 3.14**

Функционализация программ в четком иерархическом стиле — свидетельство хорошей техники программирования. Но за все надо платить.

| Глава   | Примеры и упражнения по рекурсии                                                                                                                                                                                                                                                                                  |
|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Глава 3 | Функция факториала<br>Функции Фибоначчи<br>Наибольший общий делитель<br>Сумма двух целых чисел<br>Перемножение двух целых чисел<br>Возведение целого числа в целую степень<br>Ханойская башня<br>Рекурсивная <b>main</b><br>Печать информации с клавиатуры в обратной последовательности<br>Визуализация рекурсии |
| Глава 4 | Суммирование элементов массива<br>Печать массива<br>Печать массива в обратной последовательности<br>Печать строки в обратной последовательности<br>Проверка, является ли строка палиндромом<br>Минимальное значение в массиве<br>Сортировка отбором<br>Восемь ферзей<br>Линейный поиск<br>Бинарный поиск          |
| Глава 5 | Быстрая сортировка<br>Обход лабиринта<br>Печать вводимой с клавиатуры строки в обратной последовательности                                                                                                                                                                                                        |
| Глава 6 | Вставка связного списка<br>Удаление связного списка<br>Поиск в связном списке<br>Печать связного списка в обратной последовательности<br>Вставка в бинарное дерево<br>Обход бинарного дерева в ширину<br>Последовательный обход бинарного дерева<br>Обход бинарного дерева в глубину                              |

Рис. 3.17. Список примеров и упражнений по рекурсии в тексте книги

### Совет по повышению эффективности 3.6

Программа, разбитая на множество функций, потенциально требует большего количества вызовов функций по сравнению с монолитной программой без функций. Это увеличивает время выполнения и затраты памяти компьютера. Но монолитные программы трудно программировать, тестировать, отлаживать, сопровождать и развивать.

Так что функционализируйте ваши программы с учетом здравого смысла, всегда учитывая хрупкий баланс между производительностью и хорошей техникой программирования.

## 3.15. Функции с пустыми списками параметров

В C++ пустой список параметров определяют, либо записывая **void**, либо совсем ничего не записывая в круглых скобках. Объявление

```
void print();
```

указывает, что функция **print** не требует никаких аргументов и не возвращает значений. Рис. 3.18 демонстрирует оба используемых в C++ способа объявления и применения функций, не требующих аргументов.

```
// Функции, не требующие аргументов
#include <iostream.h>

void f1();
void f2(void);

main()
{
 f1();
 f2();

 return 0;
}

void f1()
{
 cout << "Функция f1 не требует аргументов" << endl;
}
void f2(void)
{
 cout << "Функция f2 также не требует аргументов" << endl;
}
```

---

**Функция f1 не требует аргументов**  
**Функция f2 также не требует аргументов**

**Рис. 3.18.** Два способа объявления и использования функций, не требующих аргументов

### Замечание по мобильности 3.3

Роль пустого списка параметров функции в C++ существенно отличается от С. В С это означает, что все проверки аргументов отсутствуют (т.е. вызов функции может передать любой аргумент, который требуется). А в C++ пустой список означает отсутствие аргументов. Таким образом, программа на С, использующая эту особенность, может сообщить о синтаксической ошибке при компиляции в C++.

Теперь, когда мы обсуждаем, что можно опустить при записи программы, необходимо отметить, что функция, определенная в файле до любого обращения к ней, не требует отдельного прототипа функции. В этом случае в качестве прототипа выступает заголовок функции.

### Типичная ошибка программирования 3.21

Программы на C++ не компилируются, если для каждой функции нет соответствующего ей прототипа или каждая функция не определяется до того, как она используется.

## 3.16. Встраиваемые функции

Реализация программы как набора функций хороша с точки зрения разработки программного обеспечения, но вызовы функций, как говорилось, приводят к накладным расходам во время выполнения. В C++ для снижения

этих накладных расходов на вызовы функций — особенно небольших функций — предусмотрены *встраиваемые (inline)* функции. Спецификация **inline** перед указанием типа результата в объявлении функции «советует» компилятору сгенерировать копию кода функции в соответствующем месте, чтобы избежать вызова этой функции. В результате получается множество копий кода функции, вставленных в программу, вместо единственной копии, которой передается управление при каждом вызове функции. Компилятор может игнорировать спецификацию **inline** и обычно так и делает для всех функций, кроме самых малых.

### **Замечание по технике программирования 3.15**

Любые изменения функции **inline** могут потребовать перекомпиляции всех «потребителей» этой функции. Это может оказаться существенным моментом для развития и поддержки некоторых программ.

### **Хороший стиль программирования 3.13**

Спецификацию **inline** целесообразно применять только для небольших и часто используемых функций.

### **Совет по повышению эффективности 3.7**

Использование функций **inline** может уменьшить время выполнения программы, но может увеличить ее размер.

Программа на рис. 3.19 использует встроенную функцию **cube** для расчета величины куба со стороной **s**. Ключевое слово **const** в списке параметров функции **cube** говорит компилятору о том, что функция не изменяет параметр **s**. Ключевое слово **const** подробно обсуждается в главах 4, 5 и 7.

```
//Использование встраиваемой функции для расчета объема куба
#include <iostream.h>

inline float cube(const float s) { return s * s * s; }

main()
{
 cout << "Введите длину стороны вашего куба: ";
 float side;
 cin >> side;
 cout << "Объем куба со стороной "
 << side << " равен " << cube(side) << endl;

 return 0;
}
```

---

Введите длину стороны вашего куба: 3.5  
Объем куба со стороной 3.5 равен 42.875

**Рис. 3.19.** Использование функции **inline** для расчета объема куба

### 3.17. Ссылки и ссылочные параметры

Во многих языках программирования имеются два способа обращения к функциям — *вызов по значению* и *вызов по ссылке*. Когда аргумент передается вызовом по значению, создается копия аргумента и она передается вызываемой функции. Изменения копии не влияют на значение оригинала в операторе вызова. Это предотвращает случайный побочный эффект, который так сильно мешает развитию корректного и надежного программного обеспечения. Все аргументы, которые передавались в функции в программах этой главы, передавались вызовом по значению. Один из недостатков вызова по значению состоит в том, что если передается большой элемент данных, создание копии этих данных может привести к значительным потерям времени выполнения.

В этом разделе мы познакомимся со *ссылочными параметрами* — первым из двух средств, которыми снабжен C++ для вызова по ссылке. В случае вызова по ссылке оператор вызова дает вызываемой функции возможность прямого доступа к передаваемым данным, а также возможность изменения этих данных. Вызов по ссылке хорош в смысле производительности, потому что он исключает накладные расходы на копирование больших объемов данных, но вызов по ссылке может ослабить защищенность, потому что вызываемая функция может испортить передаваемые в нее данные.

Ссылочный параметр — это псевдоним соответствующего аргумента. Чтобы показать, что параметр функции передан по ссылке, после типа параметра в прототипе функции ставится символ амперсанда (&); такое же обозначение используется в списке типов параметров в заголовке функции. Например, объявление

```
int &count
```

в заголовке функции может читаться как «`count` является ссылкой на `int`». В вызове функции достаточно указать имя переменной и она будет передана по ссылке. Тогда упоминание в теле вызываемой функции переменной по имени ее параметра в действительности является обращением к исходной переменной в вызывающей функции и эта исходная переменная может быть изменена непосредственно вызываемой функцией.

Программа на рис. 3.20 сравнивает вызовы по значению и по ссылке с использованием ссылочных параметров. «Стили» аргументов в обращениях к `squareByValue` и `squareByReference` идентичны, т.е. обе переменных просто упоминаются по имени. Без проверки прототипов или описаний функций невозможно выяснить, вызовы какой из этих двух функций могут изменять ее аргументы.

#### Типичная ошибка программирования 3.22

Поскольку ссылочные параметры упоминаются в теле вызываемой функции только по имени, программист может нечаянно принять ссылочные параметры за параметры, передаваемые по значению. Это может привести к неожиданным эффектам, если исходные копии переменных изменяются вызывающей функцией.

В главе 5 мы будем рассматривать указатели; мы увидим, что указатели можно использовать как альтернативную форму вызова по ссылке, в которой способ вызова ясно указывает на передачу по ссылке (и на возможное изменение аргументов в операторе вызова).

```
// Сравнение вызовов по значению и по ссылке

#include <iostream.h>

int squareByValue(int);
void squareByReference(int &);

main ()
{
 int x = 2, z = 4;

 cout << "x = " << x << " перед squareByValue" << endl
 << "Значение, возвращенное squareByValue: "
 << squareByValue(x) << endl
 << "x = " << x << " после squareByValue" << endl << endl;

 cout << "z = " << z << " перед squareByReference" << endl;
 squareByReference(z);
 cout << "z = " << z << " после squareByReference" << endl;

 return 0;
}

int squareByValue(int a)
{
 return a *= a; // аргумент оператора вызова не изменяется
}

void squareByReference(int &cRef)
{
 cRef *= cRef; // аргумент оператора вызова изменяется
}

x = 2 перед squareByValue
Значение, возвращенное squareByValue: 4
x = 2 после squareByValue

z = 4 перед squareByReference
z = 16 после squareByReference
```

Рис. 3.20. Пример вызова по ссылке

### Совет по повышению эффективности 3.8

Для передачи больших объектов используйте константный ссылочный параметр, чтобы обеспечить защиту параметров, как при вызове по значению, и в то же время избежать накладных расходов при передаче копии большого объекта.

Чтобы определить ссылку как константу, разместите перед описателем типа в объявлении параметра спецификацию **const**.

Обратите внимание на место символа **&** в списке параметров функции **squareByReference**. Некоторые программисты на C++ предпочитают записывать **int& cRef** вместо **int &cRef**.

### Типичная ошибка программирования 3.23

При объявлении множества ссылок в одном операторе делается неверное предположение, что действие символа амперсанд **&** распространяется на весь записанный через запятую список имен переменных. Чтобы объявить переменные **x**, **y** и **z** как ссылки на целое число, используйте запись **int &x, &y, &z;** вместо неправильной записи **int& x, y, z;** или другой распространенной неправильной записи **int &x, y, z;**

### Замечание по технике программирования 3.16

Чтобы обеспечить одновременно ясность программы и ее производительность, многие программисты на C++ предпочитают, чтобы аргументы, которые могут изменяться, передавались бы функциям с помощью указателей, аргументы, не предназначенные для изменения, передавались бы вызовом по значению, а большие неизменяемые аргументы передавались бы функциям путем использования константных ссылок.

Ссылки можно также использовать как псевдонимы для других переменных внутри функций (хотя для этого и мало оснований). Например, фрагмент

```
int count = 1; // объявление целой переменной count
int &cRef = count; // создание cRef как псевдонима для count
++cRef; // приращение count (используется псевдоним)
```

дает приращение переменной **count**, используя ее псевдоним **cRef**. Ссылочные переменные должны получать начальные условия в их объявлениях (смотри рис. 3.21 и рис. 3.22) и не могут переприсваиваться как псевдонимы другим переменным. Как только ссылка объявляется как псевдоним другой переменной, все операции, выполняемые с псевдонимом (т.е. ссылкой), на самом деле будут выполняться с самой истинной переменной. Псевдоним — лишь другое имя для истинной переменной, для него не резервируется никакого места в памяти. Определение адреса ссылки и сравнение ссылок не вызывают синтаксических ошибок; на самом деле каждая операция выполняется над переменной, для которой ссылка является псевдонимом. Аргумент-ссылка должен быть L-величиной, не константой и не выражением, которое возвращает R-величину.

```
// Ссылка должна получить начальное значение
#include <iostream.h>

main()
{
 int x = 3, &y; //Ошибка: y должна получить начальное значение
 cout << "x = " << x << endl << "y = " << y << endl;
 y = 7;
 cout << "x = " << x << endl << "y = " << y << endl;

 return 0;
}
```

Compiling FIG3\_21.CPP:

Error FIG3\_21.CPP 6: Reference variable 'y' must be initialized

Рис. 3.21. Попытка использовать ссылку без заданного начального значения

```
// Ссылка должна получить начальное значение
#include <iostream.h>

main()
{
 int x = 3, &y = x; // y теперь является псевдонимом x

 cout << "x = " << x << endl << "y = " << y << endl;
 y = 7;
 cout << "x = " << x << endl << "y = " << y << endl;

 return 0;
}
```

```
x = 3
y = 3
x = 7
y = 7
```

Рис. 3.22. Использование ссылки с заданным начальным значением

Функции могут возвращать ссылки, но здесь надо быть осторожным. Если возвращение ссылки переменной объявлено в вызываемой функции, переменная должна быть объявлена внутри этой функции как `static`. В противном случае ссылка адресуется автоматической переменной, которая после завершения функции уничтожается; в таком случае говорят, что переменная «не определена» и поведение программы непредсказуемо.

#### Типичная ошибка программирования 3.24

В объявлении ссылочной переменной ей не присваивается начальное значение.

#### Типичная ошибка программирования 3.25

Попытка переприсвоить предварительно объявленную ссылку как псевдоним другой переменной.

#### Типичная ошибка программирования 3.26

Возвращение указателя или ссылки автоматической переменной в вызываемой функции.

## 3.18. Аргументы по умолчанию

Обычно при вызове функций в нее передается конкретное значение каждого аргумента. Но программист может указать, что аргумент является *аргументом по умолчанию* и присвоить этому аргументу значение по умолчанию. Если аргумент по умолчанию не указан в вызове функции, то в вызов автоматически передается значение этого аргумента по умолчанию. Аргументы по умолчанию должны быть самыми правыми (последними) аргументами в списке параметров функции. Если вызывается функция с двумя или более аргументами по умолчанию и если пропущенный аргумент не яв-

ляется самым правым в списке аргументов, то все аргументы справа от пропущенного тоже пропускаются. Аргументы по умолчанию должны быть указаны при первом упоминании имени функции — обычно в прототипе. Значения по умолчанию могут быть константами, глобальными переменными или вызовами функций. Аргументы по умолчанию можно использовать также с функциями `inline`.

Рисунок 3.23 демонстрирует использование аргументов по умолчанию при вычислении значения объема параллелепипеда.

```
//Использование аргументов по умолчанию

#include <iostream.h>

// Расчет объема параллелепипеда
inline int boxVolume(int length = 1, int width = 1,
 int height = 1)
{ return length * width * height; }

main()
{
 cout << " Объем параллелепипеда по умолчанию равен: "
 << boxVolume() << endl << endl
 << " Объем параллелепипеда с длиной 10, "
 << endl << endl
 << " шириной 1 и высотой 1 равен: "
 << boxVolume(10) << endl << endl
 << " Объем параллелепипеда с длиной 10, "
 << endl << endl
 << " шириной 5 и высотой 1 равен: "
 << boxVolume(10, 5) << endl << endl
 << " Объем параллелепипеда с длиной 10, "
 << endl << endl
 << " шириной 5 и высотой 2 равен: "
 << boxVolume(10, 5, 2)
 << endl;

 return 0;
}
```

---

```
Объем параллелепипеда по умолчанию равен: 1
Объем параллелепипеда с длиной 10,
шириной 1 и высотой 1 равен: 10
Объем параллелепипеда с длиной 10,
шириной 5 и высотой 1 равен: 50
Объем параллелепипеда с длиной 10,
шириной 5 и высотой 2 равен: 100
```

**Рис. 3.23.** Использование аргументов по умолчанию

Всем трем аргументам было дано значение по умолчанию 1. В первом вызове встроенной функции `boxVolume` не указаны аргументы, поэтому используются все три значения по умолчанию. Во втором вызове передается аргумент `length`, поэтому используются значения по умолчанию аргументов `width` и `height`. В третьем вызове передаются аргументы `width` и `height`, поэтому используется значение по умолчанию аргумента `height`. В последнем вызове передаются аргументы `length`, `width` и `height`, поэтому значения по умолчанию не используются ни для одного аргумента.

### Хороший стиль программирования 3.14

Использование аргументов по умолчанию может упростить написание вызовов функций. Однако некоторые программисты считают, что для ясности лучше явно указывать все аргументы.

### Типичная ошибка программирования 3.27

Определение и попытка использовать аргумент по умолчанию, не являющийся самым правым (последним) аргументом (если одновременно не являются аргументами по умолчанию все более правые аргументы).

## 3.19. Унарная операция разрешения области действия

Локальную и глобальную переменные можно объявлять одним и тем же именем. В C++ имеется *унарная операция разрешения области действия*(`::`), которая дает доступ к глобальной переменной, даже если под тем же именем в области действия объявлена локальная переменная. Унарная операция разрешения области действия не может быть использована, чтобы получить доступ к локальной переменной, объявленной с тем же именем во внешнем блоке. Доступ к глобальной переменной имеется и непосредственно без унарной операции разрешения области действия, если имя глобальной переменной не совпадает с именем локальной переменной в области действия. В главе 6 мы обсудим использование *бинарной операции разрешения области действия* для классов.

Рисунок 3.24 демонстрирует применение унарной операции разрешения области действия при наличии локальной и глобальной переменных, имеющих одинаковое имя. Чтобы подчеркнуть отличие локальной и глобальной версий переменной `value`, программа объявляет одну из этих переменных как `float`, а другую как `int`.

```
// Использование унарной операции разрешения области действия.
#include <iostream.h>

float value = 1.2345;

main()
{
 int value = 7;

 cout << "Локальное значение = " << value << endl
 << "Глобальное значение = " << ::value << endl;

 return 0;
}
```

---

Локальное значение = 7  
Глобальное значение = 1.2345

Рис. 3.24. Использование унарной операции разрешения области действия

### Типичная ошибка программирования 3.28

Попытка получить доступ к неглобальной переменной внешнего блока, используя унарную операцию разрешения области действия.

### Хороший стиль программирования 3.15

Избегайте использования в программе переменных с одинаковыми именами для разных целей. Хотя это и допускается в различных случаях, но может приводить к путанице.

## 3.20. Перегрузка функций

C++ позволяет определить несколько функций с одним и тем же именем, если эти функции имеют разные наборы параметров (по меньшей мере разные типы параметров). Эта особенность называется *перегрузкой функции*. При вызове перегруженной функции компилятор C++ определяет соответствующую функцию путем анализа количества, типов и порядка следования аргументов в вызове. Перегрузка функции обычно используется для создания нескольких функций с одинаковым именем, предназначенных для выполнения сходных задач, но с разными типами данных.

На рисунке 3.25 перегруженная функция `square` используется для расчета квадрата переменной типа `int` и квадрата переменной типа `double`. В главе 8 мы обсудим, как осуществлять перегрузку операций, чтобы они работали с объектами, имеющими тип данных, определенный пользователем. (В действительности, мы уже использовали к настоящему моменту перегруженные операции, включая операцию поместить в поток `<<` и операцию взять из потока `>>`. Мы еще обсудим перегрузку операций `<<` и `>>` в главе 8.) Раздел 3.21 познакомит нас с шаблонами функций, используемыми для выполнения идентичных задач с разными типами данных. В главе 12 шаблоны функций и шаблоны классов обсуждаются более подробно.

```
// Использование перегруженных функций
#include <iostream.h>

int square(int x) { return x * x; }

double square(double y) { return y * y; }

main()
{
 cout << "Квадрат целого числа 7 равен "
 << square(7) << endl
 << "Квадрат числа 7.5 типа double равен "
 << square(7.5) << endl;

 return 0;
}
```

---

Квадрат целого числа 7 равен 49  
 Квадрат числа 7.5 типа double равен 56.25

Рис. 3.25. Использование перегруженных функций

### Хороший стиль программирования 3.16

Перегруженные функции, которые выполняют тесно связанные задачи, делаю программы более понятными и легко читаемыми.

Перегруженные функции различаются с помощью их *сигнатуры* — комбинации имени функции и типов ее параметров. Компилятор кодирует идентификатор каждой функции по числу и типу ее параметров (иногда это называется *декорированием имени*), чтобы иметь возможность осуществлять надежное связывание типов. Надежное связывание типов гарантирует, что вызывается надлежащая функция и что аргументы согласуются с параметрами. Компилятор выявляет ошибки связывания и выдает сообщения о них. Программа на рис. 3.26 была скомпилирована компилятором Borland C++. Вместо показа результатов работы программы (как мы обычно делали раньше), мы привели на рис. 3.26 декорированные имена функций на языке ассемблер, полученные с помощью Borland C++. Каждое декорированное имя начинается с символа @, предшествующего имени функции. Закодированный список параметров начинается с символов \$q. В списке параметров для функции *nothing2* zc соответствует типу *char*, i соответствует типу *int*, pf соответствует типу *float \** и pd соответствует типу *double \**. В списке параметров для функции *nothing1*, i соответствует типу *int*, f соответствует типу *float \**, zc соответствует типу *char* и pi соответствует типу *int \**. Две эти функции *square* отличаются своими списками параметров; одна определяет d как *double*, а другая определяет i как *int*. Типы возвращаемых значений функций не отражаются в декорированных именах. Декорирование имен функций — специфика компилятора. Перегруженные функции могут иметь и различные типы возвращаемых значений, но обязательно должны иметь различные списки параметров.

```
// Декорирование имен
int square(int x) { return x * x; }

double square(double y) { return y * y; }

void nothing(int a, float b, char c, int *d)
{ } // пустое тело функции

char *nothing2(char a, int b, float *c, double *d)
{ return 0; }

main()
{
 return 0;
}

public _main
public @_nothing2$qzciplfpd
public @_nothing1$qifzcpf
public @_square$qd
public @_square$qi
```

Рис. 3.26. Декорирование имен для обеспечения надежного согласования типов

### **Типичная ошибка программирования 3.29**

Создание перегруженных функций с идентичными списками параметров и различными типами возвращаемых значений приводит к синтаксической ошибке.

Для различения функций с одинаковыми именами компилятор использует только списки параметров. Перегруженные функции не обязательно должны иметь одинаковое количество параметров. Программисты должны быть осторожными, имея дело с перегруженными функциями с параметрами по умолчанию, поскольку это может стать причиной неопределенности.

### **Типичная ошибка программирования 3.30**

Функция с пропущенными аргументами по умолчанию может оказаться вызванной аналогично другой перегруженной функции; это синтаксическая ошибка.

## **3.21. Шаблоны функции**

Перегруженные функции обычно используются для выполнения сходных операций над различными типами данных. Если операции идентичны для каждого типа, это можно выполнить более компактно и удобно, используя *шаблоны функций* — свойство, введенное в последней версии C++. Программист пишет единственное определение шаблона функции. Основываясь на типах аргументов, указанных в вызовах этой функции, C++ автоматически генерирует разные функции для соответствующей обработки каждого типа. Таким образом, определение единственного шаблона определяет целое семейство решений.

Все определения шаблона функции начинаются с ключевого слова *template*, за которым следует список формальных типов параметров функции, заключенный в угловые скобки (< и >). Каждый формальный тип параметра предваряется ключевым словом *class*. Формальные типы параметров — это встроенные типы или типы, определяемые пользователем. Они используются для задания типов аргументов функции, для задания типов возвращаемого значения функции и для объявления переменных внутри тела описания функции. После шаблона следует обычное описание функции.

Следующее определение шаблона функции используется в программе, приведенной на рис. 3.27.

```
template <class T>
T maximum(T value1, T value2, T value3)
{
 T max = value1;

 if (value2 > max)
 max = value2;

 if (value3 > max)
 max = value3;

 return max;
}
```

Этот шаблон функции объявляет единственный формальный параметр *T* как тип данных, который должен проверяться функцией *maximum*. Когда

компилятор обнаруживает вызов **maximum** в исходном коде программы, этот тип данных, переданных в **maximum**, подставляется вместо T всюду в определении шаблона и C++ создает законченную функцию для определения максимального из трех значений указанного типа данных. Затем заново созданная функция компилируется. Таким образом, шаблоны в действительности играют роль средств генерации кода. В программе на рис. 3.27 обрабатываются три функции — одна ожидает три значения **int**, вторая ожидает три значения **double** и третья ожидает три значения **char**. Обработка для типа **int** имеет вид:

```
int maximum(int value1, int value2, int value3)
{
 int max = value1;

 if (value2 > max)
 max = value2;

 if (value3 > max)
 max = value3;

 return max;
}
```

Каждый формальный параметр в определении шаблона должен хотя бы однажды появиться в списке параметров функции. Каждое имя формального параметра в списке определения шаблона должно быть уникальным.

Программа на рис. 3.27 иллюстрирует применение шаблона функции **maximum** для поиска максимального из трех целых чисел, трех чисел с плавающей запятой и трех символов.

### Типичная ошибка программирования 3.31

Отсутствие ключевого слова **class** перед каждым формальным параметром шаблона функции.

### Типичная ошибка программирования 3.32

В сигнатуре функции не используются все формальные параметры шаблона функции.

```
// Использование шаблонов функций
#include <iostream.h>

template <class T>
T maximum(T value1, T value2, T value3)
{
 T max = value1;

 if (value2 > max)
 max = value2;

 if (value3 > max)
 max = value3;

 return max;
}
```

Рис. 3.27. Использование шаблонов функций (часть 1 из 2)

```

main()
{
 int int1, int2, int3;
 cout << "Введите три целых значения: ";
 cin >> int1 >> int2 >> int3;
 cout << "Максимальное целое значение равно: "
 << maximum(int1, int2, int3); // версия int

 double double1, double2, double3;
 cout << endl << "Введите три значения double: ";
 cin >> double1 >> double2 >> double3;
 cout << "Максимальное значение double равно: "
 << maximum(double1, double2, double3); // версия double

 char char1, char2, char3;
 cout << endl << "Введите три символа: ";
 cin >> char1 >> char2 >> char3;
 cout << "Максимальное значение символа равно: "
 << maximum(char1, char2, char3) << endl; // версия char

 return 0;
}

```

---

```

Введите три целых значения: 1 2 3
Максимальное целое значение равно: 3
Введите три значения double: 3.3 2.2 1.1
Максимальное значение double равно: 3.3
Введите три символа: А В С
Максимальное значение символа равно: С

```

**Рис. 3.27.** Использование шаблонов функций (часть 2 из 2)

## 3.22. Размышления об объектах: идентификация атрибутов объектов

В разделе «Размышления об объектах» в конце главы 2 мы начали первый этап объектно-ориентированного проектирования программы, моделирующей лифт, а именно — идентификацию объектов, необходимых для разработки модели. В качестве отправной точки вам было предложено составить список имен существительных, использованных в постановке задачи. Выполняя это задание, вы обнаружили, что в число объектов вашей модели должны войти сам лифт, люди, здание, различные кнопки, часы, лампочки и звонки на каждом этаже и т.д.

В нашем введении в объекты в главе 1 мы указали, что объекты имеют атрибуты и способы поведения. Атрибуты объекта представляются в программе на C++ данными; варианты поведения объекта представляется функциями. Сейчас мы сконцентрируем внимание на атрибутах объектов, необходимых для разработки модели лифта. В главе 4 мы перейдем к поведению. В главе 5 мы сосредоточимся на взаимодействии между объектами в модели лифта.

Перед началом выполнения нашего задания давайте обсудим атрибуты объектов реального мира. Атрибуты человека включают его рост и вес. Атрибуты радиоприемника включают частотный диапазон, вид модуляции — амплитудную или частотную, напряжение сети. Атрибуты автомобиля включают текущие показания спидометра и счетчика пройденного расстояния. Атрибуты дома включают стиль («колониальный», «ранчо» и т.д.), количество комнат, площадь, размер участка. Атрибуты персонального компьютера включают фирму-изготовителя (Apple, IBM, Сони и т.д.), тип экрана (монохромный или цветной), объем основной памяти (в мегабайтах), объем дисковой памяти (в мегабайтах) и т.д.

### Лабораторное задание 2 по лифту

1. Начните с того, что напечатайте в любом текстовом редакторе текст постановки задачи моделирования лифта (из раздела 2.22).
2. Извлеките из постановки задачи все факты. Исключите весь не относящийся к сути дела текст и поместите каждый факт в отдельной строке вашего текстового файла (в постановке задачи содержится около 60 фактов). Первая группа фактов в файле может выглядеть примерно следующим образом:

#### *Файл фактов*

двухэтажное здание офиса  
лифт  
пассажир лифта  
двери  
этаж 1  
направления — вверх и вниз  
часы  
время 0  
отсчет времени с тактом, соответствующим 1 секунде  
компонент «планировщик» программы  
случайное планирование появления первого пассажира на любом этаже  
время первого появления  
моделирующая программа  
«создание» нового пассажира для указанного этажа  
размещение пассажира на этом этаже  
пассажир нажимает на этаже кнопку «вверх» или «вниз»  
кнопка «вверх» или «вниз»  
этаж, нужный пассажиру  
этаж, на котором появился пассажир  
первый пассажир каждый день появляется на этаже 1  
пассажир появляется на этаже 1  
пассажир входит в лифт  
пассажир нажимает кнопку «вверх»

3. Сгруппируйте все ваши факты по объектам. Это поможет убедиться в том, что вы соответствующим образом идентифицировали объекты в лабораторном задании главы 2. Используйте табличную форму, в которой объекты перечисляются с левого края страницы, а факты, относящиеся к объектам, перечисляются ниже этих объектов со сдвигом на одну табуляцию. Некоторые факты упоминают только один объект,

тогда как другие — несколько объектов. Каждый факт должен вначале быть перечислен под каждым объектом, к которому он относится. Отметим, что некоторые факты типа «направления — вверх и вниз» не упоминают объекты явно, но тем не менее должны быть сгруппированы с объектами (в данном случае очевидно, что направление — это направление движения лифта). Этот файл со схемой объектов будет использован в этом и нескольких последующих заданиях.

4. Теперь разделите факты для каждого объекта на две группы. Пометьте первую группу как Атрибуты, а вторую группу как Другие Факты. В дальнейшем действия (варианты поведения) должны помещаться в группу Другие Факты. Как только вы поместили действие в группу Другие Факты, посмотрите, не надо ли создать дополнительные элементы в группе Атрибуты. Например, факт «лифт закрывает свои двери» — это действие, которое входит в группу Другие Факты, но оно указывает, что атрибут дверей — это то, что они либо открыты, либо закрыты. Факт «этаж занят» заставляет ввести атрибут этажа, который в любой момент времени будет показывать, занят ли этаж (пассажиром), или не занят. Приведем еще некоторые атрибуты лифта: движется он или стоит, имеет пассажира или нет, а если движется, то вверх или вниз. Атрибут кнопки — «включена» или «выключена». Атрибут пассажира — нужный ему этаж. И так далее.

### Замечания

1. Начните с составления списка атрибутов тех объектов, которые явно упомянуты в постановке задачи. Затем включайте в список атрибуты, которые подразумеваются в постановке задачи.
2. Добавляйте соответствующие атрибуты, как только становится очевидной их необходимость.
3. Проектирование системы — процесс не имеющий четкого момента окончания. Сделайте наилучшим образом то, что пока можете. И будьте готовы к модификации проекта, так как это упражнение будет продолжено в последующих главах.
4. Один объект может быть атрибутом другого объекта. Это называется композицией. Например, объекты-кнопки этажа 1 и этажа 2 внутри лифта — пассажир нажимает одну из этих кнопок, чтобы выбрать нужный ему этаж. Для целей данного лабораторного задания будем считать все объекты независимыми — не образующими композиции. Мы включим композицию в модель лифта в главе 5.
5. В этой главе вы научились создавать «случайность». Когда вы со временем разработаете модель лифта, можно будет использовать оператор `arrivalTime = currentTime + (5 + rand() % 16);` для планирования случайного прибытия пассажира на этаж.

### Резюме

- Лучший способ разработки и поддержки большой программы — разделить ее на несколько меньших модулей, каждый из которых более

управляем, чем исходная программа. Модули пишутся на C++ в виде функций и классов.

- Функция активизируется посредством вызова функции. В вызове указывается имя функции и передается информация (в виде аргументов), которая нужна вызываемой функции для выполнения ее задачи.
- Цель скрытия информации в функциях заключается в том, чтобы дать доступ только к той информации, которая нужна для выполнения их задач. Это средство реализации принципа наименьших привилегий, одного из наиболее важных принципов разработки хорошего программного обеспечения.
- Функции обычно активизируются в программе написанием имени функции, за которым следуют аргументы функции в круглых скобках.
- Тип данных **double** — тип с плавающей запятой, подобный **float**. Переменная типа **double** может хранить значения гораздо большего диапазона и точности, чем **float**.
- Каждый аргумент функции может быть константой, переменной или выражением.
- Локальная переменная известна только в описании данной функции. Функции не знают детали реализации другой функции (включая локальные переменные).
- Общий формат описания функции:

Тип-возвращаемого-значения имя-функции (список-параметров)  
{  
    объявления и операторы  
}

*Тип-возвращаемого-значения* устанавливает тип значения, возвращаемого в вызывающую функцию. Если функция не возвращает значение, *тип-возвращаемого-значения* объявляется как **void**. *Имя-функции* — любой правильно написанный идентификатор. *Список-параметров* — написанный через запятые список, содержащий *объявления* переменных, которые будут переданы функции. Если функция не предусматривает передачу в нее никаких значений, *список-параметров* объявляется как **void**. *Тело-функции* — набор объявлений и операторов, которые составляют функцию.

- Аргументы, передаваемые функции, должны быть согласованы по количеству, типу и порядку следования с параметрами в описании функции.
- Когда программа доходит до вызова функции, управление передается из точки активации к вызываемой функции, функция выполняется и управление возвращается оператору вызова.
- Вызываемая функция может вернуть управление оператору вызова одним из трех способов. Если функция не возвращает никакого значения, управление возвращается при достижении правой заканчивающей функцию фигурной скобки или при выполнении оператора **return**;

Если функция возвращает значение, оператор  
**return выражение;**  
возвращает значение *выражения*.

- Прототип функции объявляет тип возвращаемого значения функции, количество, типы и порядок следования параметров, передаваемых в функцию.
- Прототипы функций дают возможность компилятору проверить, правильно ли вызывается функция.
- Компилятор игнорирует имена переменных, упомянутые в прототипе функции.
- Каждая стандартная библиотека имеет соответствующий заголовочный файл, содержащий прототипы всех функций этой библиотеки, а также определения различных символических констант, необходимых для этих функций.
- Программист может и должен создавать свои собственные заголовочные файлы.
- Если аргумент передается в функцию по значению, создается копия значения переменной и именно она передается вызываемой функции. Изменения копии в вызываемой функции не влияют на значение исходной переменной.
- Функция `rand` генерирует целое число, лежащее в интервале от 0 до значения `RAND_MAX`, которое определяется равным по меньшей мере 32767.
- Прототипы функций `rand` и `srand` содержатся в `<stdlib.h>`.
- Значения, вырабатываемые функцией `rand`, могут быть масштабированы и смещены, чтобы получать значения в указанном диапазоне.
- Чтобы randomизировать программу, используйте функцию `srand` стандартной библиотеки С.
- Оператор `srand` обычно вставляют в программу только после того, как программа полностью отлажена. Пока идет отладка, `srand` лучше пропустить. Это обеспечивает повторяемость, которая важна для того, чтобы доказать, что программа генерации случайных чисел работает должным образом.
- Чтобы randomизировать, не вводя каждый раз новое начальное значение числа, используемого для генерации случайной последовательности, можно использовать функцию `srand(time(NULL))`. Функция `time` обычно возвращает «календарное время» в секундах. Прототип функции `time` находится в файле `<time.h>`.
- Общее уравнение для масштабирования и сдвига случайного числа имеет вид:  
$$n = a + \text{rand}() \% b;$$
где `a` — величина сдвига, (которая равна первому числу в требуемом диапазоне последовательных целых чисел), а `b` — коэффициент масштабирования (который равен ширине требуемого диапазона последовательных целых чисел).
- Перечисление, вводимое ключевым словом `enum` перед именем типа, — это набор целых констант, представленных своими идентификаторами.

- Значения этих констант перечисления начинаются с 0, если не указано иное, и увеличиваются с приращением, равным 1.
- Идентификаторы в enum должны быть уникальными, но отдельные константы перечисления могут иметь одинаковые целые значения.
- Любой константе перечисления может быть присвоено целое значение в определении перечисления.
- Каждый идентификатор переменной характеризуется классом памяти, областью действия и компоновкой.
- C++ обеспечивает четыре спецификации класса памяти: `auto`, `register`, `extern` и `static`.
- Класс памяти идентификатора определяет время жизни этого идентификатора в памяти.
- Область действия идентификатора определяет, где в программе можно ссылаться на этот идентификатор.
- Компоновка идентификатора определяет для программы с многими исходных файлов, известен ли идентификатор только в текущем исходном файле или в любом исходном файле с соответствующим объявлением.
- Переменные автоматического класса памяти создаются при вхождении в блок, в котором они объявлены, существуют лишь пока этот блок активен и уничтожаются при выходе из блока.
- Спецификатор класса памяти `register` может быть помещен перед объявлением автоматической переменной, чтобы указать компилятору разместить переменную в одном из высокоскоростных аппаратных регистров компьютера. Компьютер может проигнорировать объявление `register`. Ключевое слово `register` можно использовать только для переменных автоматического класса памяти.
- Ключевые слова `extern` и `static` используются, чтобы объявить идентификаторы переменных и функций статического класса памяти.
- Переменные статического класса памяти размещаются и получают начальные значения в начале выполнения программы.
- Статический класс памяти имеют два типа идентификаторов: внешние идентификаторы и локальные переменные, объявленные спецификацией класса памяти `static`.
- Глобальные переменные создаются путем помещения их объявления вне какого-либо описания функции и они сохраняют свои значения в течение всего времени выполнения программы.
- Локальные переменные, объявленные как `static`, сохраняют свои значения после выхода из функции, в которой они объявлены.
- Все численные переменные статического класса памяти получают нулевые начальные значения, если программист не присвоил им явно другие начальные значения.
- Существуют четыре области действия идентификаторов: область действия функция, область действия файл, область действия блок и область действия прототип функции.

- Метки являются единственными идентификаторами с областью действия функция. Метки можно использовать всюду внутри функции, в которой они находятся, но на них нельзя ссылаться вне тела функции.
- Идентификатор, объявленный вне любой функции, имеет областью действия файл. Такой идентификатор «известен» с момента его объявления до конца файла.
- Идентификаторы, объявленные внутри блока, имеют областью действия блок. Область действия блок заканчивается завершающей правой фигурной скобкой () .
- Локальные переменные, объявленные в начале функции, имеют областью действия блок подобно параметрам функции, которые считаются локальными переменными функции.
- Любой блок может содержать объявления переменных. Если блоки вложены и идентификатор во внешнем блоке имеет такое же имя, как идентификатор во внутреннем блоке, идентификатор во внешнем блоки «невидим» (скрыт) до тех пор, пока не завершится внутренний блок.
- Единственными идентификаторами, имеющими областью действия прототип функции, являются те, которые использованы в списке параметров прототипа функции. Идентификаторы, использованные в прототипе функции, можно повторно использовать в других местах программы без опасений возникновения неопределенности.
- Рекурсивная функция — это функция, которая прямо или косвенно вызывает сама себя.
- Если рекурсивная функция вызывается базовой задачей, она просто возвращает результат. Если функция вызывается более сложной задачей, функция разделяет задачу на две части: часть, которую функция умеет выполнять, и несколько упрощенный вариант исходной задачи. Поскольку этот вариант подобен исходной задаче, функция путем рекурсивного вызова начинает работать над такой упрощенной задачей.
- Чтобы завершить процесс рекурсии, каждый раз, как функции вызывает саму себя с несколько упрощенной версией исходной задачи, должна формироваться последовательность все меньших и меньших задач, в конце концов сходящаяся к базовой задаче. В этот момент функция распознает базовую задачу, возвращает результат предыдущей копии функции и последовательность возвратов повторяет весь путь назад, пока не дойдет до первоначального вызова и не возвратит конечный результат.
- Стандарт ANSI не регламентирует порядок, в котором вычисляются операнды большинства операций. C++ определяет порядок вычисления операндов только в операциях &&, ||, последования (,) и ?: . Для первых трех бинарных операций операнды гарантированно вычисляются слева направо. Последняя операция — единственная тернарная операция в C++. Ее самый левый операнд всегда выполняется первым; если результат его вычисления отличен от нуля, то следующим вычисляется средний операнд, а последний операнд игнорируется; если же результат вычисления самого левого операнда равен нулю, то следующим вычисляется третий операнд, а средний операнд игнорируется.

- Как итерации, так и рекурсии основаны на управляющих структурах: итерации используют структуру повторения, рекурсии используют структуру выбора.
- Как итерации, так и рекурсии включают повторение: итерации используют структуру повторения явным образом, рекурсии реализуют повторение посредством повторных вызовов функции.
- Как итерации, так и рекурсии включают проверку условия окончания: итерации заканчиваются после нарушения условия продолжения цикла, рекурсии заканчиваются после распознавания базовой задачи.
- Как итерации, так и рекурсии могут оказаться бесконечными: бесконечный итеративный цикл возникает, если условие продолжения цикла никогда не становится ложным; бесконечная рекурсия возникает, если шаг рекурсии не упрощает исходную задачу таким образом, чтобы она сходилась к базовой.
- Повторный запуск рекурсивного механизма вызовов функции приводит к росту накладных расходов: к нарастающим затратам процессорного времени и требуемого объема памяти.
- Программа на C++ не компилируется, если какая-то функция не имеет прототипа и не описана перед ее использованием.
- Функция, не возвращающая значение, объявляется с типом `void`. Если предпринять попытку вернуть значение функции или использовать результат активизации функции в вызывающем выражении, компилятор сообщит об ошибке.
- Пустой список параметров указывается пустыми круглыми скобками или ключевым словом `void` в круглых скобках.
- Встраиваемые функции исключают накладные расходы, связанные с вызовом функции. Программист может использовать ключевое слово `inline`, чтобы посоветовать компилятору генерировать машинные коды функции в нужных местах программы (если это возможно), чтобы минимизировать вызовы функции. Компилятор может проигнорировать `inline`.
- C++ предусматривает возможность прямой формы вызова по ссылке с помощью ссылочных параметров. Чтобы указать, что параметр функции передается по ссылке, после типа параметра в прототипе функции пишется символ `&`. В вызове функции переменная указывается по имени, но она будет передана по ссылке. В вызываемой функции обозначение переменной ее локальным именем на самом деле отсылает к исходной переменной в вызывающей функции. Таким образом, исходная переменная может быть изменена с помощью вызываемой функции.
- Ссылочные параметры могут быть также созданы для локального применения как псевдонимы других переменных внутри функции. Ссылочные переменные должны получить начальные значения в своих объявлениях и они не могут быть переприсвоены как псевдонимы другим переменным. Как только ссылочная переменная объявлена как псевдоним другой переменной, все операции, предположительно выполняемые над псевдонимами, на самом деле выполняются над переменными.
- Спецификация `const` может создать именованную константу. Именованная константа должна получить в качестве начального значения

постоянное выражение и после этого не может изменяться. Именованные константы часто называют постоянными переменными или переменными только для чтения. Именованные константы могут быть помещены всюду, где может быть помещено постоянное выражение. Другим распространенным применением спецификации `const` является создание ссылок на константы.

- С++ позволяет программисту задавать аргументы по умолчанию и их значения по умолчанию. Если аргумент по умолчанию пропускается при вызове функции, используется его значение по умолчанию. Аргументы по умолчанию должны быть крайними правыми (последними) в списке параметров функции. Аргументы по умолчанию должны быть указаны при первом же упоминании имени функции. Значения по умолчанию могут быть константами, глобальными переменными или вызовами функций.
- Унарная операция разрешения области действия (::) позволяет обеспечить доступ к глобальной переменной в случае, когда локальная переменная имеет в области действия такое же имя.
- Возможно определение нескольких функций с одинаковыми именами, но разными типами параметров. Эти функции называются перегруженными. При вызове перегруженной функции компилятор выбирает соответствующую функцию, анализируя количество и тип аргументов в вызове.
- Перегруженные функции могут иметь разные или одинаковые типы возвращаемых значений и обязательно должны иметь разные списки параметров. Две функции, отличающиеся только типами возвращаемых значений, вызовут ошибку компиляции.
- Шаблоны функций предоставляют возможность создания функций, которые выполняют одинаковые операции над разными типами данных, причем шаблон функции определяется только один раз.

## Терминология

|                       |                                |
|-----------------------|--------------------------------|
| <code>const</code>    | автоматическая переменная      |
| <code>enum</code>     | автоматический класс памяти    |
| <code>rand</code>     | активизация функции            |
| <code>RAND_MAX</code> | амперсанд (&)                  |
| <code>return</code>   | аргумент вызова функции        |
| <code>srand</code>    | аргументы функции по умолчанию |
| <code>template</code> | базовая задача рекурсии        |
| <code>time</code>     | библиотека математических      |
| <code>unsigned</code> | функций                        |
| <code>void</code>     |                                |

|                           |                                |
|---------------------------|--------------------------------|
| блок                      | время жизни                    |
| встраиваемая функция      | вызванная функция              |
| вызов по значению         | вызов по ссылке                |
| вызов функции             | вызывающая функция             |
| вызывающий оператор       | выражения со смешанными типами |
| генерация случайных чисел | глобальная переменная          |
| заголовочные файлы        | стандартной библиотеки         |
| заголовочный файл         | именованная константа          |
| итерация                  | класс памяти                   |
|                           | константа перечисления         |

|                                     |                                                    |
|-------------------------------------|----------------------------------------------------|
| копия значения                      | рандомизация                                       |
| локальная переменная                | рекурсивный вызов                                  |
| масштабирование                     | рекурсия                                           |
| моделирование                       | связывание                                         |
| модульная программа                 | сигнатура                                          |
| надежное связывание типов           | смещение                                           |
| область действия                    | скрытие информации                                 |
| область действия файл               | спецификации класса памяти                         |
| область действия функция            | спецификация класса памяти <code>auto</code>       |
| объявление функции                  | спецификация класса памяти <code>extern</code>     |
| описание функции                    | спецификация класса памяти                         |
| оптимизирующий компилятор           | <code>register</code>                              |
| параметры в описании функции        | спецификация класса памяти <code>static</code>     |
| перегрузка                          | спецификация связывания                            |
| переменная только для чтения        | ссылочный параметр                                 |
| переменные <code>static</code>      | стандартная библиотека С                           |
| перечислимый тип                    | тип возвращаемого значения                         |
| побочныиий эффект                   | унарная операция разрешения областей действия (::) |
| повторное использование кода        | функция                                            |
| приведение типа аргумента           | функция <code>inline</code>                        |
| принцип минимизации привилегий      | функция факториал                                  |
| прототип функции                    | шаблон функции                                     |
| разработка программного обеспечения |                                                    |

## Типичные ошибки программирования

- При использовании функций математической библиотеки забывают включать ее заголовочный файл, что приводит к ошибке компиляции. Стандартный заголовочный файл должен быть включен для любой стандартной библиотечной функции, используемой в программе.
- Пропуск типа возвращаемого значения в описании функции вызывает синтаксическую ошибку, если прототип функции определяет возвращаемый тип иначе, чем `int`.
- Если забыть вернуть значение из функции, в которой предполагается возвращение результата, это может привести к неожиданным ошибкам. Описание языка C++ указывает, что результат такой оплошности не определен. В этом случае обычно компиляторы C++ выдают предупреждающее сообщение.
- Возвращение какого-то значения из функции, для которой тип возвращаемого значения объявлен как `void`, вызывает синтаксическую ошибку.
- Объявление параметров функции, имеющих одинаковый тип, в виде `float x, y` вместо `float x, float y`. Объявление параметра `float x, y`, у вызовет ошибку компиляции, так как типы надо указывать для каждого параметра в списке.
- Точка с запятой после правой круглой скобки, закрывающей список параметров в описании функции, является синтаксической ошибкой.

- 3.7. Повторное определение параметра функции как локальной переменной этой функции является синтаксической ошибкой.
- 3.8. Описание функции внутри другой функции является синтаксической ошибкой.
- 3.9. Отсутствие точки с запятой в конце прототипа функции является синтаксической ошибкой.
- 3.10. Преобразование от высшего типа в иерархии типов к низшему может изменить значение данных.
- 3.11. Отсутствие прототипа функции, когда функция не определена перед ее первым вызовом, приводит к синтаксической ошибке.
- 3.12. Использование `strand` вместо `rand` для генерации случайных чисел.
- 3.13. Присвоение целого эквивалента константы перечисления переменной перечислимого типа приводит к замечанию (предупреждению) компилятора.
- 3.14. После того, как константа перечисления определена, попытка присвоить ей другое значение является синтаксической ошибкой.
- 3.15. Использование нескольких спецификаций класса памяти для одного идентификатора. Для идентификатора может быть указана только одна спецификация класса памяти. Например, если вы указали `register`, нельзя указать также `auto`.
- 3.16. Непредумышленное использование одинаковых имен идентификаторов во внутреннем и внешнем блоках, когда на самом деле программист хочет, чтобы идентификатор во внешнем блоке был активным во время работы внутреннего блока.
- 3.17. Забывают возвращать значение из рекурсивной функции, когда оно необходимо. Большинство компиляторов вырабатывает при этом предупреждающее сообщение.
- 3.18. Пропуск базовой задачи или неправильная запись шага рекурсии, из-за чего процесс не сходится к базовой задаче, приводят к бесконечной рекурсии и существенным затратам памяти. Это аналог бесконечного цикла в итеративном (нерекурсивном) процессе. Бесконечная рекурсия может быть также вызвана вводом неправильной величины.
- 3.19. Написание программ, которые зависят от последовательности вычисления операндов каких-то операций, отличных от `&&`, `||`, последования `(,)` и `?:`, может привести к ошибкам, потому что компиляторы могут вычислять операнды не в той последовательности, которую ожидает программист.
- 3.20. Случайный вызов нерекурсивной функции самой себя либо непосредственно, либо косвенно, через другую функцию.
- 3.21. Программы на C++ не компилируются, если для каждой функции нет соответствующего ей прототипа или каждая функция не определяется до того, как она используется.

- 3.22. Поскольку ссылочные параметры упоминаются в теле вызываемой функции только по имени, программист может нечаянно принять ссылочные параметры за параметры, передаваемые по значению. Это может привести к неожиданным эффектам, если исходные копии переменных изменяются вызывающей функцией.
- 3.23. При объявлении множества ссылок в одном операторе делается неверное предположение, что действие символа амперсанд & распространяется на весь записанный через запятую список имен переменных. Чтобы объявить переменные x, y и z как ссылки на целое число, используйте запись int &x, &y, &z; вместо неправильной записи int &x, y, z;
- 3.24. В объявлении ссылочной переменной ей не присваивается начальное значение.
- 3.25. Попытка переприсвоить предварительно объявленную ссылку как псевдоним другой переменной.
- 3.26. Возвращение указателя или ссылки автоматической переменной в вызываемой функции.
- 3.27. Определение и попытка использовать аргумент по умолчанию, не являющийся самым правым (последним) аргументом (если одновременно не являются аргументами по умолчанию все более правые аргументы).
- 3.28. Попытка получить доступ к неглобальной переменной внешнего блока, используя унарную операцию разрешения области действия.
- 3.29. Создание перегруженных функций с идентичными списками параметров и различными типами возвращаемых значений приводит к синтаксической ошибке.
- 3.30. Функция с пропущенными аргументами по умолчанию может оказаться вызванной аналогично другой перегруженной функции; это синтаксическая ошибка.
- 3.31. Отсутствие ключевого слова **class** перед каждым формальным параметром шаблона функции.
- 3.32. В сигнатуре функции не используются все формальные параметры шаблона функции.

## Хороший стиль программирования

- 3.1. Внимательно изучайте широкий набор функций в стандартной библиотеке ANSI C и классов в различных библиотеках классов.
- 3.2. При использовании функций математической библиотеки включайте соответствующий заголовочный файл с помощью директивы препроцессора `#include <math.h>`.
- 3.3. Размещайте пустую строку между описаниями функций, чтобы отделить функции и облегчить чтение программы.

- 3.4. Несмотря на то, что пропущенный тип возвращаемого значения по умолчанию `int`, всегда задавайте тип возвращаемого значения явным образом. Исключением является функция `main`, для которой тип возвращаемого значения обычно не указывается.
- 3.5. Не используйте одинаковые имена для аргументов, передаваемых в функцию, и соответствующих параметров в описании функции, хотя это и не является ошибкой. Использование разных имен помогает избежать двусмыслинности.
- 3.6. Выбор осмысленных имен функций и осмысленных имен параметров делает программу более легко читаемой и помогает избежать излишних комментариев.
- 3.7. Имена параметров могут быть включены в прототипы функции с целью документирования. Компилятор эти имена игнорирует.
- 3.8. Делайте заглавной первую букву идентификатора, используемого как имя типа, определенного пользователем.
- 3.9. Используйте в именах констант перечисления только заглавные буквы. Это выделяет константы в тексте программы и напоминает программисту о том, что константы перечисления не являются переменными.
- 3.10. Использование перечислений вместо целых констант облегчает чтение программы.
- 3.11. Переменные, используемые только в отдельной функции, предпочтительнее объявлять как локальные переменные этой функции, а не как глобальные переменные.
- 3.12. Избегайте применения имен переменных, которые незримо уже используются во внешних областях действия. Этого можно достигнуть, вообще избегая использования в программе одинаковых идентификаторов.
- 3.13. Спецификацию `inline` целесообразно применять только для небольших и часто используемых функций.
- 3.14. Использование аргументов по умолчанию может упростить написание вызовов функций. Однако некоторые программисты считают, что для ясности лучше явно указывать все аргументы.
- 3.15. Избегайте использования в программе переменных с одинаковыми именами для разных целей. Хотя это и допускается в различных случаях, но может приводить к путанице.
- 3.16. Перегруженные функции, которые выполняют тесно связанные задачи, делают программы более понятными и легко читаемыми.

## Советы по повышению эффективности

- 3.1. Автоматический класс памяти — это средство экономии памяти, так как переменные этого класса создаются при входе в блок, в котором они объявлены, и уничтожаются при выходе из блока.

- 3.2. Спецификация класса памяти `register` может быть помещена перед объявлением автоматической переменной, чтобы компилятор сохранял переменную не в памяти, а в одном из высокоскоростных аппаратных регистров компьютера. Если интенсивно используемые переменные, такие как счетчики или суммы могут сохраняться в аппаратных регистрах, накладные расходы на повторную загрузку переменных из памяти в регистр и обратную загрузку результата в память могут быть исключены.
- 3.3. Часто объявления `register` не являются необходимыми. Современные оптимизирующие компиляторы способны распознавать часто используемые переменные и решать, помещать их в регистры или нет, не требуя от программиста объявления `register`.
- 3.4. Избегайте рекурсивных программ, подобных программе для вычисления чисел Фибоначчи, которые приводят к экспоненциальному нарастанию количества вызовов.
- 3.5. Избегайте использования рекурсий в случаях, когда требуется высокая эффективность. Рекурсивные вызовы требуют времени и дополнительных затрат памяти.
- 3.6. Программа, разбитая на множество функций, потенциально требует большего количества вызовов функций по сравнению с монолитной программой без функций. Это увеличивает время выполнения и затраты памяти компьютера. Но монолитные программы трудно программировать, тестировать, отлаживать, сопровождать и развивать.
- 3.7. Использование функций `inline` может уменьшить время выполнения программы, но может увеличить ее размер.
- 3.8. Для передачи больших объектов используйте константный ссылочный параметр, чтобы обеспечить защиту параметров, как при передаче по значению, и в то же время избежать накладных расходов при передаче копии большого объекта.

## Замечания по мобильности

- 3.1. Использование стандартной библиотеки ANSI C увеличивает мобильность программы.
- 3.2. Программы, которые зависят от последовательности вычисления операндов операций, отличных от `&&`, `||`, последования `(,)` и `?:`, могут по-разному работать в системах с различными компиляторами.
- 3.3. Роль пустого списка параметров функции в C++ существенно отличается от C. В C это означает, что все проверки аргументов отсутствуют (т.е. вызов функции может передать любой аргумент, который требуется). А в C++ пустой список означает отсутствие аргументов. Таким образом, программа на C, использующая эту особенность, может сообщить о синтаксической ошибке при компиляции в C++.

## Замечания по технике программирования

- 3.1. Избегайте повторного изобретения колеса. Если возможно, используйте стандартную библиотеку ANSI C вместо того, чтобы писать новые функции. Это сокращает затраты времени на создание программы.
- 3.2. В программах, содержащих много функций, *main* должна быть построена как группа вызовов функций, которые и выполняют основную часть работы.
- 3.3. Каждая функция должна выполнять одну хорошо определенную задачу и имя функции должно наглядно отражать данную задачу. Это способствует успешному повторному использованию программного обеспечения.
- 3.4. Если вы не можете выбрать подходящее имя, которое бы выражало суть работы функции, то может быть на нее возложено слишком много различных задач. Обычно лучше разбить такую функцию на несколько более мелких.
- 3.5. Обычно функция должна быть не длиннее одной страницы. Еще лучше, если она будет не длиннее половины страницы. Безотносительно к длине функции она должна хорошо определять только одну задачу. Небольшие функции способствуют повторному использованию программных кодов.
- 3.6. Программа должна быть написана как совокупность небольших функций. Это облегчает написание, отладку, сопровождение и модификацию программы.
- 3.7. Функция, требующая большого количества параметров, возможно, выполняет слишком много задач. Попробуйте разделить такую функцию на небольшие функции, которые выполняют отдельные задачи. Заголовок функции по возможности не должен занимать более одной строки.
- 3.8. Прототип функции, заголовок функции и вызовы функции должны быть согласованы между собой по количеству, типу и порядку следования аргументов и параметров и по типу возвращаемых результатов.
- 3.9. В C++ требуются прототипы функций. Используйте директиву препроцессора `#include`, чтобы получить прототипы стандартных библиотечных функций из заголовочных файлов соответствующих библиотек. Используйте также `#include` для заголовочных файлов, содержащих прототипы функций, используемых вами или членами вашей группы.
- 3.10. Прототип функции, размещенный вне описания какой-то другой функции, относится ко всем вызовам данной функции, появляющимся после этого прототипа в данном файле. Прототип функции, размещенный внутри описания некоторой функции, относится только к вызовам внутри этой функции.

- 3.11. Автоматическое хранение — еще один пример принципа наименьших привилегий. Зачем хранить в памяти и делать доступными переменные, когда они не нужны?
- 3.12. Объявление переменной глобальной, а не локальной, приводит к неожиданным побочным эффектам, когда функции, не нуждающиеся в доступе к этой переменной, случайно или намеренно изменяют ее. Вообще лучше избегать использования глобальных переменных за исключением особых случаев, когда требуется уникальная производительность (это будет рассмотрено в главе 18).
- 3.13. Любые задачи, которые можно решить рекурсивно, могут быть решены также и итеративно (нерекурсивно). Обычно рекурсивный подход предпочитают итеративному, если он более естественно отражает задачу и ее результаты, то есть более нагляден и легче отлаживается. Другая причина предпочтения рекурсивного решения состоит в том, что итеративное решение может не быть очевидным.
- 3.14. Функционализация программ в четком иерархическом стиле — свидетельство хорошей техники программирования. Но за все надо платить.
- 3.15. Любые изменения функции `inline` могут потребовать перекомпиляции всех «потребителей» этой функции. Это может оказаться существенным моментом для развития и поддержки некоторых программ.
- 3.16. Чтобы обеспечить одновременно ясность программы и ее производительность, многие программисты на C++ предпочитают, чтобы аргументы, которые могут изменяться, передавались бы функциям с помощью указателей, аргументы, не предназначенные для изменения, передавались бы по значению, а большие неизменяемые аргументы передавались бы функциям путем использования константных ссылок.

### Упражнения для самопроверки

- 3.1. Заполнить пробелы в следующих утверждениях:
- Программные модули на C++ называются \_\_\_\_\_ и \_\_\_\_\_.
  - Функция активируется с помощью \_\_\_\_\_.
  - Переменная, которая известна только внутри функции, в которой она определена, называется \_\_\_\_\_.
  - Оператор \_\_\_\_\_ в вызываемой функции используется, чтобы передать значение выражения обратно в вызывающую функцию.
  - Ключевое слово \_\_\_\_\_ используется в заголовке функции, чтобы указать, что функция не возвращает значение или указать, что она не содержит параметров.
  - \_\_\_\_\_ идентификатора — это часть программы, в которой идентификатор может быть использован.
  - Существуют три пути возвращения управления из вызванной функции в оператор вызова \_\_\_\_\_, \_\_\_\_\_ и \_\_\_\_\_.

- h) \_\_\_\_\_ позволяет компилятору проверить количество, типы и порядок следования аргументов, передаваемых функции.
- i) Функция \_\_\_\_\_ используется для получения случайных чисел.
- j) Функция \_\_\_\_\_ используется, чтобы установить случайное начальное значение числа для randomизации программы.
- k) Спецификациями классов памяти являются \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ и \_\_\_\_\_.
- l) Переменные, объявленные в блоке или в списке параметров функции, имеют класс памяти \_\_\_\_\_, если не указано иное.
- m) Спецификация класса памяти \_\_\_\_\_ представляет собой рекомендацию компилятору хранить переменную в одном из регистров компьютера.
- n) Переменная, объявленная вне любого блока или функции, является \_\_\_\_\_ переменной.
- o) Для того, чтобы локальная переменная функции сохраняла свое значение между вызовами функции, она должна быть объявлена как имеющая класс памяти \_\_\_\_\_.
- p) Четырьмя возможными областями действия идентификатора являются \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ и \_\_\_\_\_.
- q) Функция, которая прямо или косвенно вызывает сама себя, называется \_\_\_\_\_ функцией.
- r) Рекурсивная функция обычно имеет два компонента: один, который обеспечивает завершение рекурсии проверкой, не является ли задача \_\_\_\_\_, и другой, который представляет задачу как рекурсивный вызов упрощенной по сравнению с исходной задачи.
- s) В C++ можно иметь разные функции с одинаковым именем, каждая из которых оперирует с различными типами и (или) количеством аргументов. Такая функция называется \_\_\_\_\_.
- t) \_\_\_\_\_ предоставляет возможность доступа к глобальной переменной с тем же именем, что и переменная в текущей области действия.
- u) Спецификация \_\_\_\_\_ используется для объявления переменных только для чтения.
- v) \_\_\_\_\_ функции предоставляют возможность определить единственную функцию для выполнения заданий с многими различными типами данных.
- 3.2. Для приведенной ниже программы установите области действия (область действия функция, область действия файл, область действия блок или область действия прототип функции) каждого из следующих элементов:
- Переменная `x` в `main`.
  - Переменная `y` в `cube`.
  - Функция `cube`.
  - Функция `main`.

- e) Прототип функции `cube`.  
f) Идентификатор `y` в прототипе функции `cube`.

```
#include <iostream.h>
int cube(int y);

main()
{
 int x;

 for (x = 1; x <= 10; x++)
 cout << cube(x) << endl;
}

int cube(int y);
{
 return y * y * y;
}
```

- 3.3. Напишите программу, которая проверяет, получают ли на самом деле показанные на рис. 3.2 примеры вызовов математических библиотечных функций указанные результаты.
- 3.4. Напишите заголовки для каждой из следующих функций:
- Функция `hypotenuse`, которая принимает два аргумента с плавающей запятой с удвоенной точностью `side1` и `side2` и возвращает результат с плавающей запятой с удвоенной точностью.
  - Функция `smallest`, которая принимает три целых значения `x`, `y` и `z` и возвращает целое значение.
  - Функция `instructions`, которая не получает ни одного аргумента и не возвращает значение. (Заметим, что такие функции обычно используются для выдачи на экран указаний пользователю).
  - Функция `intToFloat`, которая принимает целый аргумент `number` и возвращает результат с плавающей запятой.
- 3.5. Напишите прототип для каждой из следующих функций:
- Функции, описанной в упражнении 3.4a.
  - Функции, описанной в упражнении 3.4b.
  - Функции, описанной в упражнении 3.4c.
  - Функции, описанной в упражнении 3.4d.
- 3.6. Напишите объявление для следующих переменных:
- Целая `count`, которая должна содержаться в регистре. Начальное значение `count` равно 0.
  - Переменная с плавающей запятой `lastVal`, сохраняющая свое значение между вызовами функции, в которой она определена.
  - Внешняя целая `number`, чья область действия должна быть ограничена оставшейся частью файла, в котором она определена.
- 3.7. Найдите ошибку в каждом из следующих фрагментов программ и объясните, как можно исправить ошибку (смотри также упражнение 3.53):

```

a) int g(void) {
 cout << "Внутри функции g" << endl;

 int h(void) {
 cout << "Внутри функции h" << endl;
 }
}

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;

 cout << a << endl;
}

e) void product (void) {
 int a, b, c, result;
 cout << "Введите три целых числа: ";
 cin >> a >> b >> c >>;
 result = a * b * c;
 cout << "Результат равен " << result;
 return result;
}

```

- 3.8. В каком случае прототип функции должен содержать объявление типа параметра `float&?`
- 3.9. Верно или нет, что все вызовы в C++ выполняются вызовом по значению.
- 3.10. Напишите законченную программу на C++, использующую встроенную функцию `sphereVolume`, которая предлагает пользователю ввести радиус сферы, вычисляет и печатает объем этой сферы, используя присваивание `volume = (4/3 * 3.14159 * pow(radius, 3))`.

### Ответы на упражнения для самопроверки

- 3.1. а) функции, классы. б) вызова функции. с) локальная переменная. д) `return`. е) `void`. ф) область действия. г) `return;`, `return выражение;`, закрывающая правая фигурная скобка функции. х) прототип функции. і) `rand`. ж) `rand`. к) `auto`, `register`, `extern`, `static`. л) `automatic`. м) `register`. н) глобальной. о) `static`. р) область действия функция, область действия файл, область действия блок, область действия прототип функции. ё) рекурсивной. г) базовой. с) перегруженной. т) Унарная операция разрешения области действия (():. у) `const`. в) Шаблоны.

- 3.2. а) Область действия блок. б) Область действия блок. с) Область действия файл. д) Область действия файл. е) Область действия файла.  
ф) Область действия прототип функции.

3.3. /\* Проверка математических библиотечных функций \*/  

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
main()
{
 cout << setiosflags (ios:: fixed | ios:: showpoint)
 << setprecision(1)
 << "sqrt(" << 900.0 << ") = " << sqrt(900.0) << endl
 << "sqrt(" << 9.0 << ") = " << sqrt(9.0) << endl
 << "exp(" << 1.0 << ") = " << setprecision(6)
 << exp(1.0) << endl << "exp(" << setprecision(1) << 2.0
 << ") = " << setprecision(6) << exp(2.0) << endl
 << "log(" << 2.718282 << ") = " << setprecision(1)
 << log(2.718282) << endl << "log(" << setprecision(6)
 << 7.389056 << ") = " << setprecision(1)
 << log(7.389056) << endl;
 cout << "log10(" << 1.0 << ") = " << log10(1.0) << endl
 << "log10(" << 10.0 << ") = " << log10(10.0) << endl
 << "log10(" << 100.0 << ") = " << log10(100.0) << endl
 << "fabs(" << 13.5 << ") = " << fabs(13.5) << endl
 << "fabs(" << 0.0 << ") = " << fabs(0.0) << endl
 << "fabs(" << -13.5 << ") = " << fabs(-13.5) << endl;
 cout << "ceil(" << 9.2 << ") = " << ceil(9.2) << endl
 << "ceil(" << -9.8 << ") = " << ceil(-9.8) << endl
 << "floor(" << 9.2 << ") = " << floor(9.2) << endl
 << "floor(" << -9.8 << ") = " << floor(-9.8) << endl;
 cout << "pow(" << 2.0 << ", " << 7.0 <<) = << pow(2.0, 7.0)
 << endl << "pow(" << 9.0 << ", " << 0.5 << ") = "
 << pow(9.0, 0.5) << endl << setprecision(3) << "fmod("
 << 13.675 << ", " << 2.333 << ") = " << fmod(13.675, 2.333)
 << endl << setprecision(1) << "sin(" << 0.0 << ") = "
 << sin(0.0) << endl << "cos(" << 0.0 << ") = " << cos(0.0)
 << endl << "tan(" << 0.0 << ") = " << tan(0.0) << endl;
 return 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(1.000) = 10.0
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, 7) = 128.0
```

```

pow(9,0.5) = 3.0
fmod(13.657, 2.333)= 2.010
sin(0.0) = 0.0
cos(0.0) = 0.0
tan(0.0) = 0.0

```

- 3.4.** a) double hipotenuse(double side1, double side2)
- b) int smallest(int x, int y, int z)
- c) void instructions(void) // в C++ (void) можно  
// записать как ()
- d) float intToFloat(int number)
- 3.5.** a) double hipotenuse(double, double);
- b) int smallest(int, int, int);
- c) void instructions(void); // в C++ (void) можно  
// записать как ()
- d) float intToFloat(int);
- 3.6.** a) register int cout = 0;
- b) static float lastVal;
- c) static int number;
- Замечание: это это должно быть записано вне любого описания функции.
- 3.7.** a) Ошибка: функция h описана в функции g.  
Исправление: уберите описание h из описания g.
- b) Ошибка: функция предполагает возвращение целого числа, но его нет.  
Исправление: удалите переменную result и вставьте в функцию оператор
- ```
return x + y
```
- c) Ошибка: результат n + sum(n-1) не возвращается; sum не возвращает соответствующего результата.
Исправление: перепишите оператор в операторе else в виде:
- ```
return n + sum(n-1);
```
- d) Ошибки: точка с запятой после правой круглой скобки, которая закрывает список параметров, и переопределение параметра a в описании функции.  
Исправление: удалите точку с запятой после правой круглой скобки списка параметров и удалите объявление float a;
- e) Ошибка: функция возвращает значение, хотя это не предполагается.  
Исправление: удалите оператор return.
- 3.8.** Если программист объявляет ссылочный параметр типа «ссылка на» float, чтобы получить доступ по ссылке к исходной переменной аргумента.

**3.9.** Неверно. C++ допускает прямой вызов по ссылке путем использования ссыльного параметра, а кроме того можно использовать указатели.

**3.10.** //Встраиваемая функция, вычисляющая объем сферы.

```
#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 << "Введите радиус сферы: ";
 cin >> radius;
 cout << "Объем сферы с радиусом " >> radius <<
 " равен " << sphereVolume(radius) << '\n';
 return 0;
}
```

## Упражнения

**3.11.** Укажите значения x после выполнения каждого из следующих операторов:

- 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)))

**3.12.** За стоянку до трех часов парковочный гараж запрашивает плату минимум \$2.00. В случае стоянки более трех часов гараж дополнительно запрашивает \$0.50 за каждый полный или неполный час сверх трех часов. Максимальная плата за сутки составляет \$10.00. Допустим, что никто не паркуется более, чем на сутки за раз. Напишите программу расчета и печати оплаты за парковку для каждого из трех клиентов, которые парковали свои автомобили вчера в этом гараже. Вы должны вводить длительность парковки для каждого клиента. Ваша программа должна печатать результаты в аккуратном табулированном формате и должна рассчитывать и печатать общий вчерашний доход. Программа должна использовать функцию calculateCharges, чтобы определять плату для каждого клиента. Результаты работы должны представляться в следующем формате:

| Автомобиль | Часы | Плата |
|------------|------|-------|
| 1          | 1.5  | 2.00  |
| 2          | 4.0  | 2.50  |
| 3          | 24.0 | 10.00 |
| Итого      | 29.5 | 14.50 |

**3.13.** Функция **floor** может использоваться для округления значения до ближайшего целого. Оператор

```
y = floor(x + .5);
```

округляет значение **x** до ближайшего целого и присваивает результат переменной **y**. Напишите программу, которая читает несколько чисел и использует указанный выше оператор для округления каждого из этих чисел до ближайшего целого. Для каждого обрабатываемого числа напечатайте исходное число и его округленное значение.

**3.14.** Функция **floor** может использоваться для округления значения до ближайшего целого. Оператор

```
y = floor(x * 10 + .5) / 10;
```

округляет **x** с точностью до одной десятой (первая позиция справа от десятичной точки). Оператор

```
y = floor(x * 100 + .5) / 100;
```

округляет **x** с точностью до одной сотой (вторая позиция справа от десятичной точки). Напишите программу, которая определяет четыре функции для округления **x** различными способами:

- a) `roundToInteger(number)`
- b) `roundToTenths(number)`
- c) `roundToHundredths(number)`
- d) `roundToThousandths(number)`

Для каждого прочитанного значения ваша программа должна печатать исходное значение, число, округленное до ближайшего целого; число, округленное до ближайшей десятой, число, округленное до ближайшей сотой, и число, округленное до ближайшей тысячной.

**3.15.** Ответьте на каждый из следующих вопросов:

- a) Что означает выбрать «случайные» числа?
- b) Почему функция **rand** полезна для моделирования азартных игр?
- c) Каким образом randomизируется программа при использовании функции **strand**? При каких обстоятельствах randomизация нежелательна?
- d) Почему часто необходимо масштабировать и сдвигать числа, вырабатываемые программой **rand**?
- e) Чем полезно компьютерное моделирование реальных ситуаций?

**3.16.** Напишите операторы, которые присваивают случайные целые значения переменной **n** в следующих диапазонах:

- a)  $1 \leq n \leq 2$
- b)  $1 \leq n \leq 100$
- c)  $0 \leq n \leq 9$
- d)  $1000 \leq n \leq 1112$
- e)  $-1 \leq n \leq 1$

f)  $-3 \leq n \leq 11$

- 3.17. Для каждого из следующих наборов целых чисел напишите единственный оператор, который будет печатать случайно выбранное число из набора:

- a) 2, 4, 6, 8, 10
- b) 3, 5, 7, 9, 11
- c) 6, 10, 14, 18, 22

- 3.18. Напишите функцию `integerPower(base, exponent)`, которая возвращает значение

$$\text{base}^{\text{exponent}}$$

Например,  $\text{integerPower}(3, 4) = 3 * 3 * 3 * 3$ . Считайте, что `exponent` — положительное, не равное нулю, целое число, а `base` — целое число. Функция `integerPower` должна использовать для управления вычислениями `for` или `while`. Не используйте никаких математических функций.

- 3.19. Определите функцию `hypotenuse`, которая вычисляет длину гипотенузы правильного треугольника, когда две другие стороны известны. Используйте эту функцию в программе для определения длины гипотенузы каждого из следующих треугольников. Функции должны иметь два аргумента типа `double` и возвращать значение гипотенузы как `double`.

| Треугольник | Сторона 1 | Сторона 2 |
|-------------|-----------|-----------|
| 1           | 3.0       | 4.0       |
| 2           | 5.0       | 12.0      |
| 3           | 8.0       | 15.0      |

- 3.20. Напишите функцию `multiple`, которая определяет для пары целых чисел, кратно ли второе число первому. Функция должна воспринимать два целых аргумента и возвращать 1 (истина), если второе число кратно первому, и 0 (ложь), если нет. Используйте эту функцию в программе, которая вводит последовательность пар целых чисел.

- 3.21. Напишите программу, которая вводит последовательность целых чисел и передает их по одному функции `even`, которая использует операцию вычисления остатка для определения четности числа. Функция должна принимать целый аргумент и возвращать 1, если аргумент — четное число, и 0 в противном случае.

- 3.22. Напишите программу, которая отображает у левого края экрана сплошной квадрат из звездочек, сторона которого указана целым параметром `side`. Например, если `side` равна 4, функция должна отображать следующую картинку:

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

- 3.23.** Модифицируйте функцию, созданную в упражнении 3.22, так, чтобы формировать квадрат из каких угодно символов, указанных в символьном параметре `fillCharacter`. Таким образом, если `side` равна 5 и `fillCharacter` равен `#`, то эта функция должна напечатать:

```
#
#
#
#
#
```

- 3.24.** Используйте подход, развитый в упражнениях 3.22 и 3.23, для создания программы, которая вычерчивает широкий диапазон форм.

- 3.25.** Напишите фрагменты программ, которые бы выполняли следующее:

- Вычислить целую часть частного от деления целого числа `a` на целое число `b`.
- Вычислить целый остаток от деления целого числа `a` на целое число `b`.
- Использовать фрагменты программ, созданные в пунктах а) и б), для написания функции, которая вводит целое число из диапазона от 1 до 32767 и печатает его как последовательность цифр, каждая из которых отделена от соседней двумя пробелами. Например, целое число 4562 должно быть напечатано в виде

```
4 5 6 2
```

- 3.26.** Напишите функцию, которая воспринимает время как три целых аргумента (часы, минуты и секунды) и возвращает количество секунд, прошедших со времени, когда часы в последний раз показали 12. Используйте эту функцию для вычисления интервала времени в секундах между двумя моментами, находящимися внутри двенадцатичасового цикла.

- 3.27.** Разработайте следующие целые функции:

- Функцию `celsius`, которая возвращает температуру по Цельсию, эквивалентную температуре по Фаренгейту.
- Функцию `fahrenheit`, которая возвращает температуру по Фаренгейту, эквивалентную температуре по Цельсию.
- Используйте эти функции для написания программы, которая печатает таблицу, показывающую эквивалент по Фаренгейту всех температур по Цельсию от 0 до 100 градусов и эквивалент по Цельсию всех температур по Фаренгейту от 32 до 212 градусов. Напечатайте вывод в аккуратном табулированном формате, с минимальным количеством строк при сохранении хорошей читаемости.

- 3.28.** Напишите функцию, которая возвращает наименьшее из трех чисел с плавающей запятой.

- 3.29.** Говорят, что целое число является *совершенным числом*, если его сомножители, включая 1 (но не само число) в сумме дают это число. Например, 6 — это совершенное число, так как  $6 = 1 + 2 + 3$ . Напишите функцию `perfect`, которая определяет, является ли параметр `number` совершенным числом. Используйте эту функцию в про-

граммой, которая определяет и печатает все совершенные числа в диапазоне от 1 до 1000. Напечатайте сомножители каждого совершенного числа, чтобы убедиться, что число действительно совершенное. Исследуйте мощность вашего компьютера проверкой чисел, много больших 1000.

- 3.30. Говорят, что целое число является *простым числом*, если оно делится только на 1 и на само себя. Например, 2, 3, 5 — простые числа, а 4, 6, 8 — нет.
- а) Напишите функцию, определяющую, является ли число простым или нет.
- б) Используйте эту функцию в программе, которая определяет и печатает все простые числа, лежащие в диапазоне от 1 до 10000. Сколько из этих 10000 чисел вы должны действительно проверить, чтобы быть уверенным в том, что найдены все простые числа?
- в) Вначале вы могли бы подумать, что верхней границей, до которой вы должны проводить проверку, чтобы увидеть, является ли число  $n$  простым, является  $n/2$ , но в действительности вам нужно проверить количество чисел, равное корню квадратному из  $n$ . Почему? Перепишите программу и запустите ее для обоих способов. Оцените улучшение производительности.
- 3.31. Напишите функцию, которая воспринимает целое значение и возвращает число с обратным порядком цифр. Например, воспринимается число 7631, а возвращается число 1367.
- 3.32. Наибольший общий делитель двух целых чисел — это наибольшее целое, на которое без остатка делится каждое из двух чисел. Напишите функцию `pod`, которая возвращает наибольший общий делитель двух целых чисел.
- 3.33. Напишите функцию `qualityPoints`, которая вводит среднюю оценку студентов и возвращает 4, если средняя оценка 90–100, 3, если средняя оценка 80–89, 2, если средняя оценка 70–79, 1, если средняя оценка 60–69, и 0, если средняя оценка меньше 60.
- 3.34. Напишите программу, моделирующую бросание монеты. Для каждого броска монеты программа должна печатать Орел или Решка. Промоделируйте с помощью этой программы бросание 100 раз и подсчитайте, сколько раз появилась каждая сторона монеты. Напечатайте результаты. Программа должна вызывать отдельную функцию `flip`, которая не принимает никаких аргументов и возвращает 0 для Орла и 1 для Решки. Замечание: если программа действительно моделирует бросание монеты, каждая сторона монеты должна появляться примерно в половине случаев.
- 3.35. Компьютеры играют все возрастающую роль в образовании. Напишите программу, которая поможет слушателям начальной школы изучить умножение. Используйте `rand` для выработки двух положительных одноразрядных целых чисел. Программа должна печатать вопрос типа

Сколько будет  $6 * 7$ ?

Затем учащийся печатает ответ. Ваша программа проверяет этот ответ. Если он правильный, напечатайте «Очень хорошо!» и затем задайте следующий вопрос на умножение. Если ответ неправильный, напечатайте «Нет. Повторите, пожалуйста, снова.» и затем задавайте тот же самый вопрос повторно до получения правильного ответа.

- 3.36.** Использование компьютеров в образовании относят к *системам компьютерного обучения* (СКО). Одной из проблем среды СКО является утомляемость учащихся. Этого можно избежать путем разнообразия компьютерных диалогов, удерживающих внимание ученика. Модифицируйте программу упражнения 3.35 так, чтобы для каждого правильного или неправильного ответов печатались разнообразные комментарии типа:

Отклики на правильные ответы

Очень хорошо!

Отлично!

Чудесная работа!

Продолжайте работать так же хорошо!

Отклики на неправильные ответы

Нет. Попытайтесь, пожалуйста, снова.

Неверно. Попытайтесь еще раз.

Не опускайте руки!

Нет. Продолжайте ваши попытки.

Используя генератор случайных чисел для выбора чисел от 1 до 4, выбирайте подходящую реплику для каждого ответа. Используйте структуру `switch` для представления отклика.

- 3.37.** Большинство серьезных систем компьютерного обучения осуществляют мониторинг учащегося в течение некоторого периода времени. Решение о начале новой темы часто основывается на успехе учащегося в изучении предыдущей темы. Модифицируйте программу в упражнении 3.35 так, чтобы считать количество правильных и неправильных ответов, введенных учащимся. После того как учащийся напечатал 10 ответов, ваша программа должна посчитать процент правильных ответов. Если он ниже 75%, ваша программа должна напечатать «Пожалуйста попросите срочно помочь у вашего преподавателя» и закончить свою работу.

- 3.38.** Напишите программу, которая играет в игру «Угадай число» следующим образом: ваша программа выбирает случайное число, которое должно быть отгадано, в диапазоне от 1 до 1000. Затем программа печатает:

Мое число между 1 и 1000.

Вы можете его отгадать?

Пожалуйста, напечатайте вашу первую догадку.

Затем игрок печатает свою первую догадку. Программа отвечает одним из следующих вариантов:

1. Отлично! Вы отгадали число!

Хотели бы вы сыграть еще раз (д или н)?

2. Слишком мало. Попытайтесь снова.

3. Слишком много. Попытайтесь снова.

Если догадка игрока неверна, ваша программа должна работать циклически до получения верного ответа. Программа должна говорить игроку **Слишком мало** или **Слишком много**, чтобы помочь ему угадать правильный ответ. Замечание: Техника поиска, используемая в этой задаче, называется *двоичный поиск*. Об этом мы подробнее поговорим в следующей задаче.

- 3.39.** Модифицируйте программу упражнения 3.38 так, чтобы она считала количество попыток игрока отгадать число. Если это число 10 или меньше, напечатайте **Или вы знаете секрет, или вы счастливчик!** Если игрок отгадал число за 10 попыток, напечатайте **Ага!** **Вы знаете секрет!** Если игрок отгадывает больше чем за 10 попыток, то напечатайте **Вы должны развивать свои способности!** Почему отгадка не должна требовать более 10 попыток? Да потому, что каждая «хорошая догадка» должна исключать половину чисел. Теперь покажите, почему любое число от 1 до 1000 может быть отгадано не более чем за 10 попыток.
- 3.40.** Напишите рекурсивную функцию **power (base, exponent)**, которая возвращала бы значение

$$\text{power}^{\text{exponent}}$$

Например,  $\text{power}(3, 4) = 3 * 3 * 3 * 3$ . Полагайте, что **exponent** — целое число, большее или равное 1. Подсказка: Шаг рекурсии может использовать соотношение

$$\text{base}^{\text{exponent}} = \text{base} \cdot \text{base}^{\text{exponent} - 1}$$

а завершения может иметь место, когда **exponent** равна 1, потому что

$$\text{base}^1 = \text{base}$$

- 3.41.** Последовательность чисел Фибоначчи

0, 1, 1, 2, 3, 5, 8, 13, 21,...

начинается с 0 и 1 и имеет то свойство, что каждый последующий элемент является суммой двух предыдущих элементов. а) Напишите *нерекурсивную* функцию **fibonacci (n)**, которая вычисляет n-е число Фибоначчи. б) Определите наибольшее число Фибоначчи, которое может быть напечатано в вашей системе. Модифицируйте программу части а) так, чтобы использовать **double** вместо **int** при вычислении и возвращении значения числа Фибоначчи; используйте эту модифицированную программу для повторения части б).

- 3.42. (Ханойские башни).** Каждый подающий надежды исследователь в области вычислений рано или поздно должен столкнуться с некоторыми классическими задачами и Ханойские Башни (смотри рис. 3.28) — одна из наиболее известных среди них. Легенда гласит, что в одном из монастырей Дальнего Востока монахи пытались переместить стопку дисков с одного колышка на другой. Начальная стопка имела 64 диска, нанизанных на один колышек так, что их размеры последовательно уменьшались к вершине. Монахи пытались переместить эту стопку с этого колышка на второй при условии, что при каждом перемещении можно брать только один диск и больший диск никогда не должен находиться над меньшим дис-

ком. Третий колышек предоставляет возможность временного размещения дисков. Считают, что когда монахи решат свою задачу, наступит конец света, так что у нас мало поводов облегчать их усилия.

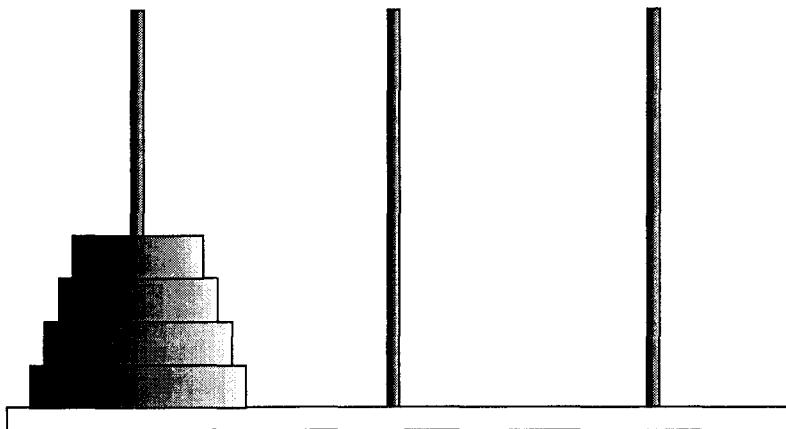


Рис. 3.28. Ханойские башни для случая четырех дисков

Давайте предположим, что монахи пытаются переместить диски с колышка 1 на колышек 3. Мы хотим построить алгоритм, который будет печатать четкую последовательность перемещений дисков с колышка на колышек.

Если бы мы пытались найти решение этой задачи обычными методами, мы быстро бы обнаружили безнадежность попыток манипуляций дисками. Но если мы решим действовать рекурсивными методами, проблема сразу становится легко разрешимой. Перемещение  $n$  дисков может быть легко представлено в терминах перемещения только  $n - 1$  диска (и следовательно рекурсивно):

1. Переместить  $n - 1$  дисков с колышка 1 на колышек 2, используя колышек 3 как место временного размещения.
2. Переместить последний диск (наибольший) с колышка 1 на колышек 3.
3. Переместить  $n - 1$  дисков с колышка 2 на колышек 3, используя колышек 1 как место временного размещения.

Этот процесс завершается, когда последняя задача будет состоять из перемещения  $n = 1$  дисков, т. е. окажется базовой задачей. Она соответствует тривиальному перемещению диска без использования места временного размещения.

Напишите программу решения задачи о Ханойских башнях. Используйте рекурсивную функцию с четырьмя параметрами:

1. Количество дисков, которое должно быть перемещено.
2. Колышек, на который эти диски нанизаны первоначально.
3. Колышек, на который эта группа дисков должна быть перемещена.
4. Колышек, используемый как место временного размещения.

Ваша программа должна печатать четкие инструкции, что нужно делать для перемещения дисков с начального колышка на конечный. Например, чтобы передвинуть группу из трех дисков с колышка 1 на колышек 3, ваша программа должна напечатать следующую последовательность перемещений:

$1 \rightarrow 3$  (Это означает перемещение одного диска с колышка 1 на колышек 3)

$1 \rightarrow 2$

$3 \rightarrow 2$

$1 \rightarrow 3$

$2 \rightarrow 1$

$2 \rightarrow 3$

$1 \rightarrow 3$

**3.43.** Любая программа, которая может быть разработана как рекурсивная, может быть разработана и как итеративная, хотя иногда с большими трудностями и меньшей ясностью. Попытайтесь написать итеративную версию задачи о Ханойских башнях. Если вам это удастся, сравните вашу итеративную версию с рекурсивной, разработанной в упражнении 3.42. Исследуйте вопросы производительности, ясности и возможности обосновать корректность программ.

**3.44.** (Визуализация рекурсии) Интересно наблюдать рекурсию «в действии». Модифицируйте функцию факториал на рис. 3.14 так, чтобы печатать ее локальную переменную и параметр рекурсивного вызова. Для каждого рекурсивного вызова отобразите выходные данные в отдельной строке и добавьте отступ. Сделайте все возможное для того, чтобы выходные данные были ясными, интересными и значимыми. Ваша цель — разработать и реализовать такой формат выходных данных, который поможет лучше понять рекурсию. Вы можете добавлять такие изобразительные возможности во многие другие примеры и упражнения по рекурсии в этой книге.

**3.45.** Наибольший общий делитель (НОД) двух целых чисел  $x$  и  $y$  — это наибольшее целое, на которое без остатка делится каждое из двух чисел. Напишите рекурсивную функцию `nod`, которая возвращает наибольший общий делитель чисел  $x$  и  $y$ . НОД для  $x$  и  $y$  определяется рекурсивно следующим образом: если  $y$  равно 0, то `nod(x, y)` возвращает  $x$ ; в противном случае `nod(x, y)` равняется `nod(y, x % y)`, где  $\%$  — это операция вычисления остатка.

**3.46.** Можно ли рекурсивно вызывать функцию `main`? Напишите программу, содержащую функцию `main`. Включите в нее локальную переменную `count` типа `static`, дав ей начальное значение 1. Давайте ей приращение и печатайте значение `count` при каждом вызове `main`. Запустите вашу программу. Что произойдет?

**3.47.** Упражнения с 3.35 по 3.37 посвящены разработке компьютерных программ для обучения слушателей начальных школ умножению. В данном упражнении попытайтесь их усовершенствовать.

а) Модифицируйте программу так, чтобы позволить пользователю ввести уровень своих возможностей. Первый уровень означает, что

в задачах используются только одноразрядные числа, второй уровень означает использование двухразрядных чисел и т. д.

b) Модифицируйте программу так, чтобы позволить пользователю выбрать тип арифметических операций для изучения. Опция 1 пусты означает только операцию сложения, 2 — операцию вычитания, 3 — операцию умножения, 4 — операцию деления, 5 — случайную смесь операций этих типов.

**3.48.** Напишите функцию `distance`, которая вычисляет расстояние между двумя точками  $(x_1, y_1)$  и  $(x_2, y_2)$ . Все числа и возвращаемые значения должны быть типа `float`.

**3.49.** Что делает следующая программа?

```
#include <iostream.h>

main()
{
 int c;

 if ((c = cin.get()) != EOF) {
 main();
 cout << c;
 }
 return 0;
}
```

**3.50.** Что делает следующая программа?

```
#include <iostream.h>

int mystery(int, int);
main()
{
 int x, y;

 cout << "Введите два целых числа: ";
 cin >> x >> y;
 cout << "Результат равен " << mystery(x, y) << endl;
 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);
}
```

**3.51.** После того, как вы определили, что делает программа в упражнении 3.50, преобразуйте программу в соответствующую функцию, удалив ограничение, требующее, чтобы второй аргумент быть неотрицательным.

3.52. Напишите программу, которая проверяет как можно больше математических функций библиотеки на рис. 3.2. Поупражняйтесь с каждой из этих функций, выводя в вашей программе на печать таблицы возвращаемых значений для различных значений аргументов.

3.53. Найдите ошибку в каждом из приведенных ниже фрагментов программ и объясните, как ее исправить:

a) float cube(float); /\* прототип функции \*/  
...  
cube(float number) /\* описание функции \*/  
{  
 return number \* number \* number;  
}  
  
b) register auto int x = 7;  
  
c) int randomNumber = srand();  
  
d) float y = 123.45678;  
int x;  
  
x = y;  
cout << (float) x << endl;  
  
e) double square(double number)  
{  
 double number  
 return number \* number;  
}  
  
f) int sum(int n)  
{  
 if (n == 0)  
 return 0;  
 else  
 return n + sum(n);  
}

3.54. Модифицируйте программу игры в крэпс на рис. 3.10 так, чтобы можно было заключать пари. Оформите в виде функции часть программы, которая моделирует одну игру в крэпс. Присвойте переменной `bankBalance` начальное значение 1000 долларов. Предложите игроку ввести ставку в переменную `wager`. Используя цикл `while`, проверьте, что `wager` не больше, чем `bankBalance`, и, если больше — предложите пользователю повторно ввести `wager` до тех пор, пока не будет введено правильное значение ставки. После введения правильного значения `wager` запустите моделирование одной игры в крэпс. Если игрок выиграл, увеличьте `bankBalance` на `wager` и напечатайте новое значение `bankBalance`. Если игрок проиграл, уменьшите `bankBalance` на `wager`, напечатайте новое значение `bankBalance`, проверьте, не стало ли значение `bankBalance` равным нулю, и если стало, напечатайте сообщение «Извините. Вы обанкротились!». Пока игра продолжается, печатайте различные сообщения, чтобы создать некоторый «разговорный фон» типа: «О! Вы собираетесь разориться, ха!», или «Испытайт судьбу!», или «Вам везет! Теберь самое время обменять фишки на деньги!».

- 3.55. Напишите программу на C++, которая использует встраиваемую функцию `circleArea`, чтобы запросить у пользователя значение радиуса круга, рассчитать и напечатать значение площади этого круга.
- 3.56. Напишите законченную программу на C++ с двумя указанными ниже альтернативными функциями, каждая из которых просто утраивает переменную `count`, определенную в `main`. Затем сравните и противопоставьте эти два подхода. Вот эти две функции:
- Функция `tripleCallByValue`, в которую передается копия `count` по значению, в функции эта копия утраивается и возвращается соответствующее значение.
  - Функция `tripleByReference`, в которую передается `count` по ссылке посредством ссылочного параметра, а функция утраивает исходную копию `count` через ее псевдоним (т.е. ссылочный параметр).
- 3.57. Каково назначение унарной операции разрешения области действия?
- 3.58. Напишите программу, которая использует шаблон функции по имени `min` для определения наименьшего из двух аргументов. Продвигните программу, используя пары целых чисел, символов и чисел с плавающей запятой.
- 3.59. Напишите программу, которая использует шаблон функции по имени `max` для определения наибольшего из двух аргументов. Продвигните программу, используя пары целых чисел, символов и чисел с плавающей запятой.
- 3.60. Определите, содержат ли следующие фрагменты программы ошибки. Для каждой ошибки укажите, как она может быть исправлена. Замечание: в некоторых фрагментах ошибки могут отсутствовать.
- ```
template <class A>
int sum(int num1, int num2, int num3)
{
    return num1 + num2 + num3;
}
```
 - ```
void printResults(int x, int y)
{
 cout << "Сумма равна " << x + y << '\n';
 return x + y;
}
```
  - ```
template <A>
A product(A num1, A num2, A num3)
{
    return num1 * num2 * num3;
}
```
 - ```
double cube(int);
int cube(int);
```

---

г л а в а

---

# 4

## Массивы



### Ц е л и

- Познакомиться со структурой массивов данных.
- Понять, как применять массивы для хранения, сортировки и поиска списков и таблиц значений.
- Понять, как объявлять массив, давать начальные значения элементам массива и ссылаться на отдельные элементы массива.
- Научиться передавать массивы функциям.
- Понять основные способы сортировки.
- Научиться объявлять многомерные массивы и манипулировать ими.

## План

- 4.1. Введение
- 4.2. Массивы
- 4.3. Объявление массивов
- 4.4. Примеры использования массивов
- 4.5. Передача массивов в функции
- 4.6. Сортировка массивов
- 4.7. Учебный пример: вычисление среднего значения, медианы и моды с использованием массивов
- 4.8. Поиск в массивах: линейный поиск и двоичный поиск
- 4.9. Многомерные массивы
- 4.10. Размышления об объектах: идентификация поведений объектов

*Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения • Упражнения на рекурсию*

### 4.1. Введение

Эта глава служит введением в важную тему — структуры данных. *Массивы* — это структуры, содержащие связанные друг с другом однотипные элементы данных. В главе 6 мы обсудим понятия *структур* и *классов*, которые также способны содержать связанные друг с другом элементы данных, причем даже разных типов. Массивы и структуры — «статические» сущности, они сохраняют свои размеры на протяжении всего времени выполнения программы (при этом они, конечно, могут быть объектами автоматического класса хранения и, следовательно, создаваться и ликвидироваться каждый раз при входе и выходе из блоков, в которых они определяются). В главе 15 мы познакомимся с динамическими структурами данных, такими как списки, очереди, стеки и деревья, которые могут увеличиваться и уменьшаться в процессе выполнения программы.

## 4.2. Массивы

Массив — это последовательная группа ячеек памяти, имеющих одинаковое имя и одинаковый тип. Чтобы сослаться на отдельную ячейку или элемент массива, мы указываем имя массива и номер позиции отдельного элемента массива.

На рис. 4.1 показан массив целых чисел, названный `c`. Этот массив содержит 12 элементов. На любой элемент массива можно сослаться, указывая имя массива и номер позиции элемента, заключенный в квадратные скобки (`[ ]`). Первый элемент каждого массива — это *нулевой элемент*. Таким образом, первый элемент массива `c`, указывают как `c[0]`, второй элемент — как `c[1]`, седьмой — как `c[6]` и вообще *i*-ый элемент массива `c` указывают как `c[i-1]`. Имена массивов должны удовлетворять тем же требованиям, которые предъявляются к другим именам переменных.

| Имя массива (заметьте, что все элементы этого массива имеют одно и то же имя, <code>c</code> ) |      |
|------------------------------------------------------------------------------------------------|------|
| <code>c[0]</code>                                                                              | -45  |
| <code>c[1]</code>                                                                              | 6    |
| <code>c[2]</code>                                                                              | 0    |
| <code>c[3]</code>                                                                              | 72   |
| <code>c[4]</code>                                                                              | 1543 |
| <code>c[5]</code>                                                                              | -89  |
| <code>c[6]</code>                                                                              | 0    |
| <code>c[7]</code>                                                                              | 62   |
| <code>c[8]</code>                                                                              | -3   |
| <code>c[9]</code>                                                                              | 1    |
| <code>c[10]</code>                                                                             | 6453 |
| <code>c[11]</code>                                                                             | 78   |

Позиция номера элемента внутри массива `c`

Рис. 4.1. Массив из 12-ти элементов

Номер позиции, указанный внутри квадратных скобок, называется *индексом*. Индекс должен быть целым числом или целым выражением. Если программа использует выражение в качестве индекса, то выражение вычисляется с целью определения индекса. Например, если мы предположим, что переменная `a` равна 5, а переменная `b` равна 6, то оператор `c[a + b] += 2;`

добавляет 2 к элементу массива `c[11]`. Заметим, что имя индексированного массива является целой L-величиной — оно может быть использовано в левой части оператора присваивания.

Проанализируем массив `c` на рис. 4.1 более детально. *Имя* массива — это `c`. Его двенадцать элементов обозначены как `c[0], c[1], c[2],...,c[11]`. Значение `c[0]` равно 45, значение `c[1]` равно 6, значение `c[7]` равно 62, значение `c[11]` равно 78. Чтобы напечатать сумму значений, содержащихся в первых трех элементах массива `c`, мы должны были бы написать

```
cout << c[0] + c[1] + c[2] << endl;
```

Чтобы разделить значение седьмого элемента массива `c` на 2 и присвоить результат переменной `x`, мы должны были бы написать

```
x = c[6] / 2;
```

#### Типичная ошибка программирования 4.1

Важно отметить различие между «седьмым элементом массива» и «элементом массива семь». Поскольку индексы массива начинаются с 0, «седьмой элемент массива» имеет индекс шесть, тогда как «элемент массива семь» имеет индекс 7 и на самом деле является восьмым элементом массива. Это — источник ошибок типа завышения (или занижения) на единицу.

Квадратные скобки, внутри которых записывается индекс массива, на самом деле рассматриваются в C++ как операция индексации. Квадратные скобки имеют тот же уровень старшинства, что и круглые скобки. Таблица на рис. 4.2 показывает приоритеты и ассоциативность тех операций, с которыми мы уже познакомились к настоящему моменту. Операции расположены сверху вниз в порядке понижения старшинства с указанием их ассоциативности и типа.

| Операции                           | Ассоциативность | Тип операций           |
|------------------------------------|-----------------|------------------------|
| <code>() []</code>                 | слева направо   | наивысший              |
| <code>+ -- ! (тип)</code>          | справа налево   | унарные                |
| <code>* / %</code>                 | слева направо   | мультипликативные      |
| <code>+ -</code>                   | слева направо   | аддитивные             |
| <code>&lt;&lt; &gt;&gt;</code>     | слева направо   | поместить в/взять из   |
| <code>&lt; &lt;= &gt; &gt;=</code> | слева направо   | отношение              |
| <code>== !=</code>                 | слева направо   | проверка на равенство  |
| <code>&amp;&amp;</code>            | слева направо   | логическое И           |
| <code>  </code>                    | слева направо   | логическое ИЛИ         |
| <code>?:</code>                    | справа налево   | условная               |
| <code>= += -= *= /= %=</code>      | справа налево   | присваивание           |
| <code>,</code>                     | слева направо   | запятая (последование) |

**Рис. 4.2.** Приоритеты и ассоциативность операций

## 4.3. Объявление массивов

Массивы занимают область в памяти. Программист указывает тип каждого элемента, количество элементов, требуемое каждым массивом, и компилятор может зарезервировать соответствующий объем памяти. Чтобы указать компилятору зарезервировать память для 12 элементов массива целых чисел **c**, используется объявление

```
int c[12];
```

Память может быть зарезервирована для нескольких массивов с помощью одного объявления. Следующее объявление резервирует память для 100 элементов массива целых чисел **b** и 27 элементов массива целых чисел **x**.

```
int b[100], x[27];
```

Массивы могут быть объявлены и для хранения других типов данных. Например, для хранения строки символов можно использовать массив типа **char**. Строки символов и их схожесть с массивами, а также соотношение между указателями и массивами обсуждаются в главе 5.

## 4.4. Примеры использования массивов

В программе на рис. 4.3 используется структура повторения **for** для присвоения начальных нулевых значений элементам массива **n**, содержащего десять целых чисел, и для печати массива в табулированном формате. Первый оператор вывода выводит на экран заголовки столбцов, печатаемых структурой **for**. Напомним, что **setw** указывает ширину поля, в котором будет выведено *следующее* значение.

```
// присваивание массиву начальных значений

#include <iostream.h>
#include <iomanip.h>

main()
{
 int n[10];
 for (int i = 0; i < 10; i++) // присваивание массиву
 // начальных значений
 n[i] = 0;

 cout << "Элемент" << setw(13) << "Значение" << endl;

 for(i = 0; i < 10; i++) // печать массива
 cout << setw(7) << i << setw(13) << n[i] << endl;
 return 0;
}
```

Рис. 4.3. Присваивание нулевых начальных значений элементам массива (часть 1 из 2)

| Элемент | Значение |
|---------|----------|
| 0       | 0        |
| 1       | 0        |
| 2       | 0        |
| 3       | 0        |
| 4       | 0        |
| 5       | 0        |
| 6       | 0        |
| 7       | 0        |
| 8       | 0        |
| 9       | 0        |

Рис. 4.3. Присваивание нулевых начальных значений элементам массива (часть 2 из 2)

Элементам массива можно присваивать начальные значения (*инициализировать их*) в объявлении массива с помощью следующего за объявлением списка (заключенного в фигурные скобки) одинаковых по смыслу и разделенных запятыми начальных значений. Программа на рис. 4.4 присваивает десять начальных значений элементам массива целых чисел и печатает массив в табулированном формате.

Если начальных значений меньше, чем элементов в массиве, оставшиеся элементы автоматически получают нулевые начальные значения. Например, элементам массива *n* на рис. 4.3 можно присвоить нулевые начальные значения с помощью объявления

```
int n[10] = {0};
```

которое явно присваивает нулевое начальное значение первому элементу и неявно присваивает нулевые начальные значения оставшимся девяти элементам, потому что здесь начальных значений меньше, чем элементов массива. Напомним, что автоматически массивы не получают нулевые начальные значения неявно. Программист должен присвоить нулевое начальное значение по крайней мере первому элементу для того, чтобы автоматически были обнулены оставшиеся элементы. Метод, использованный на рис. 4.3, можно применять повторно во время выполнения программы.

### Типичная ошибка программирования 4.2

Забывают присвоить начальные значения элементам массива, требующим такого присваивания.

```
// Присваивание начальных значений в объявлении массива
#include <iostream.h>
#include <iomanip.h>

main()
{
 int n[10] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};

 cout << "Элемент" << setw(13) << "Значение" << endl;
 for(int i = 0; i < 10; i++)
 cout << setw(7) << i << setw(13) << n[i] << endl;
 return 0;
}
```

Рис. 4.4. Присваивание начальных значений элементам массива в объявлении (часть 1 из 2)

| Элемент | Значение |
|---------|----------|
| 0       | 32       |
| 1       | 27       |
| 2       | 64       |
| 3       | 18       |
| 4       | 95       |
| 5       | 14       |
| 6       | 90       |
| 7       | 70       |
| 8       | 60       |
| 9       | 37       |

Рис. 4.4. Присваивание начальных значений элементам массива в объявлении (часть 2 из 2)

#### Следующее объявление массива

```
int n[5] = {32, 27, 64, 18, 95, 14};
```

вызвало бы синтаксическую ошибку, поскольку здесь список инициализации содержит шесть начальных значений, а массив имеет только 5 элементов.

#### Типичная ошибка программирования 4.3

Задание в списке начальных значений большего числа значений, чем имеется элементов в массиве, является синтаксической ошибкой.

Если размер массива не указан в объявлении со списком инициализации, то количество элементов массива будет равно количеству элементов в списке начальных значений. Например, объявление

```
int n[] = {1, 2, 3, 4, 5};
```

создало бы массив из пяти элементов.

Программа на рис. 4.5 присваивает начальные целые значения 2, 4, 6, ..., 20 элементам массива s из десяти элементов и печатает массив в табулированном формате. Эти значения генерируются путем умножения каждого последующего значения счетчика цикла на 2 и прибавления 2.

```
// Присваивание элементам массива четных начальных значений
// от 2 до 20.
#include <iostream.h>
#include <iomanip.h>

main()
{
 const int arraySize = 10;
 int s[arraySize];

 for (int j = 0; j < arraySize; j++) // задание значений
 s[j] = 2 + 2 * j;

 cout << "Элемент" << setw(13) << "Значение" << endl;
 for(j = 0; j < arraySize; j++) // печать значений
 cout << setw(7) << j << setw(13) << s[j] << endl;
 return 0;
}
```

Рис. 4.5. Генерация значений, размещаемых в элементах массива (часть 1 из 2)

| Элемент | Значение |
|---------|----------|
| 0       | 2        |
| 1       | 4        |
| 2       | 6        |
| 3       | 8        |
| 4       | 10       |
| 5       | 12       |
| 6       | 14       |
| 7       | 16       |
| 8       | 18       |
| 9       | 20       |

Рис. 4.5. Генерация значений, размещаемых в элементах массива (часть 2 из 2)

### Строка

```
const int arraySize = 10;
```

использует спецификацию `const` для объявления так называемой *постоянной переменной* `arraySize`, имеющей значение 10. Постоянные переменные должны получать при объявлении в качестве начальных значений постоянные выражения, которые после этого не могут быть модифицированы (рис. 4.6 и рис. 4.7). Постоянные переменные называются также *именованными константами*, или *переменными только для чтения*, или *типовыми константами* (однако, это может вызвать путаницу, поскольку в других языках, например, в Паскале, типизированная переменная — это нечто совершенно иное). Мы остановимся на термине *именованная константа*. Заметим, что термин «постоянная переменная» является оксюмороном — противоречием терминов, подобным «большой малютке» или «ледяному ожогу». (Пожалуйста, пришлите нам ваши любимые оксюмороны по электронной почте, указанной в предисловии. Спасибо!).

### Типичная ошибка программирования 4.4

Присваивание значения именованной константе в выполняемом операторе является синтаксической ошибкой.

```
//Постоянный объект должен получить начальное значение
main()
{
 const int x; // Ошибка: x должен получить
 // начальное значение
 x = 7; // Ошибка: постоянный объект (именованную
 // константу) изменять нельзя
 return 0;
}
```

Compiling FIG4\_6.CPP:

Error FEG4\_6.CPP 4: Constant variable 'x' must be  
initialized

Error FEG4\_6.CPP 6: Cannot modify a constant object

Рис. 4.6. Постоянный объект должен получить начальное значение

```
// Использование именованной константы, получившей
// соответствующее начальное значение
#include <iostream.h>

main()
{
 const int x = 7; // присваивание начального значения
 // именованной константе
 cout << "Значение именованной константы x равно: "
 << x << '\n';
 return 0;
}
```

---

Значение именованной константы x равно: 7

**Рис. 4.7.** Правильное присваивание начального значения и использование именованной константы

Именованные константы могут быть помещены в любом месте, где ожидается постоянное выражение. На рис. 4.5 именованная константа `arraySize` используется для задания размера массива `s` в объявлении

```
int s[arraySize];
```

При объявлении автоматических и статических массивов можно использовать только константы.

Использование именованных констант для задания размеров массивов делает программу более масштабируемой. На рис. 4.5 первый цикл `for` мог бы заполнить 1000-элементный массив, если просто изменить значение `arraySize` в его объявлении с 10 на 1000. Если бы мы не использовали именованную константу `arraySize`, мы должны были бы изменить программу в трех разных местах, чтобы масштабировать программу для обработки массива из 1000 элементов. В больших программах эта техника становится более полезной для написания понятных программ.

### Замечание по технике программирования 4.1

Определение размера каждого массива с помощью именованной константы делает программу более масштабируемой.

### Типичная ошибка программирования 4.5

Завершение директивы препроцессора `#include` точкой с запятой. Помните, что директивы препроцессора не являются операторами C++.

Программа на рис. 4.8 суммирует значения, которые содержатся в массиве `a` из 12 целых чисел. Оператор в теле цикла `for` осуществляет суммирование. Важно помнить, что значения, используемые в качестве начальных для массива `a`, должны были бы вводиться в программу пользователем с клавиатуры. Например, следующая структура `for`

```
for (int j = 0; j < arraySize; j++)
 cin >> a[j];
```

считывает значения одно за одним с клавиатуры и запоминает их в элементах `a`.

```
// Вычисление суммы элементов массива

#include <iostream.h>

main()
{
 const int arraySize = 12;
 int a [arraySize] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45};
 int total = 0;

 for (int i = 0; i < arraySize; i++)
 total += a[i];

 cout << "Сумма значений элементов массива равна "
 << total << endl;
 return 0;
}
```

**Сумма значений элементов массива равна 383**

**Рис. 4.8.** Вычисление суммы элементов массива

Наш следующий пример использует массивы для суммирования результатов данных, полученных при опросе. Рассмотрим следующую постановку задачи:

*К сорока студентам обратились с просьбой оценить качество пищи в студенческом кафетерии по десятибалльной шкале (1 означает отвратительное качество, а 10 означает отличное). Поместите сорок ответов в массив целых чисел и просуммируйте результаты опроса.*

Это типичное применение массива (см. рис. 4.9.). Мы хотим просуммировать количество ответов каждого типа (т.е. от 1 до 10). Массив `responses` — это массив с 40 элементами, предназначенный для ответов студентов. Мы используем массив с 11 элементами `frequency` для подсчета числа каждого из ответов. Мы игнорируем первый элемент `frequency[0]`, потому что более логично накапливать ответ 1 в элементе `frequency[1]`, чем в `frequency[0]`. Это позволяет нам использовать каждый ответ непосредственно как индекс массива `frequency`.

### Хороший стиль программирования 4.1

Старайтесь программировать понятно. Иногда имеет смысл пожертвовать более высокой эффективностью использования памяти или процессорного времени в пользу написания более понятной программы.

### Совет по повышению эффективности 4.1

Иногда соображения эффективности далеко перевешивают соображения понятности.

```
// Программа опроса студентов
#include <iostream.h>
#include <iomanip.h>

main()
{
 const int responseSize = 40, frequencySize = 11;
 int responses[responseSize] = {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[frequencySize] = {0};

 for (int answer = 0; answer < responseSize; answer++)
 ++frequency[responses[answer]];

 cout << "Рейтинг" << setw(17) << "Частота" << endl;

 for (int rating = 1; rating < frequencySize; rating++)
 cout << setw(6) << rating
 << setw(17) << frequency[rating] << endl;
}

return 0;
}
```

| Рейтинг | Частота |
|---------|---------|
| 1       | 2       |
| 2       | 2       |
| 3       | 2       |
| 4       | 2       |
| 5       | 5       |
| 6       | 11      |
| 7       | 5       |
| 8       | 7       |
| 9       | 1       |
| 10      | 3       |

Рис. 4.9. Программа простого анализа опроса студентов

Первый цикл **for** берет ответы один за одним из массива **response** и дает приращение одному из десяти счетчиков (от **frequency[1]** до **frequency[10]**) в массиве **frequency**. Ключевым оператором цикла служит

```
++frequency[responses[answer]];
```

Этот оператор увеличивает соответствующий счетчик **frequency** в зависимости от значения **responses[answer]**. Например, если счетчик **answer** равен 0, то элемент **responses[answer]** равен 1, так что **++frequency[responses[answer]]**; на самом деле интерпретируется как оператор

```
++frequency[1];
```

который увеличивает элемент массива один. Если **answer** равен 1, **responses[answer]** равен 2, так что **++frequency[responses[answer]]**; интерпретируется как оператор

```
++frequency[2];
```

который увеличивает элемент массива два. Если `answer` равен 2, `responses[answer]` равен 6, так что `++frequency[responses[answer]]`; интерпретируется как оператор

```
++frequency[6];
```

который увеличивает элемент массива шесть и т.д. Заметим, что независимо от числа ответов, полученных в опросе, для суммирования результатов достаточно лишь массива с одиннадцатью элементами (при игнорировании элемента нуль). Если данные содержат неверные значения, такие как 13, программа будет пытаться прибавить 1 к `frequency[13]`. Это выходило бы за границы массива. C++ не проверяет границ массивов, чтобы предупредить компьютер о ссылках на несуществующие элементы. Таким образом, выполняемая программа может выйти за пределы любого конца массива без предупреждения. Программист должен иметь гарантию, что все ссылки на массив находятся внутри границ массива. C++ — гибкий язык. В главе 8 C++ будет расширен реализацией массива как определяемого пользователем типа с помощью классов. Наше новое определение массива позволит выполнять многие операции, которые не являются стандартом для массивов, встроенных в C++. Например, мы сможем непосредственно сравнивать массивы, присваивать один массив другому, вводить и выводить массивы целиком с помощью `cin` и `cout`, автоматически присваивать массивам начальные значения, предотвращать доступ к элементам массива за границами массива и изменять диапазон индексов, так чтобы не требовалось, чтобы первый элемент массива был элемент 0.

### Типичная ошибка программирования 4.6

Ссылка на элемент, находящийся вне границ массива.

### Хороший стиль программирования 4.2

При использовании циклов с массивом индекс массива никогда не должен быть меньше нуля и должен всегда быть меньше, чем общее количество элементов массива (меньше, чем размер массива). Убедитесь, что условия завершения цикла предотвращают обращение к элементам, выходящим за пределы этого диапазона.

### Хороший стиль программирования 4.3

В программах должна быть обеспечена правильность всех вводимых значений, чтобы не допустить возможность влияния ошибочной информации на ход вычислений в программе.

### Замечание по мобильности 4.1

Эффекты (обычно серьезные) ссылок на элементы, выходящие за границы массива, системно зависимы.

В нашем следующем примере (рис. 4.10) числа читаются из массива и графически представляются в форме таблицы полосок или гистограммы — печатается каждое число, а затем строка, содержащая столько же звездочек. Строки из звездочек рисуются с помощью вложенного цикла `for`. Обратите внимание на использование `endl`, чтобы закончить гистограмму.

```
// Программа печати гистограммы
#include <iostream.h>
#include <iomanip.h>

main()
{
 const int arraySize = 10;
 int n[arraySize] = {19, 3, 15, 7, 11, 9, 13, 5, 17, 1};

 cout << "Элемент" << setw(13) << "Значение"
 << setw(17) << "Гистограмма" << endl;
 for (int i = 0; i < arraySize ; i++) {
 cout << setw(7) << i << setw(13) << n[i] << " ";
 for(int j = 1; j <= n[i]; j++) // печать одной строки
 cout << '*';
 cout << endl;
 }
 return 0;
}
```

| Элемент | Значение | Гистограмма |
|---------|----------|-------------|
| 0       | 19       | *****       |
| 1       | 3        | ***         |
| 2       | 15       | *****       |
| 3       | 7        | *****       |
| 4       | 11       | *****       |
| 5       | 9        | *****       |
| 6       | 13       | *****       |
| 7       | 5        | *****       |
| 8       | 17       | *****       |
| 9       | 1        | *           |

Рис. 4.10. Программа печати гистограммы

В главе 3 мы обещали, что покажем более элегантный метод написания программы игры в кости, приведенной на рис. 3.8. Проблема заключалась в том, чтобы сделать 6000 бросков шестигранной кости для проверки, вырабатывает ли генератор случайных чисел действительно случайные числа. Вариант программы с использованием массивов показан на рис. 4.11.

До сих пор мы обсуждали только массивы целых чисел. Однако, массивы могут быть и других типов. Теперь мы обсудим хранение строк в массивах символов. До сих пор единственной возможностью обработки строки, с которой мы были знакомы, был вывод строки с помощью `cout` и `<<`. Стока, например, "hello", на самом деле является массивом символов. Массив символов имеет несколько специфических особенностей.

Массиву символов можно задать начальные значения, используя лiteralную константу. Например, объявление

```
char string1[] = "first";
```

присваивает элементам массива `string1` в качестве начальных значений отдельные символы строки "first". Размер массива `string1` в предыдущем объявлении, определяется компилятором на основе длины строки. Важно заметить, что строка "first" содержит пять символов плюс специальный символ оконча-

ния строки, называемый *нулевым символом*. Таким образом, массив `string1` на самом деле содержит шесть элементов. Нуевой символ представляется символьной константой '\0'. Все строки заканчиваются этим символом. Символьный массив, представляющий строку, должен всегда объявляться достаточно большим для того, чтобы в него можно было поместить количество символов в строке и завершающий нулевой символ.

Символьному массиву можно также задать в качестве начального значения список отдельных символьных констант, указанных в списке инициализации. Например, предыдущее объявление эквивалентно следующему

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' }
```

Поскольку строка является массивом символов, мы можем получить доступ к отдельным символам строки, используя индексную запись массива. Например, `string1[0]` — это символ 'f', а `string1[3]` — это символ 's'.

```
// Бросание шестигранной кости 6000 раз

#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

main()
{
 const int arraySize = 7;
 int face, frequency[arraySize] = {0};

 srand(time(NULL));

 for (int roll = 1; roll <= 6000; roll++) {
 face = rand() % 6 + 1;
 ++frequency[face]; // замена 20-строчного
 // оператора switch на рис. 3.10
 }

 cout << "Грань" << setw(17) << "Частота" << endl;

 for (face = 1; face < arraySize ; face++) // игнорируется
 // элемент 0
 cout << setw(4) << face
 << setw(17) << frequency[face] << endl;

 return 0;
}
```

---

| Грань | Частота |
|-------|---------|
| 1     | 1037    |
| 2     | 987     |
| 3     | 1013    |
| 4     | 1028    |
| 5     | 952     |
| 6     | 983     |

Рис. 4.11. Программа бросания кости, использующая массивы вместо `switch`

Мы также можем ввести строку непосредственно в символьный массив с клавиатуры, используя `cin` и `>>`. Например, объявление

```
char string2[20];
```

создает символьный массив, способный хранить строку из 19 символов и завершающий нулевой символ. Оператор

```
cin >> string2;
```

считывает строку с клавиатуры в `string2`. Обратите внимание, что в предыдущем операторе в `cin` использовано только имя массива и нет никакой информации о его размере. В обязанности программиста входит убедиться в том, что массив, в который записывается строка, способен вместить любую строку, которую пользователь напечатает на клавиатуре. `cin` читает символы с клавиатуры до тех пор, пока не встретится первый разделитель в тексте, и при этом не заботится о размерах массива. Таким образом `cin` может вставлять данные и после конца массива (смотри раздел 5.12, где обсуждается вопрос о предотвращении вставки после конца символьного массива).

### Типичная ошибка программирования 4.7

Недостаточный размер массива, в который с помощью `cin` вводится символьная строка с клавиатуры, может привести к потере данных в программе и к другим ошибкам выполнения.

Символьный массив, представляющий строку, может быть выведен с помощью `cout` и `<<`. Массив `string2` печатается оператором

```
cout << string2 << endl;
```

Заметим, что `cout`, подобно `cin`, не заботится о размерах символьного массива. Символы строки печатаются до тех пор, пока не встретится завершающий нулевой символ.

Рисунок 4.12 демонстрирует присваивание начальных условий символьному массиву с помощью литеральной константы, чтение строки в символьный массив, печать символьного массива как строки и доступ к отдельным символам строки.

В программе рис. 4.12 использована структура `for` для циклической обработки массива `string1` и печати отдельных символов, разделенных пробелами. Условие `string1[ i ] != '\0'` в цикле `for` выполняется до тех пор, пока в строке не встретится завершающий нулевой символ.

В главе 3 обсуждалась спецификация класса памяти `static`. Локальная переменная класса `static` в описании функции существует в течение всего времени выполнения программы, но она видима только в теле функции. Мы можем применить `static` при объявлении локального массива, чтобы не создавать его и не задавать ему начальных условий при каждом вызове функции и не уничтожать его при каждом выходе из функции.

Массивы, объявленные как `static`, получают начальные значения при загрузке программы. Если массив `static` не получает начальные значения явно, от программиста, то компилятор присваивает ему нулевые начальные значения.

```
// Представление символьного массива как строки
#include <iostream.h>

main()
{
 char string1[20], string2[] = "литеральная константа";

 cout << "Введите строку: ";
 cin >> string1;

 cout << "строка1 равна: " << string1 << endl
 << "строка2 равна: " << string2 << endl
 << "строка1 с пробелами между символами равна: "
 << endl;

 for (int i = 0; string1[i] != '\0'; i++)
 cout << string1[i] << ' ';

 cout << endl;
 return 0;
}
```

---

```
Введите строку: Привет
строка1 равна: Привет
строка2 равна: литеральная константа
строка1 с пробелами между символами равна:
Привет
```

**Рис. 4.12.** Представление символьного массива как строки

Рис. 4.13 демонстрирует функцию `staticArrayInit` с локальным массивом, объявленным как `static`, и функцию `automaticArrayInit` с автоматическим локальным массивом. Функция `staticArrayInit` вызывается дважды. Локальный массив класса `static` получает начальные условия, равные нулю, от компилятора. Функция печатает массив, добавляет 5 к каждому элементу и печатает массив снова. При повторном вызове функции массив класса `static` содержит значения, запомненные при первом вызове функции. Функция `automaticArrayInit` также вызывается дважды. Элементы автоматического локального массива получают начальные значения 1, 2 и 3. Функция печатает массив, добавляет 5 к каждому элементу и печатает массив снова. При повторном вызове функции элементы массива снова получают начальные значения 1, 2 и 3, потому что массив имеет автоматический класс памяти.

#### Типичная ошибка программирования 4.8

Предположение, что элементы локального массива класса `static` в функции получают нулевые начальные значения при каждом вызове функции.

```
// Массивы типа static получают нулевые начальные условия
#include <iostream.h>

void staticArrayInit(void);
void automaticArrayInit(void);

main()
{
 cout << "Первый вызов каждой функции:" << endl;
 staticArrayInit();
 automaticArrayInit();

 cout << endl << endl << "Второй вызов каждой функции:" << endl;
 staticArrayInit();
 automaticArrayInit();

 return 0;
}

// функция, демонстрирующая статический локальный массив
void staticArrayInit(void)
{
 static int array1[3];

 cout << endl << "Значения при входе в staticArrayInit:" << endl;

 for (int i = 0; i < 3; i++)
 cout << "массив1[" << i << "] = " << array1[i] << " ";

 cout << endl << "Значения при выходе из staticArrayInit:" << endl;

 for (i = 0; i < 3; i++)
 cout << "массив1[" << i << "] = " << (array1[i] += 5) << " ";
}

// функция для демонстрации автоматического локального массива
void automaticArrayInit(void)
{
 int array2[3] = {1, 2, 3};

 cout << endl << endl
 << "Значения при входе в automaticArrayInit: " << endl;

 for (int i = 0; i < 3; i++)
 cout << "массив2[" << i << "] = " << array2[i] << " ";

 cout << endl << "Значения при выходе из automaticArrayInit: " << endl;

 for (i = 0; i < 3; i++)
 cout << "массив2[" << i << "] = "
 << (array2[i] += 5) << " ";
}
```

Рис. 4.13. Сравнение присваивания начальных значений массиву типа **static** и автоматическому массиву (часть 1 из 2)

Первый вызов каждой функции:

```
Значения при входе в staticArrayInit:
array1[0] = 0 array1[1] = 0 array1[2] = 0
Значения при выходе из staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5
```

```
Значения при входе в automaticArrayInit:
array2[0] = 1 array2[1] = 2 array2[2] = 3
Значения при выходе из automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8
```

Второй вызов каждой функции:

```
Значения при входе в staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5
Значения при выходе из staticArrayInit:
array1[0] = 10 array1[1] = 10 array1[2] = 10
```

```
Значения при входе в automaticArrayInit:
array2[0] = 1 array2[1] = 2 array2[2] = 3
Значения при выходе из automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8
```

Рис. 4.13. Сравнение присваивания начальных значений массиву типа **static** и автоматическому массиву (часть 2 из 2)

## 4.5. Передача массивов в функции

Чтобы передать массив в качестве аргумента в функцию, укажите имя массива без каких-либо квадратных скобок. Например, если массив `hourlyTemperatures` объявлен как

```
int hourlyTemperatures[24];
```

то оператор вызова функции

```
modifyArray(hourlyTemperatures, 24);
```

передает массив `hourlyTemperatures` и его размер функции `modifyArray`. При передаче массива в функцию его размер часто передается, чтобы функция могла обрабатывать заданное число элементов в массиве. В главе 8, когда мы познакомимся с классом `Array`, мы будем встраивать размер массива внутрь типа, определяемого пользователем — каждый объект `Array`, который мы создадим, будет «знать» свой собственный размер. Таким образом, при передаче объекта `Array` функции нет необходимости передавать ей в качестве аргумента размер массива.

C++ автоматически передает массивы функциям, используя моделируемый вызов по ссылке — вызываемые функции могут изменять значения элементов в исходных массивах источника вызова. Значение имени массива является адресом первого элемента массива. Поскольку в функцию передается начальный адрес массива, вызываемая функция знает, где хранится массив. Поэтому, когда вызываемая функция модифицирует элементы массива в теле функции, она модифицирует реальные элементы массива в их истинных ячейках памяти.

## Совет по повышению эффективности 4.2

Передача массивов с помощью моделируемого вызова по ссылке ощутимо влияет на производительность. Если бы массивы передавались по значению, передавалась бы копия каждого элемента. Для больших, часто передаваемых массивов это привело бы к значительному потреблению времени и памяти для хранения копий массивов.

## Замечание по технике программирования 4.2

Передавать массив по значению можно, используя простой прием, который мы объясним в главе 6.

Хотя массивы как целое передаются моделируемым вызовом по ссылке, отдельные элементы массива передаются вызовом по значению подобно простым переменным. Такие отдельные простые элементы данных называются *скалярами* или *скалярными параметрами*. Чтобы передать в функцию элемент массива, используйте индексированное имя элемента массива как аргумент в вызове функции. В главе 5 мы покажем, как можно моделировать вызов по ссылке для скаляров (т.е. отдельных переменных и элементов массива).

Чтобы функция могла принять массив, переданный через вызов функции, список параметров функции должен указывать на необходимость принятия массива. Например, заголовок функции `modifyArray` мог бы быть записан как

```
void modifyArray(int b[], int arraySize)
```

указывая, что `modifyArray` ожидает принятия массива целых чисел в параметре `b`, и количество элементов массива в параметре `arraySize`. Размер массива в квадратных скобках указывать не нужно. Если он включен, компилятор его проигнорирует. Поскольку массивы передаются моделируемым вызовом по ссылке, то вызываемая функция, используя имя массива `b`, в действительности будет работать с истинным массивом в источнике вызова (массив `hourlyTemperatures` в предыдущем вызове). В главе 5 мы познакомимся с другими формами записи, указывающими, что функция принимает массив. Как мы увидим, эти записи основаны на тесных связях между массивами и указателями.

Отметим необычный вид прототипа функции для `modifyArray`

```
void modifyArray (int [], int);
```

Этот прототип мог бы быть записан в виде

```
void modifyArray (int anyArrayName[], int anyVariableName)
```

но, как мы узнали в главе 3, компиляторы C++ игнорируют имена переменных в прототипах.

## Хороший стиль программирования 4.4

Некоторые программисты, чтобы сделать программу понятнее, включают имена переменных в функции прототипов. Компиляторы игнорируют эти имена.

Напомним, что прототип сообщает компилятору количество аргументов и типы каждого аргумента (в порядке их ожидаемого появления).

Программа на рис. 4.14 демонстрирует различие между передачей всего массива и его отдельных элементов. Программа сначала печатает пять элементов массива целых чисел *a*. Далее массив *a* и его размеры передаются функции *modifyArray*, где каждый элемент массива *a* умножается на 2. Затем *a* повторно печатается в *main*. Как показывают выходные данные, элементы *a* действительно модифицируются функцией *modifyArray*. Затем программа печатает значение *a[3]* и передает его функции *modifyElement*. Функция *modifyElement* умножает свой аргумент на 2 и печатает его новое значение. Заметим, что при повторной печати элемента *a[3]* в *main* он не модифицировался, потому что отдельные элементы массива передаются вызовом по значению.

В ваших программах возможна ситуация, когда функции нельзя позволять модифицировать элементы массива. Поскольку массивы всегда передаются моделируемым вызовом по ссылке, модификации значений в массиве управлять затруднительно. C++ имеет спецификацию типа *const* для предотвращения модификации значений массива в функции. Когда параметр массив предварен спецификатором *const*, элементы массива в теле функции становятся постоянными и любая попытка модифицировать элементы массива в теле функции приводит к ошибке при трансляции. Сообщение об этой ошибке дает программисту возможность исправить программу так, чтобы она не пыталась модифицировать элементы массива.

Программа на рис. 4.15 демонстрирует спецификацию *const*. Функция *tryToModifyArray* определяется с параметром *const int b[ ]*, который указывает, что массив *b* — постоянный и не может быть модифицирован. В выходных данных показаны сообщения об ошибках, вырабатываемые компилятором Borland C++. Каждая из трех попыток функции модифицировать элементы массива приводит к ошибке компилятора «Постоянный объект не может быть модифицирован». Спецификация *const* будет еще раз обсуждаться в главе 7.

### Замечание по технике программирования 4.3

Для предотвращения модификации исходного массива внутри тела функции можно в определении функции применять к параметру массив спецификатор типа *const*. Это еще один пример принципа наименьших привилегий. Функциям не должно быть позволено модифицировать массивы без крайней необходимости.

```
// Передача функциям массивов и их отдельных элементов
#include <iostream.h>
#include <iomanip.h>

void modifyArray(int [], int); // необычный вид прототипа
void modifyElement(int);

main()
{
 const int arraySize = 5;
 int a[arraySize] = {0, 1, 2, 3, 4};
```

Рис. 4.14. Передача функциям массивов и отдельных элементов массива (часть 1 из 3)

```
cout << "Результаты передачи всего массива по ссылке"
 << endl << endl
 << "Значения исходного массива:" << endl;
for (int i = 0; i < arraySize; i++)
 cout << setw(3) << a[i];

cout << endl;

modifyArray(a, arraySize); //массив, передается по ссылке

cout << "Значения модифицированного массива:" << endl;

for (i = 0; i < arraySize; i++)
 cout << setw(3) << a[i];

cout << endl << endl << endl
 << "Результаты передачи элемента массива по значению:"
 << endl << endl << "Значение a[3] равно "
 << a[3] << endl;

modifyElement(a[3]);

cout << "Значение a[3] равно " << a[3] << endl;

return 0;
}

void modifyArray(int b[], int sizeOfArray)
{
 for (int j = 0; j < sizeOfArray; j++)
 b[j] *= 2;
}

void modifyElement(int e)
{
 cout << "Значение в modifyElement равно "
 << (e *= 2) << endl;
}
```

Рис. 4.14. Передача функциям массивов и отдельных элементов массива (часть 2 из 3)

**Результаты передачи всего массива по ссылке:**

**Значения исходного массива:**

0 1 2 3 4

**Значения модифицированного массива**

0 2 4 6 8

**Результаты передачи элемента массива по значению:**

**Значение a[3] равно 6**

**Значение в modifyElement равно 12**

**Значение a[3] равно 6**

Рис. 4.14. Передача функциям массивов и отдельных элементов массива (часть 3 из 3)

```
//Демонстрация спецификации типа const
#include <iostream.h>

void tryToModifyArray(const int b[])
{
 int a[] = {10, 20, 30};

 cout << a[0] << ' ' << a[1] << ' ' << a[2] << endl;
 return 0;
}

void tryToModifyArray(const int b[])
{
 b[0] /= 2; //ошибка
 b[1] /= 2; //ошибка
 b[2] /= 2; //ошибка
}

Compiling FIG4_15.CPP:
Error FIG4_15.CPP 16: Cannot modify a cost object
Error FIG4_15.CPP 17: Cannot modify a cost object
Error FIG4_15.CPP 18: Cannot modify a cost object
Warning FIG4_15.CPP 19: Parametr 'b' is never used
```

**Рис. 4.15.** Демонстрация спецификации типа const

## 4.6. Сортировка массивов

Сортировка данных (т.е. размещение данных в определенном порядке, таком как возрастание или уменьшение) является одним из наиболее важных применений компьютеров. Банк сортирует все чеки по номеру счета, чтобы в конце каждого месяца можно было подготовить индивидуальные банковские отчеты. Телефонные компании сортируют свои списки счетов по фамилиям, а внутри них — по имени и отчеству, что позволяет легко найти номера телефонов. В сущности, каждая организация должна сортировать некоторые данные, а во многих случаях очень значительные объемы данных. Сортировка данных представляется интересующей проблемой, которая является объектом наиболее интенсивных исследований в области компьютерных наук. В этой главе мы обсудим простейшие известные способы сортировки. В упражнениях и в главе 15 мы введем в рассмотрение более сложные схемы, которые обеспечивают наивысшую производительность.

### Совет по повышению эффективности 4.3

Часто простейшие алгоритмы обеспечивают низкую производительность. Их достоинство лишь в том, что их легко писать, проверять и отлаживать. Однако, часто для получения максимальной производительности необходимы более сложные алгоритмы.

Программа на рис. 4.16 сортирует значения массива *a* из 10 элементов в возрастающем порядке. Используемая при этом техника получила название *пузырьковая сортировка* или *сортировка погружением*, потому что наимень-

шее значение постепенно «всплывает», продвигаясь к вершине (началу) массива, подобно пузырьку воздуха в воде, тогда как наибольшее значение погружается на дно (конец) массива. Этот прием требует нескольких проходов по массиву. При каждом проходе сравнивается пара следующих друг за другом элементов. Если пара расположена в возрастающем порядке или элементы одинаковы, то мы оставляем значения как есть. Если же пара расположена в убывающем порядке, значения меняются местами в массиве.

Сначала программа сравнивает  $a[0]$  и  $a[1]$ , затем  $a[1]$  и  $a[2]$ , потом  $a[2]$  и  $a[3]$  и т.д. до тех пор, пока проход не закончится сравнением  $a[8]$  и  $a[9]$ . Хотя элементов 10, производится только девять сравнений. При выбранном способе последовательных сравнений большое значение может перемещаться в массиве вниз на много позиций за один проход, но малое значение может быть передвинуто вверх только на одну позицию. При первом проходе наибольшее значение гарантированно опустится на место нижнего элемента массива  $a[9]$ . При втором проходе второе наибольшее значение гарантированно опустится на место  $a[8]$ . При девятом проходе девятое наибольшее значение опустится на место  $a[1]$ . Это оставляет наименьшему значению место  $a[0]$ , так что для сортировки массива из 10 элементов нужно только девять проходов.

Сортировка выполняется с помощью вложенного цикла `for`. Если необходима перестановка, она выполняется тремя присваиваниями

```
hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

где дополнительная переменная `hold` временно хранит одно из двух переставляемых значений. Перестановку нельзя выполнить двумя присваиваниями

```
a[i] = a[i + 1];
a[i + 1] = a[i];
```

Если, например,  $a[i]$  равно 7, а  $a[i + 1]$  равно 5, то после первого присваивания оба значения будут 5, а значение 7 будет потеряно. Следовательно, необходима дополнительная переменная `hold`.

Главное достоинство пузырьковой сортировки заключается в простоте ее программирования. Однако, пузырьковая сортировка выполняется медленно. Это становится очевидным при сортировке больших массивов. В упражнениях мы разработаем более эффективные варианты пузырьковой сортировки и введем некоторые гораздо более эффективные, чем пузырьковый, способы сортировки. В более серьезных курсах по методам вычислений сортировка и поиск изучаются более углубленно.

```
// Эта программа сортирует значения массива в возрастающем порядке
#include <iostream.h>
#include <iomanip.h>

main()
{
 const int arraySize = 10;
 int a[arraySize] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
 int hold;

 cout << "Элементы данных в исходном порядке" << endl;
```

**Рис. 4.16.** Пузырьковая сортировка массива (часть 1 из 2)

```

for (int i = 0; i < arraySize; i++)
 cout << setw(4) << a[i];

for (int pass = 1; pass < arraySize; pass++) // проход

 for (i = 0; i < arraySize - 1; i++) // один проход

 if (a[i] > a[i + 1]) { // одно сравнение
 hold = a[i];
 a[i] = a[i + 1];
 a[i + 1] = hold;
 }

 cout << endl << "Элементы данных в порядке возрастания "
 << endl;

 for (i = 0; i < arraySize; i++)
 cout << setw(4) << a[i];

 cout << endl;
 return 0;
}

```

**Элементы данных в исходном порядке**  
2 6 4 8 10 12 89 68 45 37  
**Элементы данных в порядке возрастания**  
2 4 6 8 10 12 37 45 68 89

Рис. 4.16. Пузырьковая сортировка массива (часть 2 из 2)

## 4.7. Учебный пример: вычисление среднего значения, медианы и моды с использованием массивов

Рассмотрим теперь более серьезный пример. Часто компьютеры используются для обработки и анализа результатов обследований и голосований. Программа на рис. 4.17 использует массив `response`, которому в качестве начальных значений присвоено 99 (размер представлен именованной константой `responseSize`) значений ответов при обследовании. Каждый ответ является числом от 1 до 9. Программа вычисляет среднее значение, медиану и моду 99 значений.

Среднее — это среднее арифметическое 99 значений. Функция `mean` вычисляет среднее путем суммирования 99 элементов и деления результата на 99.

Медиана — это «середина». Функция `median` определяет медиану, вызывая функцию `bubbleSort`, чтобы отсортировать массив ответов в порядке их возрастания, и выбирая средний элемент `answer[responseSize / 2]` отсортированного массива. Заметим, что если массив содержит четное число элементов, медиана должна вычисляться как среднее значение двух элементов в середине массива. Функция `median` этой возможности не обеспечивает. Функция `printArray` вызывается для вывода на экран массива `response`.

Мода — это наиболее часто встречающееся значение среди 99 ответов. Функция `mode` определяет моду, подсчитывая количество ответов каждого типа и выделяя затем наиболее часто встречающийся ответ. Этот вариант функции `mode` не обрабатывает случай равного числа нескольких ответов (см. упражнение 4.14). Функция `mode` строит также гистограмму, чтобы помочь определить моду графически. Рис. 4.18 содержит выходную информацию прогона программы. Этот пример включает наиболее общие операции, необходимые обычно при работе с массивами, включая передачу массивов функциям.

```
// Эта программа знакомит с вопросами анализа данных обследования
// Она вычисляет среднее значение, медиану и моду данных.

#include <iostream.h>
#include <iomanip.h>

void mean(const int [], int);
void median(const int [], int);
void mode(const int [], int [], int);
void bubbleSort(int[], int);
void printArray(const int[], int);

main()
{
 const int responseSize = 99;
 int frequency[10] = {0},
 response[responseSize] = {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, responseSize);
 median(response, responseSize);
 mode(frequency, response, responseSize);

 return 0;
}

void mean(const int answer[], int arraySize)
{
 int total = 0;

 cout << "*****" << endl << " Среднее" << endl
 << "*****" << endl;

 for (int j = 0; j < arraySize; j++)
 total += answer[j];
}
```

Рис. 4.17. Программа анализа данных обследования (часть 1 из 3)

```

cout << "Среднее является средним значением " << endl
 << "элементов данных. Среднее значение " << endl
 << "равно сумме данных, деленной на количество " << endl
 << "элементов (" << arraySize << "). Среднее значение для"
<< endl
 << "данного расчета равно: "
 << total << " / " << arraySize << " = "
 << setiosflags(ios::fixed | ios::showpoint)
 << setprecision(4) << (float) total / arraySize
 << endl << endl;
}

void median(const int answer[], int size)
{
 cout << endl << "*****" << endl << " Медиана" << endl
 << "*****" << endl
 << "Несортированный массив ответов";

 printArray(answer, size);
 bubbleSort(answer, size);
 cout << endl << endl << "Сортированный массив ответов";
 printArray(answer, size);
 cout << endl << endl << "Медиана - это элемент " << size / 2
 << " из массива " << endl << size << " сортированных элементов. "
<< endl
 << "Для данного расчета медиана равна "
 << answer[size / 2] << endl << endl;
}

void mode(const int freq[], int answer[], int size)
{
 int largest = 0, modeValue = 0;

 cout << endl << "*****" << endl << " Мода" << endl
 << "*****" << endl;

 for (int rating = 1; rating <= 9; rating++)
 freq[rating] = 0;

 for (int j = 0; j < size; j++)
 ++freq[answer[j]];

 cout << "Ответ" << setw(11) << "Частота"
 << setw(19) << "Гистограмма" << endl << endl << setw(54)
 << "1 1 2 2" << endl << setw(54)
 << "5 0 5 0 5" << endl << endl;

 for (rating = 1; rating <= 9; rating++) {
 cout << setw(8) << rating << setw(11)
 << freq [rating] << " ";
 }

 if (freq[rating] > largest) {
 largest = freq[rating];
 modeValue = rating;
 }
}

```

**Рис. 4.17.** Программа анализа данных обследования (часть 2 из 3)

```
for (int h = 1; h <= freq[rating]; h++)
 cout << '*';

cout << endl;
}

cout << "Мода - наиболее часто встречающееся значение." << endl
 << "Для данного расчета мода равна " << modeValue
 << ", это число встречается " << largest << " раз."
 << endl;

void bubbleSort(int a[], int size)
{
 int hold;

 for (int pass = 1; pass < size; pass++)

 for (int j = 0; j < size - 1; j++)

 if (a[j] > a[j+1]) {
 hold = a[j];
 a[j] = a[j+1];
 a[j+1] = hold;
 }
 }

void printArray(const int a[], int size)
{
 for (int j = 0; j < size; j++) {

 if (j % 20 == 0)
 cout << endl;

 cout << setw(2) << a[j];
 }
}
```

Рис. 4.17. Программа анализа данных обследования (часть 3 из 3)

## 4.8. Поиск в массивах: линейный поиск и двоичный поиск

Часто программисту приходится работать с большими объемами данных, хранящимися в виде массивов. Может оказаться необходимым определить, содержит ли массив значение, которое соответствует определенному *ключевому значению*. Процесс нахождения какого-то элемента массива называют *поиском*. В этом разделе мы обсудим два способа поиска — простой *линейный поиск* и более эффективный *двоичный поиск* — *дихотомию*. Упражнения 4.33 и 4.34 содержат задачи реализации рекурсивных вариантов линейного поиска и двоичного поиска.

\*\*\*\*\*

**Среднее**

\*\*\*\*\*

Среднее является средним значением элементов данных. Среднее значение равно сумме данных, деленной на количество элементов (99). Среднее значение для данного расчета равно:  $681 / 99 = 6.8788$

\*\*\*\*\*

**Медиана**

\*\*\*\*\*

**Несортированный массив ответов**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |   |

**Сортированный массив ответов**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 | 7 | 7 | 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 |

Медиана – это элемент 49 из массива 99 сортированных элементов.

Для данного расчета медиана равна 7

\*\*\*\*\*

**Мода**

\*\*\*\*\*

**Ответ      Частота**

**Гистограмма**

|   | 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 | ***** |

Мода – наиболее часто встречающееся значение.

Для данного расчета мода равна 8, это число встречается 27 раз.

**Рис. 4.18.** Пример выполнения программы анализа данных обследования

Линейный поиск (рис. 4.19) сравнивает каждый элемент массива с *ключом поиска*. Поскольку массив не упорядочен, вполне вероятно, что отыскиваемое значение окажется первым же элементом массива. Но в среднем, однако, программа должна сравнить с ключем поиска половину элементов массива.

```
// Линейный поиск в массиве

#include <iostream.h>

int linearSearch(int [], int, int);

main()
{
 const int arraySize = 100;
 int a[arraySize], searchKey, element;

 for (int x = 0; x < arraySize; x++) //создание
 // некоторых данных
 a[x] = 2 * x;

 cout << "Введите ключ поиска - целое число:" << endl;
 cin >> searchKey;
 element = linearSearch(a, searchKey, arraySize);

 if (element != -1)
 cout << "Найдено значение в элементе " << element << endl;
 else
 cout << "Значение не найдено" << endl;

 return 0;
}

int linearSearch(int array[], int key, int sizeOfArray)
{
 for (int n = 0; n < sizeOfArray; n++)
 if (array[n] == key)
 return n;

 return -1;
}
```

Введите ключ поиска - целое число:

36

Найдено значение в элементе 18

Введите ключ поиска - целое число:

37

Значение не найдено

Рис. 4.19. Линейный поиск в массиве

Метод линейного поиска хорошо работает для небольших или для несортированных массивов. Однако, для больших массивов линейный поиск неэффективен. Если массив отсортирован, можно использовать высокоэффективный метод двоичного поиска.

Алгоритм двоичного поиска исключает половину еще непроверенных элементов массива после каждого сравнения. Алгоритм определяет местоположение среднего элемента массива и сравнивает его с ключом поиска. Если они равны, то ключ поиска найден и выдается индекс этого элемента. В противном случае задача сокращается на половину элементов массива. Если ключ поиска меньше, чем средний элемент массива, то дальнейший поиск осуществляется в первой половине массива, а если больше, то во второй половине. Если ключ поиска не совпадает со средним элементом выбранного подмассива (части исходного массива), то алгоритм повторно применяется и сокращает область поиска до четверти исходного массива. Поиск продолжается до тех пор, пока ключ поиска не станет равным среднему элементу или пока оставшийся подмассив содержит хотя бы один элемент, не равный ключу поиска (т.е. пока не найден ключ поиска).

В наихудшем случае двоичный поиск в массиве из 1024 элементов потребует только 10 сравнений. Повторяющееся деление 1024 на 2 (поскольку после каждого сравнения мы можем исключить половину элементов массива) дает 512, 256, 128, 64, 32, 16, 8, 4, 2 и 1. Число 1024 ( $2^{10}$ ) делится на 2 только десять раз. Деление на 2 эквивалентно одному сравнению в алгоритме двоичного поиска. Массив из 1048576 ( $2^{20}$ ) элементов требует для нахождения ключа поиска самое большое 20 сравнений. Массив из одного миллиарда элементов требует для нахождения ключа поиска максимум 30 сравнений. Это огромное увеличение эффективности по сравнению с линейным поиском, который в среднем требует числа сравнений, равного половине числа элементов в массиве. Для миллиарда элементов выигрыш равен разнице между 500 миллионами сравнений и 30 сравнениями! Максимальное количество сравнений, необходимое для двоичного поиска в любом отсортированном массиве, может быть определено как первый показатель степени, при возведении в который числа 2 будет превышено число элементов в массиве.

Рисунок 4.20 представляет итеративную версию функции `binarySearch`. Функция получает четыре аргумента — массив целых чисел `b`, целое число `searchKey`, индекс массива `low` и индекс массива `high`. Если ключ поиска не соответствует среднему элементу массива, то устанавливается такое значение индекса `low` или `high`, что дальнейший поиск проводится в меньшем подмассиве. Если ключ поиска меньше среднего элемента, индекс `high` устанавливается как `middle - 1`, и поиск продолжается среди элементов от `low` до `middle - 1`. Если ключ поиска больше среднего элемента, индекс `low` устанавливается как `middle + 1`, и поиск продолжается среди элементов от `middle + 1` до `high`. Программа использует массив из 15 элементов. Степень двойки для первого числа, большего, чем количество элементов в данном массиве, равна 4 ( $16 = 2^4$ ), так что для нахождения ключа поиска нужно максимум четыре сравнения. Функция `printHeader` выводит на экран индексы массива, а функция `printRow` выводит каждый подмассив в процессе двоичного поиска. Средний элемент в каждом подмассиве отмечается символом звездочки (\*), чтобы указать тот элемент, с которым сравнивается ключ поиска.

```
// Двоичный поиск в массиве
#include <iostream.h>
#include <iomanip.h>

int binarySearch(int [], int, int, int, int);
void printHeader(int);
void printRow(int [], int, int, int, int);

main()
{
 const int arraySize = 15;
 int a[arraySize], key, result;

 for (int i = 0; i < arraySize; i++) // размещение
 // данных в массиве
 a[i] = 2 * i;

 cout << "Введите ключ - число, находящееся между 0 и 28: ";
 cin >> key;

 printHeader(arraySize);
 result = binarySearch(a, key, 0, arraySize - 1, arraySize);
 if (result != -1)
 cout << endl << key << " найден в элементе массива "
 << result << endl;
 else
 cout << endl << key << " не найден" << endl;

 return 0;
}

// Двоичный поиск
int binarySearch (int b[], int searchKey, int low, int high,
 int size)
{
 int middle;

 while (low <= high) {
 middle = (low + high) / 2;

 printRow(b, low, middle, high, size);

 if (searchKey == b[middle]) // соответствие
 return middle;
 else if (searchKey < b[middle])
 high = middle - 1; // определение нижнего
 // конца массива
 else
 low = middle + 1; // определение верхнего
 // конца массива
 }
 return -1; // searchKey не найден
}
```

Рис. 4.20. Двоичный поиск в сортированном массиве (часть 1 из 3)

```

// Печать заголовка выходных данных
void printHeader(int size)
{
 cout << endl << "Индексы:" << endl;

 for (int i = 0; i < size; i++)
 cout << setw(3) << i << ' ';

 cout << endl;

 for (i = 0; i < size; i++)
 cout << "----";

 cout << endl;
}

// Печать одной строки, показывающей текущую
// обрабатываемую часть массива
void printRow(int b[], int low, int mid, int high, int size)
{
 for (int i = 0; i < size; i++)
 if (i < low || i > high)
 cout << " ";
 else if (i == mid)
 cout << setw(3) << b[i] << '*' ; // отметить
 // среднее значение
 else
 cout << setw(3) << b[i] << ' ';

 cout << endl;
}

```

**Рис. 4.20.** Двоичный поиск в сортированном массиве (часть 2 из 3)

## 4.9. Многомерные массивы

Массивы в C++ могут иметь много индексов. Обычным представлением многомерных массивов являются *таблицы* значений, содержащие информацию в *строках* и *столбцах*. Чтобы определить отдельный табличный элемент, нужно указать два индекса: первый (по соглашению) указывает номер строки, а второй (по соглашению) указывает номер столбца. Таблицы или массивы, которые требуют двух индексов для указания отдельного элемента, называются *двумерными массивами*. Заметим, что многомерный массив может иметь более двух индексов. Компиляторы C++ поддерживают по меньшей мере 12-мерные массивы.

Рис. 4.21 иллюстрирует двумерный массив *a*. Массив содержит три строки и четыре столбца, так что, как говорят, — это массив три на четыре. Вообще, массивы с *m* строками и *n* столбцами называют массивами *m* на *n*.

Каждый элемент в массиве *a* определяется именем элемента в форме *a[i][j]*; *a* — это имя массива, *a* *i* и *j* — индексы, которые однозначно определяют каждый элемент в *a*. Заметим, что имена элементов первой строки имеют первый индекс 0; имена элементов в четвертом столбце имеют второй индекс 3.

Введите ключ - число, находящееся между 0 и 28: 25

Индексы:

| 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 не найден

Введите ключ - число, находящееся между 0 и 28: 8

Индексы:

| 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 найден в элементе массива 4

Введите ключ - число, находящееся между 0 и 28: 6

Индексы:

| 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 найден в элементе массива 3

Рис. 4.20. Двоичный поиск в сортированном массиве (часть 3 из 3)

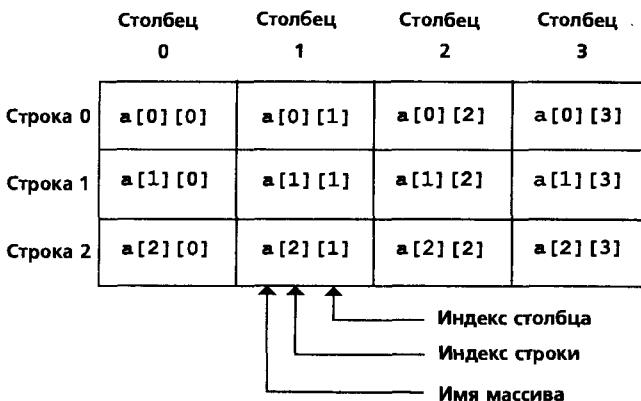


Рис. 4.21. Двумерный массив с тремя строками и четырьмя столбцами

#### Типичная ошибка программирования 4.9

Неправильная ссылка на элемент двумерного массива `a[x] [y]` как `a[x,y]`. На самом деле, `a[x,y]` воспринимается как `a[y]`, потому что C++ оценивает выражение (содержащее операцию последовательности – запятую) `x , y` просто как `y` (последнее из разделенных запятыми выражений).

Многомерные массивы могут получать начальные значения в своих объявлениях точно так же, как массивы с единственным индексом. Например, двумерный массив `b[2][2]` можно объявить и дать ему начальные значения таким образом:

```
int b[2][2] = {{1,2}, {3, 4}};
```

Значения группируются в строки, заключенные в фигурные скобки. Таким образом, элементы `b[0][0]` и `b[0][1]` получают начальные значения 1 и 2, а элементы `b[1][0]` и `b[1][1]` получают начальные значения 3 и 4. Если начальных значений в данной строке не хватает для их присвоения всем элементам строки, то остающимся элементам строки присваиваются нулевые начальные значения. Таким образом, объявление

```
int b[2][2] = {{1,}, {3, 4}};
```

будет означать что `b[0][0]` получает начальное значение 1, `b[0][1]` получает начальное значение 0, `b[1][0]` получает начальное значение 3 и `b[1][1]` получает начальное значение 4.

Рисунок 4.22 демонстрирует присваивание начальных значений двумерным массивам в объявлениях. Программа объявляет три массива, каждый с тремя строками и тремя столбцами. Объявление `array1` имеет шесть начальных значений в двух подсписках. Первый под список присваивает начальные значения 1, 2 и 3 элементам первой строки массива; второй под список присваивает начальные значения 4, 5 и 6 элементам второй строки массива. Если фигурные скобки вокруг каждого под списка удалить из списка начальных значений `array1`, то компилятор автоматически присвоит первые начальные значения элементам первой строки, а следующие — элементам второй строки.

Объявление `array2` содержит пять начальных значений. Начальные значения присваиваются первой строке, затем второй строке. Любые элементы, которые не имеют явно заданных начальных значений, автоматически получают нулевые начальные значения, так что элемент `array[1][2]` получит нулевое начальное значение.

Объявление `array3` имеет три начальные значения в двух подсписках. Под список для первой строки явно присваивает начальные значения 1 и 2 первым двум элементам первой строки. Третий элемент автоматически получает нулевое начальное значение. Под список для второй строки явно присваивает начальное значение 4 первому элементу. Остальные два элемента автоматически получают нулевые начальные значения.

Программа вызывает функцию `printArray` для вывода элементов каждого массива. Заметим, что описание функции указывает параметр — массив как `int a[ ][3]`. Когда мы задаем как аргумент функции одномерный массив, скобки в списке параметров функции пусты. Размерность первого индекса многомерного массива также не требуется, но все последующие размерности индексов необходимы. Компилятор использует размерности этих индексов для определения ячеек в памяти элементов многомерных массивов. В памяти все элементы массива хранятся последовательно, независимо от количества индексов. В двумерном массиве первая строка хранится в памяти перед второй строкой.

```
// Присваивание начальных значений в многомерных массивах
#include <iostream.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} };

 cout << "Значения массива array1 по строкам:" << endl;
 printArray(array1);

 cout << "Значения массива array2 по строкам:" << endl;
 printArray(array2);

 cout << "Значения массива array3 по строкам:" << endl;
 printArray(array3);

 return 0;
}

void printArray(int a[][3])
{
 for (int i = 0; i <= 1; i++) {

 for (int j = 0; j <= 2; j++)
 cout << a[i][j] << ' ';

 cout << endl;
 }
}
```

```
Значения массива array1 по строкам:
1 2 3
4 5 6
Значения массива array2 по строкам:
1 2 3
4 5 0
Значения массива array3 по строкам:
1 2 0
4 0 0
```

Рис. 4.22. Присваивание начальных значений в многомерных массивах

Наличие размерностей индексов в объявлении параметра дает возможность компилятору сообщить функции о том, как расположены элементы в массиве. В двумерном массиве каждая строка по существу является одномерным массивом. Чтобы определить местоположение элемента в некоторой строке, функция должна точно знать, сколько элементов находится в каждой строке, чтобы она смогла пропустить соответствующее количество ячеек памяти при обращении к массиву. Таким образом, при обращении к `a[1][2]` функция знает, что для доступа ко второй строке (строка 1) нужно пропустить в памяти три элемента первой строки, а затем обратиться к третьему элементу этой строки (элементу 2).

Многие типовые операции с массивами используют структуру повторения **for**. Например, следующая структура **for** обнуляет все элементы третьей строки массива **a** на рис. 4.21:

```
for (column = 0; column < 4; column++)
 a[2][column] = 0;
```

Мы указали третью строку, потому что знаем, что ее первый индекс всегда равен 2 (0 — это первая строка, а 1 — это вторая строка). Цикл **for** варьирует только второй индекс (т.е. индекс столбца). Предыдущая структура **for** эквивалентна операторам присваивания:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

Следующий вложенный цикл **for** определяет сумму всех элементов массива **a**:

```
total = 0;
for (row = 0; row < 3; row++)
 for (column = 0; column < 3; column++)
 total += a[row][column];
```

Внутренняя структура **for** суммирует элементы одной строки массива. Внешняя структура **for** начинает работу с установки **row** (т.е. индекса строки) в нуль, так что во внутренней структуре **for** могут быть просуммированы элементы первой строки. Затем внешняя структура **for** увеличивает **row** на 1, так что могут быть просуммированы элементы второй строки. Далее внешняя структура **for** увеличивает **row** до значения 2, так что могут быть просуммированы элементы третьей строки. После завершения работы вложенной структуры **for** печатается результат.

Программа на рис. 4.23 выполняет несколько других типовых операций над массивом **studentGrades** размером 3 на 4 . Каждая строка массива представляет студента, а каждый столбец представляет оценку, полученную студентом на одном из четырех экзаменов в течение семестра. Манипуляции с массивом выполняются с помощью четырех функций. Функция **minimum** определяет наихудшую оценку всех студентов в течение семестра. Функция **maximum** определяет наилучшую оценку всех студентов в течение семестра. Функция **average** определяет среднесеместровую оценку отдельного студента. Функция **printArray** выводит двумерный массив в табличном виде.

Каждая из функций **minimum**, **maximum** и **printArray** получает три аргумента — массив **studentGrades** (называемый **grades** в каждой функции), количество студентов (число строк массива) и количество экзаменов (число столбцов массива). Каждая функция циклически обрабатывает массив **grades**, используя вложенные структуры **for**. Приведем пример вложенной структуры **for** из описания функции **minimum**:

```
for (i = 0; i < pupils; i++)
 for (j = 0; j < tests; j++)
 if (grades[i][j] < lowGrade)
 lowGrade = grades[i][j];
```

```
// Пример использования двумерного массива
#include <iostream.h>
#include <iomanip.h>

const int students = 3; // количество студентов
const int exams = 4; // количество экзаменов

int minimum(int [][]exams, int, int); int
maximum(int [][]exams, int, int);
float average(int [], int);
void printArray(int [][]exams, int, int);

main()
{
 int studentGrades [students][exams] = {{77, 68, 86, 73},
 {96, 87, 89, 78},
 {70, 90, 86, 81}};

 cout << "Массив:" << endl;
 printArray(studentGrades, students, exams);
 cout << endl << endl << "Наиходшая оценка: "
 << minimum(studentGrades, students, exams) << endl
 << "Наилучшая оценка: "
 << maximum(studentGrades, students, exams) << endl;
 for (int person = 0; person < students; person++)
 cout << "Средняя оценка студента " << person << " равна "
 << setiosflags(ios::fixed | ios::showpoint)
 << setprecision(2)
 << average(studentGrades[person], exams) << endl;

 return 0;
}
// Поиск минимальной оценки
int minimum(int grades[][exams], int pupils, int tests)
{
 int lowGrade = 100;

 for (int i = 0; i < pupils; i++)
 for (int j = 0; j < tests; j++)
 if (grades[i][j] < lowGrade)
 lowGrade = grades[i][j];

 return lowGrade;
}
// Поиск максимальной оценки
int maximum(int grades [] [exams], int pupils, int tests)
{
 int highGrade = 0;

 for (int i = 0; i < pupils; i++)
 for (int j = 0; j < tests; j++)
 if (grades [i][j] > highGrade)
 highGrade = grades [i][j];

 return highGrade;
}
```

Рис. 4.23. Пример использования двумерного массива (часть 1 из 2)

```

// Определение средней оценки для отдельного студента
float average(int setOfGrades [], int tests)
{
 int total = 0;

 for (int i = 0; i < tests; i++)
 total += setOfGrades[i];

 return (float) total / tests;
}

// Печать массива
void printArray(int grades [] [exams], int pupils, int tests)
{
 cout<< " [0] [1] [2] [3]";

 for (int i = 0; i < pupils; i++) {
 cout << endl << "Оценки студента [" << i << "] ";

 for (int j = 0; j < tests; j++)
 cout << setiosflags(ios::left) << setw(5)
 << grades [i] [j];
 }
}

```

**Массив:**

|                            |     |     |     |     |
|----------------------------|-----|-----|-----|-----|
|                            | [0] | [1] | [2] | [3] |
| <b>Оценки студента [0]</b> | 77  | 68  | 86  | 73  |
| <b>Оценки студента [1]</b> | 96  | 87  | 89  | 78  |
| <b>Оценки студента [2]</b> | 70  | 90  | 86  | 81  |

**Наихудшая оценка:** 68

**Наилучшая оценка:** 96

**Средняя оценка студента 0 равна** 76.00

**Средняя оценка студента 1 равна** 87.50

**Средняя оценка студента 2 равна** 81.75

**Рис. 4.23.** Пример использования двумерного массива (часть 2 из 2)

Внешняя структура **for** начинается с установки **i** (т.е. индекса строки) равным нулю, чтобы элементы первой строки можно было сравнивать с переменной **lowGrade** в теле внутренней структуры **for**. Внутренняя структура **for** циклически обрабатывает четыре оценки каждой строки и сравнивает каждую оценку с **lowGrade**. Если оценка меньше, чем **lowGrade**, **lowGrade** устанавливается равной этой оценке. Затем внешняя структура **for** увеличивает индекс строки до значения 1. Элементы второй строки сравниваются с переменной **lowGrade**. Затем внешняя структура **for** увеличивает индекс строки до значения 2. Элементы третьей строки сравниваются с переменной **lowGrade**. Когда выполнение вложенной структуры заканчивается, **lowGrade** содержит наименьшую оценку в двумерном массиве. Функция **maximum** работает аналогично функции **minimum**.

Функция **average** принимает два аргумента — одномерный массив результатов экзаменов для одного студента и количество результатов экзаменов в массиве. При вызове **average** первый аргумент — это **studentGrades[stu-**

`dent`], который указывает, что в `average` передается отдельная строка двумерного массива `studentGrades`. Например, аргумент `studentGrades[1]` представляет собой четыре значения (одномерный массив оценок), хранимый во второй строке двумерного массива `studentGrades`. Двумерный массив можно рассматривать как массив с элементами, представляющими собой одномерные массивы. Функция `average` подсчитывает сумму элементов массива, делит ее на количество результатов экзаменов и возвращает результат в форме с плавающей запятой.

## 4.10. Размышления об объектах: идентификация поведений объектов

В разделах «Размышления об объектах» в конце глав 2 и 3 мы выполнили два первых этапа объектно-ориентированного проектирования для нашей модели лифта, а именно, идентификацию объектов, необходимых для разработки модели, и идентификацию атрибутов этих объектов.

В данном лабораторном задании мы сосредоточимся на определении поведения объектов, необходимых для разработки модели лифта. В главе 5 мы уделим внимание взаимодействию между объектами.

Рассмотрим поведение некоторых реально существующих объектов. Поведение радио включает настройку на станцию и установку громкости. Поведение автомобиля включает ускорение (нажатием педали акселератора) и замедление (нажатием педали тормоза).

Как мы увидим, объекты обычно не определяют свое поведение случайным образом. Тот или иной вариант поведения определяется посылкой объекту сообщения, требующего, чтобы объект выработал именно такой способ поведения. Это созвучно вызову функции — тому, как посылаются сообщения объектам в C++.

### Лабораторное задание 3 по лифту

1. Продолжите работу с файлом фактов, созданным вами в главе 3. Вы разделили факты, относящиеся к каждому объекту, на две группы. Вы пометили первую группу как *Атрибуты*, а вторую — как *Другие Факты*.
2. Для каждого объекта добавьте третью группу, названную *Варианты поведения*. Поместите в эту группу каждый вариант поведения этого объекта, который может быть вызван какой-либо переданной ему информацией, т.е. посылкой объекту некоторого сообщения. Например, кнопка может быть нажата (пассажиром), так что включите в список вариантов поведения кнопки как объекта элемент «нажатие кнопки» — `pushButton`. Функцию `pushButton` и другие варианты поведения объекта кнопка называют *функциями-элементами* объекта кнопка. Атрибуты объекта (такие как положение кнопки «включено» и «выключено») называются *данными-элементами* объекта кнопка. Функции-элементы объекта обычно манипулируют данными-элементами объекта (подобно тому как `pushButton` изменяет один из атрибутов кнопки на «включено»). Функции-элементы обычно посыпают сообщения другим объектам (подобно тому, как объект кнопка посыпает сообщение «иди ко мне» — `comeGetMe`, чтобы вызвать лифт). Допустим, что лифт будет

иметь световые кнопки, которые зажигаются при их нажатии. Когда объект лифт прибывает на этаж, он захочет послать сообщение «*выключить кнопку*» — *resetButton*, чтобы выключить освещение кнопки. Объект лифт может захотеть определить, была ли нажата определенная кнопка, так что мы можем предусмотреть еще одно поведение — «*получение состояния кнопки*» — *getButton*, которое анализирует кнопку и возвращает 1 или 0, указывающие текущее состояние клавиши: «включено» или «выключено». Возможно, вы захотите, чтобы двери лифта посылали сообщения «*двери открыты*» — *openDoors*, «*двери закрыты*» — *closeDoors* и так далее.

3. Для каждого варианта поведения, который вы приписываете объекту, дайте краткое описание того, в чем заключается поведение в данном варианте. Составьте список всех изменений атрибутов, вызывающих соответствующее поведение, и список всех сообщений, которые поведение посылает другим объектам.

### Замечания

1. Начните с составления списка вариантов поведения объекта, явно упоминаемых к постановке задачи. Затем перечислите варианты поведения, которые подразумеваются постановкой задачи.
2. Добавляйте соответствующие варианты поведения, как только становится очевидной их необходимость.
3. Не забывайте что проектирование системы — процесс не имеющий четкого момента окончания. Сделайте наилучшим образом то, что пока можете, и будьте готовы к модификации проекта, так как это упражнение будет продолжено в последующих главах.
4. Как видите, на этом этапе проектирования весьма трудно собирать по крохам возможные варианты поведения. Возможно, вы добавите новые варианты поведения вашего объекта при продолжении задания в главе 5.

### Резюме

- C++ хранит списки значений в массивах. Массив — это последовательная группа связанных ячеек памяти. Эти ячейки связаны тем, что все они имеют одно и то же имя и один и тот же тип. Чтобы сослаться на отдельную ячейку или элемент массива, нужно указать имя массива и индекс.
- Индекс может быть целым числом или целым выражением. Если программа использует в качестве индекса выражение, то выражение вычисляется, чтобы определить конкретный элемент массива.
- Важно различать ссылку на седьмой элемент массива и элемент массива семь. Седьмой элемент массива имеет индекс 6, тогда как элемент массива семь имеет индекс 7 (в действительности, это восьмой элемент массива). Это источник ошибок типа завышения (или занижения) на единицу.

- Массивы занимают место в памяти. Чтобы зарезервировать память для 100 элементов массива целых значений **b** и 27 элементов массива целых значений **x**, программист пишет

```
int b[100], x[27];
```

- Массив типа **char** можно использовать для хранения строки символов.
- Элементам массива можно задавать начальные условия тремя способами: в объявлении, присваиванием или при вводе.
- Если в списке инициализации начальных значений меньше, чем элементов массива, оставшиеся элементы массива принимают нулевые начальные значения.
- С++ не предотвращает ссылок на элементы, находящиеся за пределами массива.
- Массиву символов можно задавать начальные условия, используя лiteralную константу.
- Все строки заканчиваются нулевым символом ('\0').
- Символьным массивам можно задавать начальные значения с помощью символьных констант в списке инициализации.
- К отдельным символам строки, хранящимся в массиве, можно обращаться прямо, используя запись индексов массива.
- Для передачи массива функции нужно передать ей имя массива. Чтобы передать функции единственный элемент массива, просто передайте имя массива и после него индекс (заключенный в квадратные скобки) данного элемента.
- Массивы передаются функциям с помощью моделируемого вызова по ссылке — вызываемые функции могут модифицировать значения элементов в исходных массивах оператора вызова. Значение имени массива — это адрес первого элемента массива. Поскольку в функцию передается начальный адрес массива, вызываемая функция знает, где хранится массив.
- Чтобы принять аргумент-массив, список параметров функции должен указывать, что передается массив. Размер массива в скобках после имени массива указывать не обязательно.
- В С++ имеется спецификация типа **const**, которая запрещает модификацию значений массива в функции. Когда параметру-массиву предшествует спецификатор **const**, элементы массива становятся константами в теле функции и любая попытка модифицировать элементы массива внутри тела функции приводит к ошибке трансляции.
- Массив можно сортировать, используя технику пузырьковой сортировки. Выполняется несколько проходов массива. На каждом проходе сравниваются пары последовательных элементов. Если элементы пары расположены в нужном порядке (или элементы равны), они остаются на своих местах. Если требуемый порядок нарушен, значения переставляются местами. Для небольших массивов пузырьковая сортировка приемлема, но для больших массивов она неэффективна по сравнению с другими, более сложными алгоритмами сортировки.

- Линейный поиск сравнивает каждый элемент массива с ключом поиска. Поскольку в массиве нет определенного порядка, одинаково вероятно, что значение будет найдено как в первом элементе, так и в последнем. Поэтому в среднем программа должна сравнить с ключом поиска половину элементов массива. Метод линейного поиска хорошо работает для небольших массивов и применим для несортированных массивов.
- Двоичный поиск исключает половину элементов массива после каждого сравнения. Алгоритм выделяет средний элемент массива и сравнивает его с ключом поиска. Если они равны, ключ поиска найден и возвращается индекс массива этого элемента. В противном случае задача поиска сокращается на половину массива.
- В наихудшем случае двоичный поиск в массиве из 1024 элементов потребует всего 10 сравнений.
- Массивы можно использовать для представления таблиц значений, содержащих информацию, расположенную в виде строк и столбцов. Для идентификации отдельного элемента таблицы указываются два индекса: первый (по соглашению) определяет строку, в которой находится элемент, а второй (по соглашению) определяет столбец, содержащий этот элемент. Таблицы или массивы, требующие для идентификации отдельного элемента указания двух индексов, называются двумерными массивами.
- Когда мы передаем одномерный массив в качестве аргумента функции, в списке параметров функции скобки массива могут быть пусты. Размерность первого индекса многомерного массива тоже не нужно указывать, но размерности всех последующих индексов указывать необходимо. Компилятор использует эти размерности для того, чтобы определить расположение в памяти элементов многомерного массива.
- Чтобы передать функции одну строку двумерного массива, которая принимается как одномерный массив, просто передается имя массива и вслед за ним первый индекс.

## Терминология

|                              |                                         |
|------------------------------|-----------------------------------------|
| <code>a [ i ]</code>         | массив                                  |
| <code>a [ i ] [ j ]</code>   | массив <code>m</code> на <code>n</code> |
| временная область для обмена | масштабируемость                        |
| значений                     | многомерный массив                      |
| выход за пределы массива     | моделируемый вызов по ссылке            |
| двоичный поиск в массиве     | начальное значение                      |
| двумерный массив             | номер позиции                           |
| значение элемента            | нулевой символ '/0'                     |
| именованная константа        | нулевой элемент                         |
| имя массива                  | объявление массива                      |
| индекс                       | одномерный массив                       |
| индекс столбца               | определение пределов                    |
| индекс строки                | ошибка типа завышения (или              |
| квадратные скобки [ ]        | занизжения) на единицу                  |
| ключ поиска                  | передача массива функции                |
| линейный поиск в массиве     | передача по ссылке                      |

|                                         |                                |
|-----------------------------------------|--------------------------------|
| присваивание начальных значений массиву | спецификация типа <b>const</b> |
| проход пузырьковой сортировки           | список начальных значений      |
| пузырьковая сортировка                  | строка                         |
| скаляр                                  | таблица значений               |
| сортировка                              | табулированный формат          |
| сортировка массива                      | трехмерный массив              |
| сортировка погружением                  | элемент массива                |

## Типичные ошибки программирования

- 4.1. Важно отметить различие между «седьмым элементом массива» и «элементом массива семь». Поскольку индексы массива начинаются с 0, «седьмой элемент массива» имеет индекс шесть, тогда как «элемент массива семь» имеет индекс 7 и на самом деле является восьмым элементом массива. Это — источник ошибок типа завышения (или занижения) на единицу.
- 4.2. Забывают присвоить начальные значения элементам массива, требующим такого присваивания.
- 4.3. Задание в списке начальных значений большего числа значений, чем имеется элементов в массиве, является синтаксической ошибкой.
- 4.4. Присваивание значения именованной константе в выполняемом операторе является синтаксической ошибкой
- 4.5. Завершение директивы препроцессора `#include` точкой с запятой. Помните, что директивы препроцессора не являются операторами C++.
- 4.6. Ссылка на элемент, находящийся вне границ массива.
- 4.7. Недостаточный размер массива, в который с помощью `cin` вводится символьная строка с клавиатуры, может привести к потере данных в программе и к другим ошибкам выполнения.
- 4.8. Предположение, что элементы локального массива класса **static** в функции получают нулевые начальные значения при каждом вызове функции.
- 4.9. Неправильная ссылка на элемент двумерного массива `a[x][y]` как `a[x,y]`. На самом деле, `a[x,y]` воспринимается как `a[y]`, потому что C++ оценивает выражение (содержащее операцию последования — запятую) `x`, у просто как `y` (последнее из разделенных запятыми выражений).

## Хороший стиль программирования

- 4.1. Страйтесь программировать понятно. Иногда имеет смысл пожертвовать более высокой эффективностью использования памяти или процессорного времени в пользу написания более понятной программы.

- 4.2. При использовании циклов с массивом индекс массива никогда не должен быть меньше нуля и должен всегда быть меньше, чем общее количество элементов массива (меньше, чем размер массива). Удостоверьтесь, что условия завершения цикла предотвращают обращение к элементам, выходящим за пределы этого диапазона.
- 4.3. В программах должна быть обеспечена правильность всех вводимых значений, чтобы не допустить возможность влияния ошибочной информации на ход вычислений в программе.
- 4.4. Некоторые программисты, чтобы сделать программу понятнее, включают имена переменных в функции прототипов. Компиляторы игнорируют эти имена.

### Советы по повышению эффективности

- 4.1. Иногда соображения эффективности далеко перевешивают соображения понятности.
- 4.2. Передача массивов с помощью моделируемого вызова по ссылке ощутимо влияет на производительность. Если бы массивы передавались вызовом по значению, передавалась бы копия каждого элемента. Для больших, часто передаваемых массивов это привело бы к значительному потреблению времени и памяти для хранения копий массивов.
- 4.3. Часто простейшие алгоритмы обеспечивают низкую производительность. Их достоинство лишь в том, что их легко писать, проверять и отлаживать. Однако, часто для получения максимальной производительности необходимы более сложные алгоритмы.

### Замечания по мобильности

- 4.1. Эффекты (обычно серьезные) ссылок на элементы, выходящие за границы массива, системно зависимы.

### Замечания по технике программирования

- 4.1. Определение размера каждого массива с помощью именованной константы делает программу более масштабируемой.
- 4.2. Передавать массив по значению можно, используя простой прием, который мы объясним в главе 6.
- 4.3. Для предотвращения модификации исходного массива внутри тела функции можно в определении функции применять к параметру массив спецификатор типа `const`. Это еще один пример принципа наименьших привилегий. Функциям не должно быть позволено модифицировать массивы без крайней необходимости.

### Упражнения для самопроверки

- 4.1. Заполнить пробелы в следующих утверждениях:
  - а) Списки и таблицы значений хранятся в \_\_\_\_\_.

- b) Элементы массива связаны тем, что они имеют одно и то же \_\_\_\_\_ и \_\_\_\_\_.
- c) Число, используемое для обращения к отдельному элементу массива, называется \_\_\_\_\_.
- d) Для объявления размера массива должна использоваться \_\_\_\_\_, потому что она делает программу более масштабируемой.
- e) Процесс упорядоченного размещения элементов в массиве называется \_\_\_\_\_.
- f) Процесс определения значения ключа, содержащегося в массиве, называется \_\_\_\_\_.
- g) Массив, использующий два индекса, называется \_\_\_\_\_.
- 4.2. Укажите, верны ли следующие утверждения. Если нет, объясните почему.
- Массив может хранить много различных типов данных.
  - Индексы массива обычно должны иметь тип **float**.
  - Если количество начальных значений в списке инициализации меньше чем количество элементов массива, оставшиеся элементы автоматически получают в качестве начальных значений последние значения из списка инициализации.
  - Если список инициализации содержит начальных значений больше, чем элементов массива, то это — ошибка.
  - Отдельный элемент массива, который передается функции и модифицируется в этой функции, будет содержать модифицированное значение после завершения выполнения вызываемой функции.
- 4.3. Напишите операторы, реализующие следующие операции с массивом **fractions**.
- Определите именованную константу **arraySize** с начальным значением 10.
  - Объявите массив с числом элементов **arraySize** типа **float**, имеющими нулевые начальные значения.
  - Назовите четвертый элемент от начала массива.
  - Обратитесь к элементу массива 4.
  - Присвойте значение 1.667 элементу массива 9.
  - Присвойте значение 3.333 седьмому элементу массива.
  - Напечатайте элементы массива 6 и 9 с двумя цифрами справа от десятичной точки и покажите, как будут выглядеть выходные данные, отображаемые на экране.
  - Напечатайте все элементы массива, используя структуру повторения **for**. Определите целую переменную **x** в качестве переменной, управляющей циклом. Покажите, как будут выглядеть выходные данные.
- 4.4. Напишите операторы, реализующие следующие операции с массивом **table**.

- a) Объявите массив, который должен быть массивом целых чисел и иметь три строки и три столбца. Полагайте, что определена именованная константа `arraySize`, равная 3.
- b) Сколько элементов содержит массив?
- c) Используйте структуру повторения `for` для задания начальных значений каждому элементу массива, равных сумме его индексов. Полагайте, что объявлены целые переменные `x` и `y`, являющиеся управляющими переменными.
- d) Напишите фрагмент программы для печати каждого элемента массива `table` в табулированном формате с тремя строками и тремя столбцами. Полагайте, что массив получил начальные значения в объявлении

```
int table [arraySize] [arraySize] = {{1, 8}, {2, 4, 6}, {5}};
```

и объявлены целые переменные `x` и `y`, являющиеся управляющими переменными. Покажите, как будут выглядеть выходные данные.

- 4.5.** Найдите и исправьте ошибку в каждом из следующих фрагментов программ.

- a) #include <iostream.h>;
- b) arraySize = 10; // переменная arraySize была  
// объявлена как const
- c) Допустим, что int b[10] = {0};  
for (int i = 0; i <= 10; i++)  
 b[i] = 1;
- d) Допустим, что a[2][2] = {{1, 2}, {3, 4}};  
a[1, 1] = 5;

### Ответы на упражнения для самопроверки

- 4.1.** a) массивах. b) имя, тип. c) индекс. d) именованная константа. e) сортировка. f) поиск. g) двумерный.
- 4.2.** a) Неверно. Массив может хранить значения только одинакового типа.  
b) Неверно. Индексы массива обязательно должны быть целыми числами или целыми выражениями.  
c) Неверно. Оставшиеся элементы автоматически получают нулевые начальные значения.  
d) Верно.  
e) Неверно. Отдельные элементы массива передаются вызовом по значению. Только если функции передается массив целиком, любые его модификации будут отражаться на оригинале.
- 4.3.** a) const int arraySize = 10;  
b) float fractions[arraySize] = {0};  
c) fractions[3]  
d) fractions[4]

```
e) fractions[9] = 1.667;
f) fractions[6] = 3.333;
g) cout << setiosflags(ios::fixed | ios::showpoint)
 << setprecision(2) << fractions[6] << ' '
 << fractions[9] << endl;
```

**Выходные данные:** 3.33 1.67.

```
h) for (int x = 0; x < arraySize; x++)
 cout << "fractions[" << x << "] = " << fractions[x]
 << endl;
```

**Выходные данные:**

```
fractions[0] = 0
fractions[1] = 0
fractions[2] = 0
fractions[3] = 0
fractions[4] = 0
fractions[5] = 0
fractions[6] = 3.333
fractions[7] = 0
fractions[8] = 0
fractions[9] = 1.667
```

4.4. a) int table[arraySize] [arraySize];

b) Девять.

```
c) for (x = 0; x < arraySize; x++)
 for (y = 0; y < arraySize; y++)
 table[x][y] = x + y;
```

```
d) cout << " [0] [1] [2] " << endl;
 for (int x = 0; x < arraySize; x++) {
 cout << '[' << x << "] ";

 for (int y = 0; y < arraySize; y++)
 cout << setw(3) << table[x][y] << " ";
 cout << endl;
 }
```

**Выходные данные:**

|     | [0] | [1] | [2] |
|-----|-----|-----|-----|
| [0] | 1   | 8   | 0   |
| [1] | 2   | 4   | 6   |
| [2] | 5   | 0   | 0   |

4.5. a) Ошибка: точка с запятой в конце директивы препроцессора `#include`.

Исправление: удалить точку с запятой.

b) Ошибка: присваивание значения именованной константе оператором присваивания.

Исправление: присвоить значение именованной константе в объявлении `const int arraySize`.

c) Ошибка: ссылка на элемент массива, находящийся за границами массива (`b[10]`).

Исправление: сделать конечное значение управляющей переменной равным 9.

- d) Ошибка: индексирование массива сделано неверно.  
 Исправление: изменить оператор на `a[1][1] = 5;`

## Упражнения

4.6. Заполните пробелы в следующих предложениях:

- a) C++ хранит списки значений в \_\_\_\_\_.
- b) Элементы массива связаны тем, что они \_\_\_\_\_.
- c) При ссылке на элемент массива номер позиции, содержащейся внутри круглых скобок, называется \_\_\_\_\_.
- d) Четыре первых элемента массива `p` имеют следующие имена \_\_\_, \_\_\_, \_\_\_ и \_\_\_.
- e) Наименование массива, задание его типа и указание числа элементов массива называется \_\_\_\_\_ массива.
- f) Процесс размещения элементов массива в порядке возрастания или убывания их значений называется \_\_\_\_\_.
- g) В двумерном массиве первый индекс (по соглашению) определяет \_\_\_\_\_ элемента, а второй (по соглашению) определяет \_\_\_\_\_ элемента.
- h) Массив `m` на `n` содержит \_\_\_\_\_ строк, \_\_\_\_\_ столбцов и \_\_\_\_\_ элементов.
- i) Элемент в строке 3 и столбце 5 массива имеет имя \_\_\_\_\_.

4.7. Определите, какие из следующих утверждений верны, а какие нет; для ошибочных утверждений объясните ошибку.

- a) Чтобы сослаться на конкретную ячейку или элемент внутри массива, мы указываем имя массива и значение данного элемента.
- b) Объявление массива резервирует для него память.
- c) Чтобы указать, что для массива целых чисел `p` должно быть зарезервировано 100 ячеек памяти, программист пишет объявление `p[100];`
- d) Программа на C++, присваивающая начальные нулевые значения массиву из 15 элементов, должна содержать по меньшей мере один оператор `for`.
- e) Программа на C++, которая суммирует элементы двумерного массива, должна содержать вложенные циклы `for`.

4.8. Напишите операторы C++, соответствующие следующим задачам:

- a) Выведите на экран значение седьмого элемента символьного массива `f`.
- b) Введите значение элемента 4 одномерного массива `b` с плавающей запятой.
- c) Присвойте начальное значение 8 каждому из 5 элементов одномерного массива целых чисел `g`.
- d) Просуммируйте и напечатайте сумму 100 элементов массива `g` с плавающей точкой.

e) Скопируйте массив **a** в первую часть массива **b**. Считайте, что массивы объявлены как `float a[11], b[34];`

f) Определите и напечатайте наименьшее и наибольшее значения, содержащиеся в массиве **w** с 99 элементами с плавающей запятой.

4.9. Рассматривается массив целых чисел **t** размером 3 на 4.

a) Напишите объявление для **t**.

b) Сколько строк в массиве **t**?

c) Сколько столбцов в массиве **t**?

d) Сколько элементов в массиве **t**?

e) Напишите имена всех элементов второй строки массива **t**.

f) Напишите имена всех элементов третьего столбца массива **t**.

g) Напишите один оператор, который устанавливает в нуль элемент первой строки и второго столбца массива **t**.

h) Напишите последовательность операторов, которые присваивают нулевые начальные значения всем элементам массива **t**. Не используйте структуру повторения.

i) Напишите вложенную структуру `for`, которая присваивает нулевые начальные значения всем элементам массива **t**.

j) Напишите оператор ввода элементов массива **t** с клавиатуры.

k) Напишите последовательность операторов, которая определяет и печатает наименьшее значение в массиве **t**.

l) Напишите оператор, который выводит на экран элементы первой строки массива **t**.

m) Напишите оператор, который суммирует элементы четвертого столбца массива **t**.

n) Напишите последовательность операторов, которая печатает массив **t** в табулированном формате. Перечислите индексы столбцов как заголовки вверху и индексы строк слева в каждой строке.

4.10. Используйте одномерный массив для решения следующей задачи. Компания платит своим продавцам на комиссионной основе. Продавцы получают 200 долларов в неделю плюс 9 процентов от валовой продажи за эту неделю. Например, продавец, валовая продажа которого за неделю составила 5000 долларов, получает 200 долларов плюс 9 процентов от 5000 долларов, или всего 650 долларов. Напишите программу (используя массив счетчиков), которая определяет, сколько продавцов получили заработную плату в каждом из следующих диапазонов (примем допущение, что зарплата каждого продавца округляется до целого значения):

1. \$200–\$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

**4.11.** Пузырьковая сортировка, представленная на рис. 4.16, неэффективна для больших массивов. Выполните следующие простые модификации для улучшения эффективности пузырьковой сортировки.

- a) После первого прохода наибольшее число гарантированно окажется в элементе массива с наивысшим номером; после второго прохода «на месте» окажутся два наибольших числа и так далее. Вместо выполнения девяти сравнений на каждом проходе, модифицируйте пузырьковую сортировку так, чтобы на втором проходе было восемь сравнений, на третьем проходе — семь и так далее.
- b) Данные в массиве могут уже находиться в необходимом порядке, либо близком к нему, так зачем же делать девять проходов, если достаточно сделать меньше? Модифицируйте сортировку так, чтобы в конце каждого прохода проверять, были ли сделаны какие-либо перестановки. Если не было ни одной, значит, данные уже находятся в соответствующем порядке, так что программа должна завершаться. Если перестановки были сделаны, нужно сделать по меньшей мере еще один проход.

**4.12.** Напишите по одному оператору для выполнения следующих операций с одномерным массивом:

- a) Присвойте нулевые начальные значения 10 элементам массива целых чисел `counts`.
- b) Прибавьте 1 к каждому из 15 элементов массива целых чисел `bonus`.
- c) Прочитайте 12 значений массива `monthlyTemperatures` типа `float` с клавиатуры.
- d) Напечатайте в виде столбца 5 значений массива целых чисел `bestScores`.

**4.13.** Найдите ошибку (или ошибки) в каждом из следующих операторов:

- a) Допустим: `char str[5];`  
`cin >> str; // Пользователь печатает "hello"`
- b) Допустим: `int a[3];`  
`cout << a[1] << " " << a[2] << " " << a[3] << endl;`
- c) `float f[3] = {1.1, 10.01, 100.001, 1000.0001};`
- d) Допустим: `double d[2][10];`  
`d[1, 9] = 2.345;`

**4.14.** Модифицируйте программу на рис. 4.17 так, чтобы функция `mode` оказалась способной обрабатывать случай равного количества нескольких ответов. Модифицируйте также функцию `median` так, чтобы два элемента в середине усреднялись в массиве с четным количеством элементов.

**4.15.** Используйте одномерный массив для решения следующей задачи. Прочтите 20 чисел, каждое из которых находится в диапазоне от 10 до 100 включительно. После того, как прочли очередное число,

напечатайте его, но только в том случае, если оно не дублирует ранее прочитанные числа. Предусмотрите «наихудший случай», когда все 20 чисел различны. Используйте наименьший возможный массив для решения этой задачи.

- 4.16.** Укажите, в каком порядке будут обнуляться элементы двумерного массива sales размером 3 на 5 следующим фрагментом программы:

```
for (row = 0; row < 3; row++)
 for (column = 0; column < 5; column++)
 sales[row][column] = 0;
```

- 4.17.** Напишите программу, моделирующую бросание двух костей. Программа должна использовать `rand` для бросания первой кости и затем снова `rand` для метания второй кости. Затем должна подсчитываться сумма двух значений. *Замечание:* поскольку каждая кость может показать целое значение от 1 до 6, то сумма двух чисел может варьироваться от 2 до 12 с наиболее частым значением суммы 7 и наименее частыми значениями 2 и 12. Рис. 4.24 показывает 36 возможных комбинаций двух костей. Ваша программа должна выбрасывать две кости 36000 раз. Используйте одномерный массив, чтобы подсчитывать, сколько раз выпадает каждая возможная сумма. Напечатайте результат в табулированном формате. Определите приемлемость полученных результатов: поскольку существует шесть возможных способов выпадения 7, приблизительно в одной шестой части всех случаев бросания должно выпадать 7.

|   | 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. 4.24.** 36 возможных исходов бросания двух костей

- 4.18.** Что делает следующая программа?

```
#include <iostream.h>

int whatIsThis(int [], int);
main()
{
 const int arraySize = 10;
 int a[arraySize] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

 int result = whatIsThis(a, arraySize);
```

```

 cout << "Результат равен " << result << endl;
 return 0;
 }
 int whatIsThis(int b[], int size)
 {
 if (size == 1)
 return b[0];
 else
 return b[size - 1] + whatIsThis(b, size - 1);
 }
}

```

**4.19.** Напишите программу, которая выполняет 1000 игр в крепс и отвечает на следующие вопросы:

- Сколько игр выиграно при первом бросании, при втором бросании, ..., при двадцатом бросании, после двадцатого бросания?
- Сколько игр проиграно при первом бросании, при втором бросании, ..., при двадцатом бросании, после двадцатого бросания?
- Каковы шансы на выигрыш в крепс? (Замечание: вы должны учесть, что крепс — одна из наиболее популярных игр в казино. Как вы думаете, что бы это значило?)
- Какова средняя продолжительность игры в крепс?
- Растут ли шансы выигрыша с увеличением продолжительности игры?

**4.20.** (Система резервирования билетов авиакомпании) Небольшая авиакомпания купила компьютеры для своей новой автоматизированной системы резервирования. Вас попросили запрограммировать новую систему. Вы должны написать программу выделения мест на каждый полет единственного самолета (вместимость: 10 мест).

Ваша программа должна отображать следующее меню альтернатив:

Ведите, пожалуйста, 1 для "курящих"  
Ведите, пожалуйста, 2 для "некурящих"

Если клиент ввел 1, ваша программа должна выделять место в салоне для курящих (места 1–5). Если клиент ввел 2, ваша программа должна выделять место в салоне для некурящих (места 6–10). Ваша программа должна также напечатать посадочный талон, указывающий номер места клиента и тип салона в самолете — для курящих или некурящих.

Используйте одномерный массив для представления схемы расположения мест в самолете. Присвойте всем элементам массива нулевые начальные значения, чтобы показать, что все места свободны. Как только место выделено пассажиру, устанавливайте соответствующие элементы массива в состояние 1, чтобы показать, что место уже занято.

Ваша программа, конечно, никогда не должна выделять уже занятые места. Если салон для курящих заполнен, ваша программа должна спросить у клиента, приемлем ли для него салон для некурящих. Если да, то сделайте выделение соответствующего места. Если нет, то напечатайте сообщение «Следующий полет состоится через три часа».

**4.21.** Что делает следующая программа?

```
#include <iostream.h>

void someFunction(int [], int);
main()
{
 const int arraySize = 10;
 int a[arraySize] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};

 cout << "Значения массива: " << endl;
 someFunction(a, arraySize);
 cout << endl;
 return 0;
}

void someFunction(int b[], int size)
{
 if (size > 0) {
 someFunction(&b[1], size - 1);
 cout << b[0] << " ";
 }
}
```

**4.22.** Используйте двумерный массив для решения следующей задачи. Компания имеет четырех продавцов (их номера с 1 по 4), которые продают 5 разных продуктов (их номера с 1 по 5). Раз в день каждый продавец заносит в регистрационную карточку (отдельную для каждого типа проданных продуктов) следующие сведения:

1. Номер продавца.
  2. Номер продукта.
  3. Общую выручку в долларах за проданный в этот день продукт.
- Таким образом, каждый продавец заполняет от 0 до 5 карточек продажи в день. Допустим, что в наличии имеется информация обо всех регистрационных карточках за последний месяц. Напишите программу, которая считывает всю эту информацию о продажах за последний месяц и суммирует общую продажу по продуктам и по продавцам. Все итоги должны храниться в двумерном массиве sales. После обработки всей информации за последний месяц, напечатайте результат в табулированном формате, представляя в каждом столбце отдельного продавца и в каждой строке отдельный продукт. Общий итог каждой строки должен давать сумму продаж каждого продукта за последний месяц; общий итог каждого столбца должен давать сумму продаж каждого продавца за последний месяц. Ваши табулированные выходные данные должны включать эти итоги справа от суммируемых строк и внизу суммируемых столбцов.

**4.23.** (*Траектории черепахи*) Язык Лого, особенно популярный среди пользователей персональных компьютеров, сделал известной идею траекторий черепахи. Представьте себе механическую черепаху, которая ползает по комнате под управлением программы на C++. Черепаха несет пишущее перо, которое может находиться в одной из двух позиций — нижней или верхней. Если перо в нижней позиции,

черепаха вычерчивает траекторию движения, если в верхней, то черепаха передвигается свободно и ничего не вычерчивает. В этой задаче вы будете моделировать действия черепахи и создавать компьютеризованный эскиз пути.

Используйте массив `floor` размером 20 на 20 с нулевыми начальными условиями. Считывайте команды из содержащего их массива. Все время отмечайте текущую позицию черепахи и положение пера — нижнее или верхнее. Предполагайте, что черепаха всегда стартует из позиции 0, 0 на полу с верхним положением пера. Ваша программа должна подавать команды черепахе в соответствии со следующими обозначениями:

| Команда | Значение                                                  |
|---------|-----------------------------------------------------------|
| 1       | Перо вверху                                               |
| 2       | Перо внизу                                                |
| 3       | Поворот направо                                           |
| 4       | Поворот налево                                            |
| 5,10    | Передвижение вперед на 10 шагов (или на иное число шагов) |
| 6       | Печать массива 20 на 20                                   |
| 9       | Конец данных (сигнальная метка)                           |

Предположим, что черепаха находится где-то возле центра комнаты. Следующая «программа управления черепахой» начертила бы квадрат 12 на 12 и оставила бы перо в верхней позиции:

```

2
5,12
3
5,12
3
5,12
3
5,12
1
6
9

```

Если черепаха передвигается с пером, находящимся в нижней позиции, устанавливайте соответствующие элементы массива `floor` равными 1. При подаче команды 6 (печать) отображайте звездочкой или каким-либо другим символом все значения 1 в массиве, где бы они ни были. Все нули, где бы они ни были, отобразите пробелами. Напишите программу, реализующую рассмотренные возможности отображения траектории передвижения черепахи. Добавьте другие команды для повышения мощности вашего языка управления траекторией черепахи.

**4.24. (Путешествие коня)** Одной из наиболее интересных шахматных головоломок является задача о путешествии коня, впервые предложенная математиком Эйлером. Вопрос заключается в следующем: может ли шахматная фигура, называемая конем, обойти все 64 клет-

ки шахматной доски, побывав на каждой из них только один раз. Рассмотрим эту интересную задачу более подробно.

Конь ходит L-образно (на две клетки в каком-либо направлении и затем на одну клетку в перпендикулярном направлении). Таким образом, из клетки в середине пустой доски конь может сделать восемь разных ходов, (пронумерованных от 0 до 7) как показано на рис. 4.25.

- a) Нарисуйте на листе бумаги шахматную доску 8 на 8 и попытайтесь выполнить путешествие коня вручную. Пометьте цифрой 1 первую клетку, куда вы ходите конем, цифрой 2 вторую, цифрой 3 третью и т.д. Перед началом путешествия определите, на сколько ходов вперед вы будете думать, памятую о том, что полное путешествие состоит из 64 ходов. Как далеко вы уйдете? Что препятствует вашим планам?
- b) Теперь разработайте программу передвижения коня по шахматной доске. Доску представим двумерным массивом **board** 8 на 8. Каждой клетке дадим нулевое начальное значение. Опишем каждый из восьми возможных ходов в терминах их горизонтальной и вертикальной компонент. Например, ход типа 0, как показано на рис. 4.25, содержит перемещение на две клетки горизонтально направо и на одну клетку вертикально вверх. Ход 2 состоит из перемещения на одну клетку горизонтально налево и на две клетки вертикально вверх. Горизонтальные перемещения налево и вертикальные перемещения вверх будем отмечать отрицательными числами. Восемь ходов, которые могли бы быть описаны двумя одномерными массивами, **horizontal** и **vertical**, выглядят следующим образом:

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 |   |   |   |   |   |   |   |   |
| 1 |   |   |   | 2 |   | 1 |   |   |
| 2 |   |   | 3 |   |   |   | 0 |   |
| 3 |   |   |   |   | K |   |   |   |
| 4 |   |   | 4 |   |   |   | 7 |   |
| 5 |   |   |   | 5 |   | 6 |   |   |
| 6 |   |   |   |   |   |   |   |   |
| 7 |   |   |   |   |   |   |   |   |

Рис. 4.25. Восемь возможных ходов конем

```

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 строку и столбец текущей позиции коня. Чтобы сделать ход типа `moveNumber`, где `moveNumber` — число от 0 до 7, ваша программа использует операторы

```

currentRow += vertical[moveNumber];
currentColumn += [moveNumber];

```

Ведите счетчик, который изменяется от 1 до 64. Записывайте последний номер каждой клетки, на которую передвинулся конь. Помните, что для контроля каждого возможного хода конем нужно видеть, был ли уже конь на этой клетке. И, конечно, проверяйте каждый возможный ход, чтобы быть уверенным в том, что конь не вышел за пределы доски. Теперь напишите программу передвижения коня по доске. Запустите программу. Сколько ходов сделал конь?

с) После попытки написать и запустить программу путешествия коня вы, вероятно, получили более глубокие представления о задаче. Вы будете использовать их для создания **эвристики** (или стратегии) передвижения коня. Эвристика не гарантирует успеха, но при тщательной разработке обычно существенно повышает шансы на успех. Вы можете заметить, что клетки на краях доски более трудны для обхода, чем клетки в центре доски. Наиболее трудны для обхода или даже недоступны четыре угловые клетки.

Интуиция может подсказать вам, что в первую очередь нужно попытаться обойти конем наиболее трудные клетки и оставить «на потом» те, доступ к которым проще, чтобы когда доска окажется к концу путешествия заполненной сделанными ходами, было больше шансов на успех.

Мы можем разработать «эвристику доступности» путем классификации каждой клетки в соответствии с ее доступностью (в терминах хода конем, конечно) и перемещения коня на наиболее недоступную клетку. Мы пометим двумерный массив **accessibility** числами, указывающими, со скольких клеток доступна каждая клетка. На пустой доске каждая центральная клетка оценивается как 8, а каждая

угловая клетка как 2, остальные клетки имеют числа доступности 3, 4 или 6 в соответствии с таблицей:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |
| 3 | 4 | 6 | 6 | 6 | 4 | 3 |   |
| 4 | 6 | 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 | 4 | 3 |   |
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |

Теперь напишите вариант программы Путешествие Коня, используя эвристику доступности. В любом случае конь должен ходить на клетку с наименьшим числом доступности. В случае равенства чисел доступности для разных клеток конь может ходить на любую из них. Таким образом, путешествие можно начать в любом из четырех углов. (Замечание: По мере перемещения коня по доске ваша программа должна уменьшать числа доступности тем больше, чем больше клеток оказываются занятыми. Таким образом, в каждый данный момент путешествия число доступности каждой имеющейся в распоряжении клетки будет делаться равным количеству клеток, из которых можно пойти на данную клетку.) Выполните эту версию вашей программы. Смогли ли вы совершить полное путешествие? Теперь модифицируйте программу для выполнения 64 путешествий, каждое из которых начинается со своей клетки шахматной доски. Сколько полных путешествий удалось сделать?

d) Напишите версию программы Путешествие Коня, которая при встрече с двумя или более альтернативными клетками с равными числами доступности решала бы, какую клетку выбрать, просматривая вперед достижимость клеток из этих альтернативных клеток. Ваша программа должна ходить на клетку, для которой следующий ход достигал бы клетки с наименьшим числом доступности.

4.25. (*Путешествие коня: методы решения «в лоб»*) В упражнении 4.24 вы разрабатывали решение задачи о Путешествии Коня. Использованный подход, названный «эвристикой доступности», генерирует множество решений и работает эффективно.

С возрастанием мощности компьютеров мы получаем возможность решать больше проблем за счет только мощности компьютера, не прибегая к изощренным алгоритмам. Назовем такой подход методом решения проблемы «в лоб» или жестким силовым вариантом.

a) Используйте генерацию случайного числа для предоставления коню возможности ходить по шахматной доске случайным образом (конечно, только допустимыми для коня ходами). Ваша программа должна запускать путешествие и печатать окончательный вид шахматной доски. Насколько далеко смог пойти конь?

b) Наиболее вероятно, что предыдущая программа совершил относительно короткое путешествие. Теперь модифицируйте вашу программу так, чтобы она сделала 1000 попыток путешествия. Используйте одномерный массив для регистрации количества путешествий каждой длины. По окончании 1000 попыток путешествия программа

должна напечатать эту информацию в строгом табулированном формате. Каков наилучший результат?

с) Наиболее вероятно, что предыдущая программа даст вам несколько «приличных», но не полных путешествий. Теперь «проигнорируйте все остановы» и просто позвольте вашей программе выполняться до получения полного путешествия. (*Предупреждение:* эта версия программы может выполнятся часами даже на мощном компьютере.) Снова заведите таблицу для регистрации количества путешествий каждой длины и напечатайте эту таблицу, как только будет выполнено первое полное путешествие. Сколько путешествий попыталась совершить программа перед выполнением полного путешествия? Сколько времени это заняло?

d) Сравните жесткий силовой вариант Путешествия Коня с вариантом эвристики доступности. Какой из них требует более тщательного изучения проблемы? Разработка какого алгоритма более трудна? Какой из них требует более высокой мощности компьютера? Могли бы вы заранее быть уверенным в выполнении полного путешествия на основе эвристики доступности? Могли бы вы заранее быть уверенным в выполнении полного путешествия на основе жесткого силового подхода? Аргументируйте доводы за и против методов решения проблемы «в лоб» вообще.

**4.26. (Восемь Ферзей)** Другой шахматной головоломкой является задача о Восьми Ферзях: можно ли поставить на пустой шахматной доске восемь ферзей так, чтобы ни один из них не «атаковал» другого, т.е. никакие два ферзя не стояли бы на одном и том же столбце или на одной и той же строке или на одной и той же диагонали? Используйте размышления, приведенные в упражнении 4.24, чтобы сформулировать эвристику для решения задачи о Восьми Ферзях. Запустите вашу программу. (*Совет:* можно присвоить значение каждой клетке шахматной доски, указывая, сколько клеток пустой шахматной доски «исключается», если ферзя поместить на эту клетку. Каждому углу должно быть присвоено значение 22, как на рис. 4.26.) Как только эти «числа исключения» будут присвоены всем 64 клеткам, можно предложить эвристику: ставить каждого следующего ферзя на клетку с наименьшим числом исключения. Почему эта стратегия интуитивно привлекательна?

**Рис. 4.26.** 22 клетки, исключаемые при размещении ферзя в верхней левой клетке

- 4.27.** (*Восемь ферзей: методы решения «в лоб»*) В этом упражнении вы будете развивать методы решения «в лоб» задачи о Восьми Ферзях, с которой вы познакомились в упражнении 4.26.
- Решите задачу о Восьми Ферзях, используя технику случайного «лобового» подхода, развитую в упражнении 4.25.
  - Используйте исчерпывающий «лобовой» подход, т.е. попробуйте все возможные комбинации восьми ферзей на шахматной доске.
  - Почему вы полагаете, что исчерпывающий «лобовой» вариант может не подойти для решения задачи о Восьми Ферзях?
  - Сравните и сопоставьте случайный «лобовой» подход и исчерпывающий «лобовой» подход в целом.
- 4.28.** (*Путешествие Коня: проверка замкнутости путешествия*) В Путешествии Коня полное путешествие означает, что конь сделал 64 хода, проходя каждую клетку шахматной доски один и только один раз. Незамкнутое путешествие имеет место тогда, когда 64-й ход — это ход вдали от места, в котором конь начал путешествие. Модифицируйте программу Путешествие Коня, которую вы написали в упражнении 4.24, чтобы проверить, является ли выполненное полное путешествие замкнутым.
- 4.29.** (*Решето Эратосфена*) Простое число — это любое целое число, которое точно делится без остатка только само на себя и на 1. Решето Эратосфена — это способ нахождения простых чисел. Он работает следующим образом:
- Создайте массив, все элементы которого имеют начальные значения 1 (истина). Элементы массива с простыми индексами останутся равными 1. Все другие элементы массива в конечном счете становятся равными нулю.
  - Начиная с индекса массива 2 (индекс 1 должен быть простым), каждый раз отыскивается элемент массива с единичным значением, циклически обрабатывается оставшаяся часть массива и устанавливается в нуль каждый элемент массива, чей индекс кратен индексу элемента с единичным значением. Для индекса 2 все элементы в массиве с большим чем 2 индексом и кратные 2 устанавливаются равными нулю (индексы 4, 6, 8 и тому подобные); для индекса 3 все элементы с индексом выше 3 и кратные 3, устанавливаются равными нулю (индексы 6, 9, 12 и тому подобные) и т.д.
- Когда процесс закончится, элементы массива с единичным значением указывают, что их индексы — простые числа. Эти индексы могут быть напечатаны. Напишите программу, которая использует массив с 1000 элементами для определения и печати простых чисел между 1 и 999. Элемент 0 массива во внимание не принимайте.
- 4.30.** (*Блочная сортировка*) Блочная сортировка требует наличия одномерного массива положительных целых чисел, который нужно сортировать, и двумерного массива целых чисел со строками, проиндексированными от 0 до 9, и столбцами, проиндексированными от 0 до  $n - 1$ , где  $n$  — количество значений в массиве, который должен сортироваться. Каждая строка двумерного массива рассматривается

как блок. Напишите функцию `bucketSort`, которая принимает массив целых чисел и его размер как аргументы и выполняет следующее:

- 1) Поместите каждое значение одномерного массива в строку массива блоков, основываясь на значении его первого разряда. Например, 97 помещается в строку 7, 3 помещается в строку 3, а 100 помещается в строку 0. Это называется «распределяющий проход».
- 2) Циклически обработайте массив блоков строка за строкой и скопируйте значения обратно в исходный массив. Это называется «собирающий проход». Новый порядок предыдущих значений в одномерном массиве будет 100, 3 и 97.
- 3) Повторите этот процесс для каждого последовательного разряда (десятки, сотни, тысячи и тому подобное).

На втором проходе 100 разместится в строке 0, 3 разместится в строке 0 (потому что 3 не имеет разряда десятков), а 97 разместится в строке 9. На третьем проходе 100 поместится в строке 1, 3 поместится в нулевой строке и 97 поместится в нулевой строке (после цифры 3). После последнего собирающего прохода исходный массив будет отсортирован.

Заметим, что двумерный массив блоков в десять раз больше размера массива, который сортируется. Эта техника сортировки обеспечивает более высокую производительность по сравнению с пузырьковой, но требует много больше памяти. Для пузырьковой сортировки требуется всего один дополнительный элемент данных. Это пример дилеммы память-время: блочная сортировка использует больше памяти, чем пузырьковая, но работает лучше. Другая возможность заключается в создании двумерного массива блоков и повторного обмена данными между двумя массивами блоков.

## Упражнения на рекурсию

**4.31. (Сортировка отбором).** Сортировка отбором анализирует массив, отыскивая наименьший элемент массива. Затем этот наименьший элемент обменивается местами с первым элементом массива. Процесс повторяется для подмассива, начинающегося со второго элемента массива. В результате каждого прохода один из элементов занимает соответствующее место. Эта сортировка по производительности сравнима с пузырьковой — для массива из  $n$  элементов нужно выполнить  $n - 1$  проход, а для каждого подмассива нужно выполнить  $n - 1$  сравнение для определения наименьшего значения. Когда обрабатываемый подмассив будет содержать только один элемент, значит массив отсортирован. Напишите рекурсивную функцию `selectionSort`, выполняяющую этот алгоритм.

**4.32. (Палиндромы)** Палиндром — это строка, которая читается одинаково от начала и от конца. Вот некоторые примеры палиндромов: «радар», «потол», «а роза упала на лапу азора» (если игнорировать пробелы) и т. д. Напишите рекурсивную функцию `testPalindrome`, которая возвращает 1, если хранящаяся в массиве строка — палиндром, и 0 в противном случае.

- 4.33. (Линейный поиск)** Модифицируйте программу на рис. 4.19 так, чтобы использовать рекурсивную функцию `linearSearch` для линейного поиска в массиве. Функция должна принимать массив целых чисел и размер массива как аргументы. Если ключ поиска найден, верните индекс массива, в противном случае верните -1.
- 4.34. (Двоичный поиск)** Модифицируйте программу на рис. 4.19 так, чтобы использовать рекурсивную функцию `binarySearch` для двоичного поиска в массиве. Функция должна принимать массив целых чисел и начальный и конечный индексы как аргументы. Если ключ поиска найден, верните индекс массива, в противном случае верните -1.
- 4.35. (Восемь Ферзей)** Модифицируйте программу Восемь Ферзей, которую вы создали в упражнении 4.26, для рекурсивного решения задачи.
- 4.36. (Печать массива)** Напишите рекурсивную функцию `printArray`, которая принимает массив и размер массива как аргументы и ничего не возвращает. Функция должна прекращать свою работу и возвращаться, если принимаемый массив имеет нулевой размер.
- 4.37. (Печать строки в обратном направлении)** Напишите рекурсивную функцию `stringReverse`, которая принимает символьный массив, содержащий строку, как аргумент, печатает строку в обратном направлении и ничего не возвращает. Функция должна прекращать свою работу и возвращаться, если обнаружен завершающий нулевой символ.
- 4.38. (Поиск минимального значения в массиве)** Напишите рекурсивную функцию `recursiveMinimum`, которая принимает массив и размер массива как аргументы и возвращает наименьший элемент массива. Функция должна прекращать свою работу и возвращаться, если принимаемый массив имеет один элемент.

# 5

## Указатели и строки



### Ц е л и

- Научиться использовать указатели.
- Научиться использовать указатели для передачи аргументов в функции по ссылке.
- Понять тесную взаимосвязь между указателями, массивами и строками.
- Понять, как использовать указатели на функции.
- Научиться объявлять и использовать массивы строк.

## План

- 5.1. Введение**
- 5.2. Объявления и инициализация переменных указателей**
- 5.3. Операции над указателями**
- 5.4. Вызов функций по ссылке**
- 5.5. Использование спецификатора *const* с указателями**
- 5.6. Пузырьковая сортировка, использующая вызов по ссылке**
- 5.7. Выражения и арифметические действия с указателями**
- 5.8. Взаимосвязи между указателями и массивами**
- 5.9. Массивы указателей**
- 5.10. Учебный пример: моделирование тасования и раздачи карт**
- 5.11. Указатели на функции**
- 5.12. Введение в обработку символов и строк**
  - 5.12.1. Основы теории символов и строк**
  - 5.12.2. Функции работы со строками из библиотеки обработки строк**
- 5.13. Размышления об объектах: взаимодействие объектов**

*Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения • Специальный раздел: построение вашего собственного компьютера • Дополнительные упражнения на указатели • Упражнения на работу со строками • Специальный раздел: упражнения повышенной сложности на работу со строками • Головоломный проект работы со строками*

## 5.1. Введение

В этой главе мы обсудим одно из наиболее мощных свойств языка программирования C++ — *указатели*. Указатели — одна из наиболее трудных для освоения возможностей C++. В главе 3 мы видели, что для передачи параметров по ссылке можно использовать ссылки. Указатели предоставляют программам возможность моделировать передачу по ссылке и создавать и манипулировать динамическими структурами данных, т. е. структурами данных, которые могут нарастать и сокращаться, например, такими, как связные списки, очереди, стеки и деревья. В данной главе объясняются основные концепции указателей. Эта глава также углубляет наши представления о взаимосвязи между массивами, указателями и строками и включает тщательно подобранный набор упражнений на обработку строк.

В главе 6 исследуется использование указателей со структурами. Глава 15 знакомит с техникой управления динамической памятью и содержит примеры создания и использования динамических структур данных.

## 5.2. Объявления и инициализация переменных указателей

Указатели — это переменные, которые содержат в качестве своих значений адреса памяти. С другой стороны, указатель содержит адрес переменной, которая содержит определенное значение. В этом смысле имя переменной отсылает к значению *прямо*, а указатель — *косвенно* (рис. 5.1). Ссылка на значение посредством указателя называется *косвенной адресацией*.

Указатели, подобно любым другим переменным, перед своим использованием должны быть объявлены. Объявление

```
int *countPtr, count;
```

объявляет переменную `countPtr` типа `int *` (т.е. указатель на целое число) и читается как «`countPtr` является указателем на целое число» или «`countPtr` указывает на объект типа `int`». Однако переменная `count` объявлена как целое число, но не как указатель на целое число. Символ `*` в объявлении относится только к `countPtr`. Каждая переменная, объявляемая как указатель, должна иметь перед собой знак звездочки (`*`). Например, объявление

```
float *xPtr, *yPtr;
```

указывает, что и `xPtr` и `yPtr` являются указателями на значения типа `float`. Использование `*` подобным образом в объявлении показывает, что переменная объявляется как указатель. Указатели можно объявлять, чтобы указывать на объекты любого типа данных.

### Типичная ошибка программирования 5.1

Операция `*` косвенной адресации не распространяется на все имена переменных в объявлении. Каждый указатель должен быть объявлен с помощью символа `*`, стоящего перед именем.

### Хороший стиль программирования 5.1

Хотя это и не обязательно, включайте буквы `Ptr` в имена переменных указателей, чтобы было ясно, что эти переменные являются указателями и требуют соответствующей обработки.

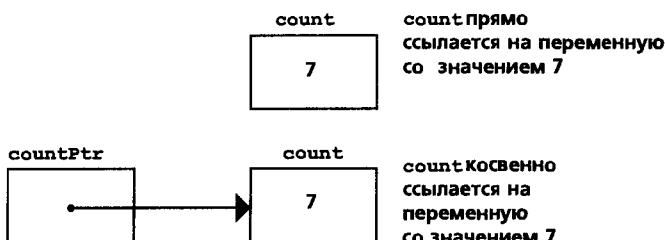


Рис. 5.1. Прямая и косвенная ссылки на переменную

Указатели должны инициализироваться либо при своем объявлении, либо с помощью оператора присваивания. Указатель может получить в качестве начального значения 0, **NULL** или адрес. Указатель с начальными значениями 0 или **NULL** ни на что не указывает. **NULL** — это символическая константа, определенная в заголовочном файле (*iostream.h* и в нескольких заголовочных файлах стандартной библиотеки С). Задание **NULL** как начального значения эквивалентно заданию начального значения 0, но в C++ 0 предпочтительнее. Если присваивается 0, то он преобразуется в указатель соответствующего типа. Значение 0 — единственное целое значение, которое можно прямо присваивать как значение указателю без предварительного приведения целого числа к типу указателя. Присваивание адреса переменной указателю обсуждается в разделе 5.3.

### Хороший стиль программирования 5.2

Присваивайте начальные значения указателям во избежание неожиданных результатов.

## 5.3. Операции над указателями

**&** или *операция адресации*, — унарная операция, которая возвращает адрес своего операнда. Например, если имеются объявления

```
int y = 5;
int *yPtr;
```

то оператор

```
yPtr = &y;
```

присваивает адрес переменной *y* указателю *yPtr*. Говорят, что переменная *yPtr* «указывает» на *y*. Рисунок 5.2 показывает схематическое представление памяти после того, как выполнено указанное выше присваивание. На рисунке показана «связь указателя» с помощью стрелки от указателя к объекту, на который он указывает.

Рисунок 5.3 показывает представление указателя в памяти в предположении, что целая переменная *y* хранится в ячейке 600000, а переменная указатель *yPtr* хранится в ячейке 500000. Операнд операции адресации должен быть L-величиной (т.е. чем-то таким, чему можно присвоить значение так же, как переменной); операция адресации не может быть применена к константам, к выражениям, не дающим результат, на который можно ссылаться, и к переменным, объявленным с классом памяти *register*.

Операция **\***, обычно называемая *операцией косвенной адресации* или *операцией разыменования*, возвращает значение объекта, на который указывает ее operand (т.е. указатель). Например, (снова соплемся на рис. 5.2) оператор

```
cout << *yPtr << endl;
```

печатает значение переменной *y*, а именно 5. Использование **\*** указанным способом называется *разыменованием указателя*.

### Типичная ошибка программирования 5.2

Разыменование указателя, который не был должным образом инициализирован или которому не присвоено указание на конкретное место в памяти. Это может вызвать неисправимую ошибку выполнения или может неожиданно изменить важные данные и программа завершится неверными результатами.

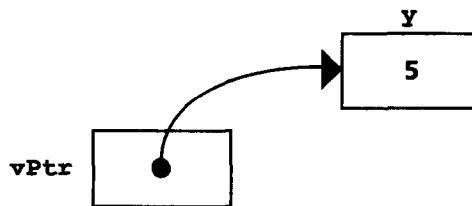


Рис. 5.2. Графическое представление указателя, указывающего на целую переменную в памяти

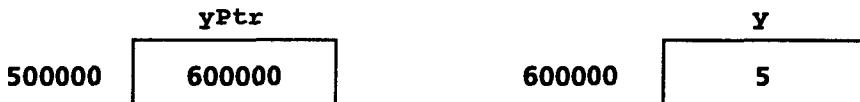


Рис. 5.3. Представление **y** и **yPtr** в памяти

Программа на рис. 5.4 демонстрирует операции с указателями. Ячейки памяти выводятся как шестнадцатиричные целые с помощью `cout` (смотри приложение Г, «Системы счисления», для получения большей информации о шестнадцатиричных целых). Заметим, что адрес `a` и значение `aPtr` идентичны; это подтверждает, что адрес `a` действительно присвоен переменной указателю `aPtr`. Операции `&` и `*` взаимно дополняют друг друга — при их поочередном применении к `aPtr` в любой последовательности будет напечатан один и тот же результат. Таблица на рис. 5.5 показывает приоритет и ассоциативность операций, рассмотренных к данному моменту.

```
// Использование операций & и *
#include <iostream.h>

main()
{
 int a; // a - целое число
 int *aPtr; // aPtr - указатель на целое число

 a = 7;
 aPtr = &a; // aPtr устанавливается равным адресу a

 cout << "Адрес a: " << &a << endl
 << "Значение aPtr: " << aPtr << endl << endl;

 cout << "Значение a: " << a << endl
 << "Значение *aPtr: " << *aPtr << endl << endl;

 cout << "Доказательство, что & и * дополняют "
 << "друг друга. " << endl << "&*aPtr = " << &*aPtr
 << endl << "*&aPtr = " << *&aPtr << endl;
 return 0;
}
```

Рис. 5.4. Операции `&` и `*` над указателями (часть 1 из 2)

```

Адрес a: 0xffff4
Значение aPtr: 0xffff4

Значение a: 7
Значение *aPtr: 7

Доказательство, что & и * дополняют друг друга.
&*aPtr = 0xffff4
*&aPtr = 0xffff4

```

Рис. 5.4. Операции &amp; и \* над указателями (часть 2 из 2)

| Операции             | Ассоциативность | Тип                    |
|----------------------|-----------------|------------------------|
| () []                | слева направо   | наивысший              |
| + - + -- ! * & (тип) | справа налево   | унарные                |
| * / %                | слева направо   | мультипликативные      |
| + -                  | слева направо   | аддитивные             |
| << >>                | слева направо   | поместить в/взять из   |
| < <= > >=            | слева направо   | отношение              |
| == !=                | слева направо   | проверка на равенство  |
| &&                   | слева направо   | логическое И           |
|                      | слева направо   | логическое ИЛИ         |
| ?:                   | справа налево   | условная               |
| = += -= *= /= %=     | справа налево   | присваивание           |
| ,                    | слева направо   | запятая (последование) |

Рис. 5.5. Приоритет операций

## 5.4. Вызов функций по ссылке

Существуют три способа передачи аргументов в функцию — *вызов по значению*, *вызов по ссылке с аргументами ссылками* и *вызов по ссылке с аргументами указателями*. В главе 3 мы сравнивали и сопоставляли вызовы по значению и по ссылке с аргументами ссылки. В этой главе мы сосредоточимся на вызове по ссылке с аргументами указателями.

Как мы видели в главе 3, `return` можно использовать для возвращения одного значения из вызываемой функции вызывающему оператору (или для передачи управления из вызываемой функции без возвращения какого-либо значения). Мы также видели, что аргументы могут быть переданы функции с использованием аргументов ссылок, чтобы дать возможность функции модифицировать исходные значения аргументов (таким образом, из функции может быть «возвращено» более одного значения), или чтобы передать функции большие объекты данных и избежать накладных расходов, сопутствующих передаче объектов вызовом по значению (которая требует копирования объекта). Указатели, подобно ссылкам, тоже можно использовать для манипуляции данными.

фикации одного или более значений переменных в вызывающем операторе, или передавать указатели на большие объекты данных, чтобы избежать накладных расходов, сопутствующих передаче объектов по значению.

В C++ программисты могут использовать указатели и операции косвенной адресации для моделирования вызова по ссылке. При вызове функции с аргументами, которые должны быть модифицированы, передаются адреса аргументов. Это обычно сопровождается операцией адресации (&) переменной, которая должна быть модифицирована. Как мы видели в главе 4, массивы не передаются с использованием операции &, потому что имя массива — это начальный адрес массива в памяти (имя массива эквивалентно `&arrayName[0]`). При передаче функции адреса переменной может использоваться операция косвенной адресации (\*) для модификации значения (если значение не объявлено как `const`) ячейки в памяти вызывающего оператора.

Программы на рис. 5.6 и 5.7 представляют два варианта функции, которая возводит в куб целое число — `cubeByValue` и `cubeByReference`. Программа на рис. 5.6, передает переменную `number` функции `cubeByValue` вызовом по значению. Функция `cubeByValue` возводит в куб свой аргумент и возвращает новое значение в `main`, используя оператор `return`. Новое значение присваивается переменной `number` в `main`. Ключевой момент вызова по значению состоит в том, что вы имеете возможность анализировать результат вызова функции перед модификацией значения переменной. Например, в этой программе мы могли бы сохранить результат `cubeByValue` в другой переменной, исследовать его значение, и после этой проверки присвоить результат переменной `number`.

Программа на рис. 5.7 передает переменную `number` по ссылке — в функцию `cubeByReference` передается адрес `number`. Функция `cubeByReference` в качестве аргумента получает `nPtr` (указатель на `int`). Функция разыменовывает указатель и возводит в куб значение, на которое указывает `nPtr`. Это изменяет значение `number` в `main`. Рисунки 5.8 и 5.9 графически анализируют программы на рис. 5.6 и 5.7 соответственно.

```
// Возведение переменной в куб с использованием вызова по значению
#include <iostream.h>

int cubeByValue(int); //прототип

main()
{
 int number = 5;

 cout << "Исходное значение числа: " << number << endl;
 number = cubeByValue(number);
 cout << "Новое значение числа: " << number << endl;
 return 0;
}
int cubeByValue(int n)
{
 return n * n * n; // куб локальной переменной n
}
```

Исходное значение числа: 5

Новое значение числа: 125

Рис. 5.6. Возведение переменной в куб с использованием вызова по значению

```
// Возвведение переменной в куб с использованием вызова по ссылке
// с аргументом указателем
#include <iostream.h>

void cubeByReference(int *); //прототип

main()
{
 int number = 5;

 cout << "Исходное значение числа: " << number << endl;
 cubeByReference(&number);
 cout << "Новое значение числа: " << number << endl;
 return 0;
}
void cubeByReference(int *nPtr)
{
 *nPtr = *nPtr * *nPtr * *nPtr; // куб числа в main
}
```

**Исходное значение числа: 5**

**Новое значение числа: 125**

**Рис. 5.7.** Возвведение переменной в куб с использованием вызова по ссылке с аргументом указателем

### Типичная ошибка программирования 5.3

Не разыменовывается указатель, когда это необходимо сделать, чтобы получить значение, на которое указывает этот указатель.

Функция, принимающая адрес в качестве аргумента, должна определить параметр как указатель, чтобы принять адрес. Например, заголовок функции `cubeByReference` имеет вид

```
void cubeByReference(int *hPtr)
```

Этот заголовок указывает, что функция `cubeByReference` принимает адрес целой переменной как аргумент, сохраняет адрес локально в `nPtr` и не возвращает значение.

Прототип функции `cubeByReference` содержит `int *` в скобках. Так же, как и с другими типами переменных, включать имена указателей в прототип функции нет необходимости. Имена, включенные с целью документирования, компилятором игнорируются.

В заголовке функции и в прототипе функции, которая ожидает в качестве аргумента одномерный массив, можно использовать запись указателя в списке параметров функции. Компилятор не делает различий между функцией, которая принимает указатель, и функцией, которая принимает одномерный массив. Это означает, конечно, что функция должна «знать», принимает ли она массив или просто одну переменную, для которой выполняется передача вызовом по ссылке. Когда компилятор сталкивается с параметром функции в виде одномерного массива, например, `int b[ ]`, он преобразует параметр в запись указателя `int * const b` (произносится как «`b` — это константный указатель на целое» — `const` объясняется в разделе 5.5). Обе формы объявления параметра функции как одномерного массива равнозначны.

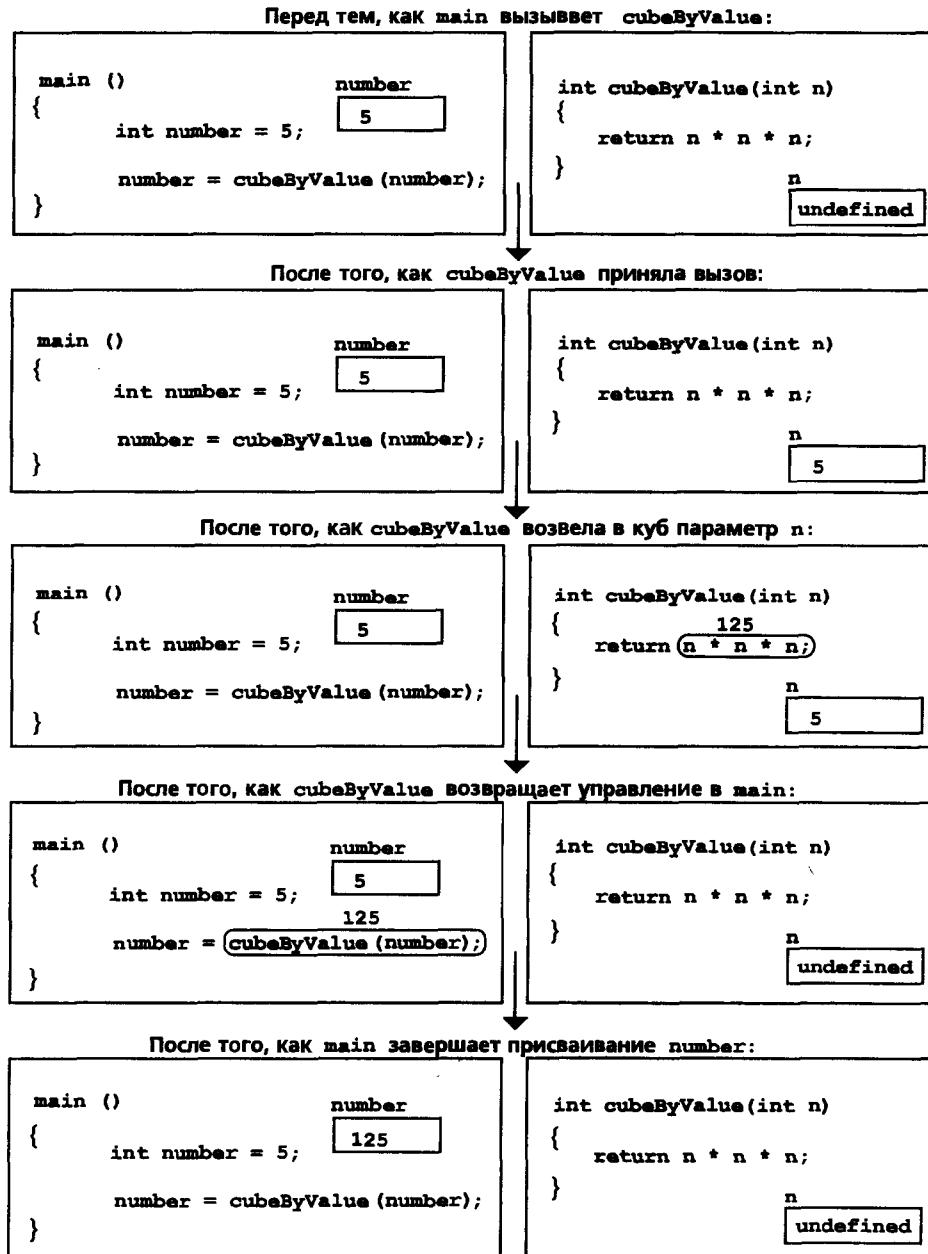


Рис. 5.8. Анализ типичного вызова по значению

**Хороший стиль программирования 5.3**

Используйте передачу по значению аргументов функции до тех пор, пока оператор вызова явно не требует, чтобы вызываемая функция модифицировала значение переменной аргумента в окружении вызывающего оператора. Это еще один пример принципа наименьших привилегий.

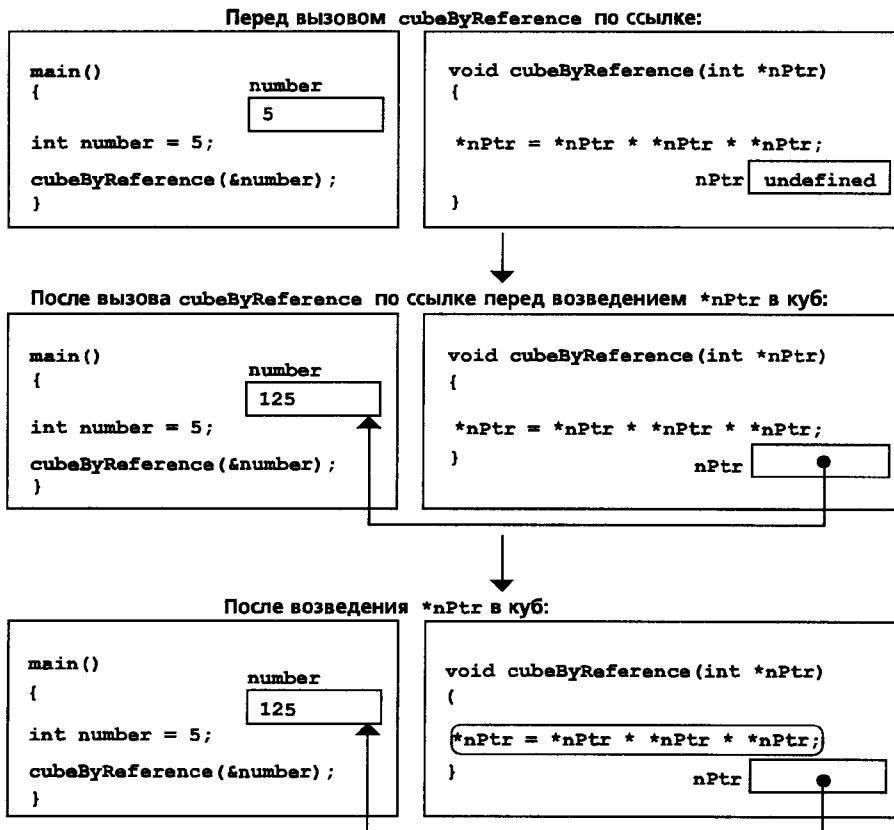


Рис. 5.9. Анализ типичного вызова по ссылке с аргументом указателем

## 5.5. Использование спецификатора **const** с указателями

Спецификация **const** дает возможность программисту информировать компилятор о том, что значение данной переменной не должно изменяться.

### Замечание по технике программирования 5.1

Спецификацию **const** можно использовать для реализации принципа минимизации привилегий. Использование должным образом принципа минимизации привилегий при проектировании программного обеспечения весьма существенно сокращает время отладки и количество нежелательных побочных эффектов, упрощает модификацию и сопровождение программы.

### Замечание по мобильности 5.1

Хотя спецификация **const** полностью определена в С ANSI, в некоторых компиляторах она не реализована.

Часто основой разработок являются доставшиеся в наследство программы, написанные на ранних версиях С, в которых спецификатор `const` не использовался, потому что его просто не было в наличии. По этой причине существуют огромные возможности для улучшения разработок программного обеспечения, написанного на старых версиях С. Хотя многие программисты, постоянно использующие С ANSI, все-таки не применяют в своих программах `const`, потому что они начинали программировать на ранних версиях С. Эти программисты упускают много возможностей по созданию хорошего программного обеспечения.

Существует шесть возможностей использования (или не использования) `const` с параметрами функции — две с передачей параметров по значению и четыре с передачей параметров по ссылке. Как выбрать одну из этих возможностей? Давайте руководствоваться принципом наименьших привилегий. Всегда обеспечивайте функцию в ее параметрах достаточным доступом к данным, чтобы она могла выполнять свои задачи, но не более того.

В главе 3 мы объясняли, что когда функция вызывается с использованием передачи параметров по значению, в вызове функции создается копия аргумента (или аргументов) и она передается функции. Если копия в функции модифицируется, исходное значение в вызывающем операторе остается без изменений. Во многих случаях значение, передаваемое функции, модифицируется для того, чтобы функция могла выполнить свою задачу. Однако, в некоторых отдельных случаях значение в вызываемой функции не должно изменяться даже несмотря на то, что она манипулирует копией исходного значения.

Рассмотрим функцию, которая получает одномерный массив и его размер как аргументы и печатает этот массив. Такая функция должна циклически обрабатывать массив и выводить значения каждого его элемента. Размер массива используется в теле функции для определения верхнего индекса массива, при котором цикл должен завершаться, так как печать окончена. Размер массива не изменяется в теле функции.

### **Замечание по технике программирования 5.2**

Если значение не изменяется (или не должно изменяться) в теле функции, которой оно передается, параметр должен быть объявлен как `const`, чтобы иметь гарантию от неожиданных его изменений.

В случае попытки изменить значение константного параметра компилятор отлавливает ее и выдает либо предупреждение, либо сообщение об ошибке в зависимости от конкретного типа компилятора.

### **Замечание по технике программирования 5.3**

При использовании вызова по значению в вызываемой функции можно изменять только одно значение. Это значение должно быть задано как возвращаемое в операторе `return`. Для модификации множества значений в вызываемой функции нужно использовать передачу параметров по ссылке.

### **Хороший стиль программирования 5.4**

Прежде, чем использовать функцию, проверьте прототип функции, чтобы определить, имеет ли она возможность модифицировать передаваемые ей значения.

Существуют четыре способа передачи в функцию указателя: неконстантный указатель на неконстантные данные, неконстантный указатель на константные данные, константный указатель на неконстантные данные и константный указатель на константные данные. Каждая комбинация обеспечивает доступ с разным уровнем привилегий.

Наивысший уровень доступа предоставляется неконстантным указателем на неконстантные данные — данные можно модифицировать посредством разыменования указателя, а сам указатель может быть модифицирован, чтобы он указывал на другие данные. Объявление неконстантного указателя на неконстантные данные не содержит `const`. Такой указатель можно применить, чтобы передать строку в функцию, которая использует операции с указателем (и, возможно, модифицирует его) для обработки каждого символа в строке. Функция `convertToUppercase` на рис. 5.10 объявляет параметр `sPtr(char *sPtr)` как неконстантный указатель на неконстантные данные. Функция обрабатывает строку `s` символ за символом, используя арифметические операции с указателем. Символы в диапазоне от '`a`' до '`z`' преобразуются в соответствующие прописные буквы с помощью функции `toupper()`, а остальные символы остаются неизменными. Функция `toupper` получает в качестве аргумента символ. Если символ — строчная буква, возвращается соответствующая прописная буква; в противном случае возвращается исходная буква. Функция `toupper` является частью библиотеки обработки символов `ctype.h` (смотри главу 16, «Символы, строки, структуры и операции с битами»). (Замечание: функция `toupper` осуществляет преобразование только латинских букв, но не символов кириллицы.)

```
// Преобразование строчных букв в прописные с использованием
// неконстантного указателя на неконстантные данные
#include <iostream.h>
#include <ctype.h>

void convertToUppercase(char *);

main()
{
 char string[] = "characters and $32.98";

 cout << "Строка перед преобразованием: " << string << endl;
 convertToUppercase(string);
 cout << "Строка после преобразования: " << string << endl;
 return 0;
}
void convertToUppercase(char *sPtr)
{
 while (*sPtr != '\0') {
 *sPtr = toupper(*sPtr); // преобразовать
 // в прописные буквы
 ++sPtr; // увеличить sPtr, чтобы указать
 // на следующий символ
 }
}
```

---

Строка перед преобразованием: characters and \$32.98  
Строка после преобразования: CHARACTERS AND \$32.98

Рис. 5.10. Преобразование строки в прописные буквы

Неконстантный указатель на константные данные — это указатель, который можно модифицировать, чтобы указывать на любые элементы данных подходящего типа, но данные, на которые он ссылается, не могут быть модифицированы. Такой указатель можно было бы использовать, чтобы передать аргументы массивы функции, которая будет обрабатывать каждый элемент массива без модификации данных. Например, функция `printCharacters` на рис. 5.11 объявляет параметры `sPtr` типа `const char *`. Объявление читается как «`sPtr` является указателем на константный символ». Тело функции использует структуру `for` для вывода каждого символа в строке до тех пор, пока не встретится нулевой символ. После того, как каждый символ будет напечатан, указатель `sPtr` увеличивается, чтобы указать на следующий символ в строке.

```
// Печать строки символ за символом с использованием
// неконстантного указателя на константные данные
#include <iostream.h>

void printCharacters(const char *);

main()
{
 char string[] = "печать символов строки";

 cout << "Строка:" << endl;
 printCharacters(string);
 cout << endl;
 return 0;
}

// В printCharacters sPtr — указатель на константный символ.
// Символы не могут модифицироваться посредством sPtr
// (т.е. sPtr — указатель "только для чтения")
void printCharacters(const char *sPtr)
{
 for (; *sPtr != '\0'; sPtr++) //нет начального условия
 cout << *sPtr;
}
```

**Строка:**  
печать символов строки

**Рис. 5.11.** Печать строки по символам с использованием непостоянного указателя на постоянные данные

Рисунок 5.12 демонстрирует сообщение об ошибке, вырабатываемое компилятором Borland C++ при попытке компилировать функцию, которая принимает неконстантный указатель на константные данные и затем пытается использовать этот указатель для модификации данных.

Как мы знаем, массивы — это агрегаты типов данных, которые хранят связанные элементы данных одинакового типа с одинаковым именем. В главе 6 мы обсудим другую форму агрегирования типов данных, называемую *структурой* (иногда называемую в других языках *записью*). Структуры способны хранить связанные элементы данных разных типов под одним именем (например, хранить информацию о каждом служащем компании). При вызове

функции с массивом как аргументом массив автоматически передается функции с помощью моделирования передачи по ссылке. Однако, структуры всегда передаются вызовом по значению — передается копия всей структуры в целом. Это требует во время выполнения накладных расходов на изготовление копии каждого элемента структуры и хранения ее в так называемом стеке (месте, где локальные переменные, используемые в вызове функции, хранятся во время ее выполнения). Когда данные структуры должны быть переданы функции, мы можем использовать указатели на константные данные, чтобы достичь производительности, характерной для передачи по ссылке, и защиты, характерной для передачи по значению. После передачи указателя на структуру должна быть сделана копия только адреса, по которому хранится структура. На машине с 4-байтовыми адресами копию адреса памяти в 4 байта выполнить предпочтительнее, чем копию возможно сотен или тысяч байтов структуры.

```
// Попытка модифицировать данные, переданные посредством
// неконстантного указателя на константные данные
#include <iostream.h>

void f(const int *);

main()
{
 int y;

 f(&y); // f пытается выполнить незаконную модификацию

 return 0;
}
// в f аргумент xPtr - указатель на целую константу
void f(const int *xPtr)
{
 *xPtr = 100; // нельзя модифицировать константный объект
}

Compiling F105_12.CPP:
Error FIG5_12.CPP 19: Cannot modify a const object
Warning FIG5_12.CPP 20 : Parameter 'xPtr' is never used
```

**Рис. 5.12.** Попытка модифицировать данные, переданные посредством неконстантного указателя на константные данные

### Совет по повышению эффективности 5.1

Передавайте большие объекты, такие, как структуры, используя указатели на константные данные или ссылки на константные данные, чтобы получить выигрыш по производительности, даваемый передачей по ссылке, и защиту информации, даваемую передачей по значению.

Константный указатель на неконстантные данные — это указатель, который всегда указывает на одну и ту же ячейку памяти, данные в которой можно модифицировать посредством указателя. Этот вариант реализуется по умолчанию для имени массива. Имя массива — это константный указатель

на начало массива. Используя имя массива и индексы массива можно обращаться ко всем данным в массиве и изменять их. Указатели, объявленные как **const**, должны получить начальные значения при своем объявлении (если указатель является параметром функции, он получает начальное значение, равное указателю, который передается в функцию). Программа на рис. 5.13 пытается модифицировать константный указатель. Указатель **ptr** объявлен имеющим тип **int \* const**. Это объявление читается как «**ptr** является константным указателем на целое число». Этот указатель получает начальное значение, равное адресу целой переменной **x**. Программа пытается присвоить **ptr** адрес **y**, но это приводит к сообщению об ошибке. Заметим, что ошибки не происходит, когда **\*ptr** присваивается значение 7, — значение, на которое указывает **ptr**, можно модифицировать.

#### Типичная ошибка программирования 5.4

Отсутствие присваивания начального значения указателю, который объявлен как **const**, приводит к ошибке компиляции.

```
// Попытка модифицировать константный указатель
// на неконстантные данные
#include <iostream.h>

main()
{
 int x, y;
 int * const ptr = &x; // ptr - константный указатель на целое.
 // Целое можно модифицировать, ссылаясь
 // на ptr, но ptr всегда указывает
 // на одну и ту же ячейку памяти.

 *ptr = 7;
 ptr = &y; //Ошибка: постоянный объект
 // модифицировать нельзя.
 return 0;
}

Compiling FIG5_13.CPP:
Error FIG5_13.CPP 14: Cannot modify a const object
Warning FIG5_13.CPP 17 : 'y' is declared but never used
```

Рис. 5.13. Попытка модифицировать константный указатель на неконстантные данные

Наименьший уровень привилегий доступа предоставляет константный указатель на константные данные. Такой указатель всегда указывает на одну и ту же ячейку памяти и данные в этой ячейке нельзя модифицировать. Это выглядит так, как если бы массив нужно было передать функции, которая только просматривает массив, использует его индексы, но не модифицирует сам массив. Программа на рис. 5.14 объявляет переменную указателя **ptr** типа **const int \* const**. Это объявление читается как «**ptr** является константным указателем на константное целое». На рисунке показаны сообщения об ошибке, генерируемые при попытке модифицировать данные, на которые указывает **ptr**, и при попытке модифицировать адрес, хранимый в переменной указателе. Заметим, что никакой ошибки не генерируется при попытке вывести значение, на которое указывает **ptr**, потому что в операторе вывода нет никаких модификаций.

```

// Попытка модифицировать константный указатель
// на константные данные
#include <iostream.h>

main()
{
 int x = 5, y;
 const int * const ptr = &x; // ptr - константный указатель
 //на константное целое. ptr всегда
 //указывает на одну и ту же ячейку
 //памяти. Целое нельзя модифицировать
 cout << *ptr << endl;
 *ptr = 7; //Ошибка: постоянный объект модифицировать нельзя.
 ptr = &y; //Ошибка: постоянный объект модифицировать нельзя.

 return 0;
}

```

**Compiling FIG5\_14.CPP:**

Error FIG5\_14.CPP 15: Cannot modify a const object  
Error FIG5\_14.CPP 16: Cannot modify a const object  
Warning FIG5\_13.CPP 19 : 'y' is declared but never used

Рис. 5.14. Попытка модифицировать константный указатель на константные данные

## 5.6. Пузырьковая сортировка, использующая вызов по ссылке

Давайте модифицируем программу пузырьковой сортировки на рис. 4.16 так, чтобы использовать две функции — `bubbleSort` и `swap` (рис. 5.15). Функция `bubbleSort` выполняет сортировку массива. Она вызывает функцию `swap`, чтобы изменить элементы массива `array[ j ]` и `array[ j + 1 ]`. Напомним, что C++ делает взаимноневидимой информацию функций, так что `swap` не имеет доступа к отдельным элементам массива в `bubbleSort`. Поскольку `bubbleSort` хочет, чтобы `swap` имела доступ к элементам массива и переставляла их, `bubbleSort` передает функции `swap` каждый из этих элементов вызовом по ссылке — явно передается адрес каждого элемента массива. Хотя массив в целом автоматически передается вызовом по ссылке, отдельные элементы массива являются скалярными величинами и передаются обычно вызовом по значению. Поэтому `bubbleSort` использует операцию адресации (&) для каждого элемента массива в вызове `swap` следующим образом

```
swap(&array[j], &array[j+1];
```

чтобы выполнить вызов по ссылке. Функция `swap` получает `&array[ j ]` в переменной указателе `element1Ptr`. Поскольку информация скрыта, `swap` не может знать имя `array[ j ]`, но `swap` может использовать `*element1Ptr` как синоним для `array[ j ]`. Таким образом, когда функция `swap` ссылается на `*element1Ptr`, она в действительности ссылается на `array[ j ]` в `bubbleSort`. Аналогичным образом, когда функция `swap` ссылается на `*element2Ptr`, она в действительности ссылается на `array[j+1]` в `bubbleSort`. Несмотря на то, что функция `swap` не имеет возможности сказать

```
temp = array[j];
array[j] = array[j+1];
array[j+1] = temp;
```

она достигает точно такого же эффекта операторами (рис. 5.15):

```
temp = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = temp;
```

Необходимо отметить несколько свойств функции **bubbleSort**. Заголовок функции объявляет **array** как **int \*array**, а не как **int array[ ]**, чтобы указать, что **bubbleSort** принимает одномерный массив как аргумент (эти записи взаимозаменяемы). Параметр **size** объявлен как **const**, чтобы задействовать принцип наименьших привилегий. Функция **bubbleSort** не нуждается в изменении **size** для выполнения своей задачи; поэтому для параметра **size** создается копия его значения в **main** и модификация этой копии не может изменить значение в **main**. Размер массива остается неизменным во время выполнения **bubbleSort**. Поэтому для гарантии того, что **size** не будет модифицирован, он объявлен как **const**. Если бы во время сортировки массива его размер изменился, это привело бы к неправильной работе алгоритма.

#### Замечание по технике программирования 5.4

Включение прототипов функции в определения других функций реализует принцип наименьших привилегий путем ограничения вызовов соответствующих функций только теми функциями, в которых появляются соответствующие прототипы.

```
// Эта программа записывает значения в массив, сортирует
// значения в порядке возрастания и печатает результирующий массив.
#include <iostream.h>
#include <iomanip.h>

void bubbleSort(int *, const int);

main()
{
 const int arraySize = 10;
 int a[arraySize] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

 cout << "Элементы данных в исходном порядке" << endl;

 for (int i = 0; i < arraySize; i++)
 cout << setw(4) << a[i];

 bubbleSort(a, arraySize); // сортировка массива
 cout << endl << "Элементы данных в возрастающем порядке"
 << endl;

 for (i = 0; i < arraySize; i++)
 cout << setw(4) << a[i];

 cout << endl;
 return 0;
}
```

Рис. 5.15. Пузырьковая сортировка вызовом по ссылке (часть1 из 2)

```

void bubbleSort(int *array, const int size)
{
 void swap(int *, int *);

 for (int pass = 1; pass < size; pass++)
 for (int j = 0; j < size - 1; j++)

 if (array[j] > array[j + 1])
 swap(&array[j], &array[j + 1]);
}

void swap(int *element1Ptr, int *element2Ptr)
{
 int temp = *element1Ptr;
 *element1Ptr = *element2Ptr;
 *element2Ptr = temp;
}

```

**Элементы данных в исходном порядке**

2 6 4 8 10 12 89 68 45 37

**Элементы данных в возрастающем порядке**

2 4 6 8 10 12 37 45 68 89

Рис. 5.15. Пузырьковая сортировка вызовом по ссылке (часть 2 из 2)

Прототип функции `swap` включен в тело функции `bubbleSort`, потому что `bubbleSort` является единственной функцией, которая вызывает `swap`. Включение прототипа в `bubbleSort` ограничивает соответствующим образом вызовы `swap` только теми, которые сделаны из `bubbleSort`. Другие функции, если бы они пытались вызвать `swap`, потерпели бы неудачу, поскольку они не имеют доступа к соответствующему прототипу функции. Это привело бы к ошибке компиляции, потому что C++ требует прототипов функций.

Заметим, что функция `bubbleSort` получает в качестве параметра размер массива. Чтобы сортировать массив, функция должна знать его размер. Когда массив передается функции, функция принимает адрес первого элемента массива. Размер массива должен передаваться функции отдельно.

Описывая функцию `bubbleSort` таким образом, что она получает в качестве параметра размер массива, мы получаем возможность использовать эту функцию в любой программе, которая сортирует целые одномерные массивы, причем массивы могут иметь любые размеры.

### **Замечание по технике программирования 5.5**

При передаче массива в функции передавайте также и размер массива (это предпочтительнее, чем встраивание в функцию знаний о размере массива). Такой подход позволяет создавать функции более общего вида. Функции общего вида часто могут повторно использоваться во многих программах.

Размер массива может быть запрограммирован заранее непосредственно внутри функции. Это ограничивает применимость функции только массивом данного размера и уменьшает возможность ее повторного использования. Такую функцию можно использовать только в программах, работающих с одномерными массивами лишь одного размера, закодированного внутри этой функции.

C++ снабжен унарной операцией *sizeof*, определяющей в байтах размер массива (или любого другого типа данных) во время компиляции программы. Будучи приложена к имени массива, как в программе на рис. 5.16, операция *sizeof* возвращает общее количество байтов в массиве как значение типа *size\_t*, которое обычно соответствует *unsigned int*. Заметим, что переменная типа *float* обычно занимает в памяти 4 байта, а объявленный массив *array* содержит 20 элементов. Поэтому *array* занимает в памяти 80 байтов.

```
//Операция sizeof, примененная к имени массива,
//возвращает количество байтов в массиве.
#include <iostream.h>

main()
{
 float array[20];

 cout <<"Количество байтов в массиве: "
 <<sizeof(array) << endl;
 return 0;
}
```

---

Количество байтов в массиве: 80

**Рис. 5.16.** Операция **sizeof**, примененная к имени массива, возвращает количество байтов в массиве

Количество элементов в массиве также может быть определено использованием результатов двух операций *sizeof*. Например, рассмотрим следующее объявление массива:

```
double realArray[22];
```

Переменные типа *double* обычно хранятся в 8 байтах памяти. Таким образом, массив *realArray* содержит в целом 176 байт. Для определения количества элементов в массиве можно использовать следующее выражение:

```
sizeof(realArray) / sizeof(double)
```

Выражение определяет количество байтов в массиве *realArray* и делит это значение на количество байтов, используемое для хранения значения *double*.

Программа на рис. 5.17 использует операцию *sizeof* для вычисления количества байтов, используемых для хранения каждого из стандартных типов данных в используемом персональном компьютере.

### Замечание по мобильности 5.2

Количество байтов, используемое для хранения отдельных типов данных, может быть различным для разных систем. При написании программ, которые зависят от размеров типа данных и которые будут выполняться на различных компьютерных системах, используйте *sizeof* для определения количества байтов, применяемых для хранения различных типов данных.

```
//Демонстрация операции sizeof
#include <iostream.h>

main()
{
 int array[20], *ptr = array;
 cout << " sizeof(char) = " << sizeof(char) << endl
 << " sizeof(short) = " << sizeof(short) << endl
 << " sizeof(int) = " << sizeof(int) << endl
 << " sizeof(long) = " << sizeof(long) << endl
 << " sizeof(float) = " << sizeof(float) << endl
 << " sizeof(double) = " << sizeof(double) << endl
 << " sizeof(long double) = " << sizeof(long double) << endl
 << " sizeof array = " << sizeof array << endl
 << " sizeof ptr = " << sizeof ptr << endl
 << endl;
 return 0;
}

sizeof(char) = 1
sizeof(short) = 2
sizeof(int) = 2
sizeof(long) = 4
sizeof(float) = 4
sizeof(double) = 8
sizeof(long double) = 10
sizeof array = 40
sizeof ptr = 2
```

**Рис. 5.17.** Использование операции `sizeof` для определения размеров стандартных типов данных

Операцию `sizeof` можно применять к любому имени переменной, имени типа или значению константы. При ее применении к имени переменной (которая не является именем массива) или к значению константы, возвращается количество байтов, используемое для хранения данного типа переменной или константы. Заметим, что скобки в операции `sizeof` требуются, если как operand используется имя типа, но не нужны, если как operand используется имя переменной.

#### Типичная ошибка программирования 5.5

Пропуск скобок в операции `sizeof`, когда operandом является имя типа, вызывает синтаксическую ошибку.

## **5.7. Выражения и арифметические действия с указателями**

Указатели могут применяться как operandы в арифметических выражениях, выражениях присваивания и выражениях сравнения. Однако, не все операции, обычно используемые в этих выражениях, разрешены применительно к переменным указателям. Этот раздел описывает, какие операции могут иметь в качестве operandов указатели и как эти операции используются.

С указателями может выполняться ограниченное количество арифметических операций. Указатель можно увеличивать (`++`), уменьшать (`--`), складывать с указателем целые числа (+ или `+=`), вычитать из него целые числа (- или `-=`) или вычитать один указатель из другого.

Допустим, что объявлен массив `int v[10]` и его первый элемент находится в памяти в ячейке 3000. Допустим, что указателю `vPtr` было присвоено начальное значение путем указания на `v[0]`, т.е. значение `vPtr` равно 3000. Рис. 5.18 схематически отображает эту ситуацию для машины с 4-байтовыми целыми. Заметим, что указателю `vPtr` можно было дать начальное значение указанием на массив `v` с помощью одного из следующих операторов

```
vPtr = v;
vPtr = &v[0];
```

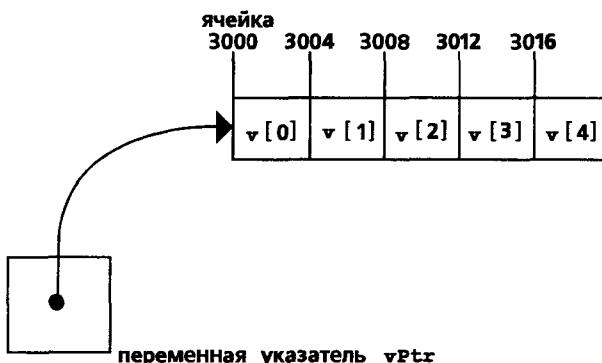


Рис. 5.18. Массив `v` и переменная указатель `vPtr`, указывающая на `v`

### Замечание по мобильности 5.3

Большинство современных компьютеров имеет 2-х или 4-х байтовые целые. Некоторые из более новых машин имеют 8-байтовые целые. Поскольку результат арифметических действий с указателями зависит от размера объектов, на которые указывает указатель, арифметические действия с указателями являются машинозависимыми.

В общепринятой арифметике сложение  $3000 + 2$  дает значение  $3002$ . Это нормально, но не в случае арифметических действий с указателями. Когда целое складывается или вычитается из указателя, указатель не просто увеличивается или уменьшается на это целое, но это целое предварительно умножается на размер объекта, на который ссылается указатель. Количество байтов зависит от типа данных. Например, оператор

```
vPtr += 2;
```

выработал бы значение  $3008$  ( $3000 + 2 * 4$ ) в предположении, что целое хранится в 4 байтах памяти. В массиве `v` указатель `vPtr` теперь указал бы на `v[2]` (рис. 5.19). Если целое хранится в 2 байтах памяти, то предыдущие вычисления имели бы результатом в памяти ячейку  $3004$  ( $3000 + 2 * 2$ ). Если бы массив имел другой тип данных, предыдущий оператор увеличивал бы указатель на количеством байтов, вдвое превышающее число байтов, необходимо

димых для хранения этого типа данных. В случае выполнения арифметических операций с указателями на массив символов, результат совпадает с обычной арифметикой, поскольку каждый символ занимает один байт.

Если указатель `vPtr` был увеличен до значения 3016, указывающего на `v[4]`, оператор

```
vPtr -= 4;
```

вернул бы `vPtr` обратно к значению 3000 — к началу массива. Если указатель увеличивается или уменьшается на 1, можно использовать операции инкремента (`++`) или декремента (`--`). Каждый из операторов

```
++vPtr;
vPtr ++;
```

увеличивает указатель так, что он указывает на следующий элемент массива. Каждый из операторов

```
--vPtr;
vPtr--;
```

уменьшает указатель так, что он указывает на предыдущий элемент массива.

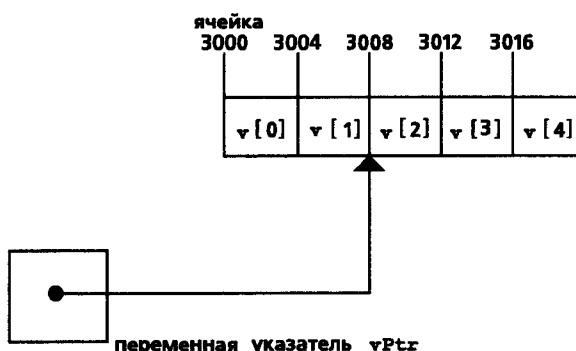


Рис. 5.19. Указатель `vPtr` после выполнения арифметических действий над указателями

Переменные указатели можно вычитать один из другого. Например, если `vPtr` содержит ячейку 3000, а `v2Ptr` содержит адрес 3008, оператор

```
x = v2Ptr - vPtr;
```

присвоит `x` значение разности номеров элементов массива, на которые указывают `vPtr` и `v2Ptr`, в данном случае 2. Арифметика указателей теряет смысл, если она выполняется не над массивами. Мы не можем предполагать, чтобы две переменные одинакового типа хранятся в памяти вплотную друг к другу, если они не соседствуют в массиве.

### Типичная ошибка программирования 5.6

Использование арифметических действий с указателями, не ссылающимися на элементы массива.

### Типичная ошибка программирования 5.7

Вычитание или сравнение двух указателей, которые не ссылаются на элементы одного и того же массива.

### Типичная ошибка программирования 5.8

Выход за пределы массива при использовании арифметических действий с указателями.

Указатель можно присваивать другому указателю, если оба указателя имеют одинаковый тип. В противном случае нужно использовать операцию приведения типа, чтобы преобразовать значение указателя в правой части присваивания к типу указателя в левой части присваивания. Исключением из этого правила является указатель на **void** (т.е. **void\***), который является общим указателем, способным представлять указатели любого типа. Указателю на **void** можно присваивать все типы указателей без приведения типа. Однако указатель на **void** не может быть присвоен непосредственно указателю другого типа — указатель на **void** сначала должен быть приведен к типу соответствующего указателя.

Указатель **void\*** не может быть разыменован. Например, компилятор знает, что указатель на **int** ссылается на четыре байта памяти на машине с 4-байтовыми целыми, но указатель на **void** просто содержит ячейку памяти для неизвестного типа данных — точное количество байтов, на которое ссылается указатель, неизвестно компилятору. Компилятор должен знать тип данных, чтобы определить количество байтов, которое должно быть разыменовано для определенного указателя. В случае указателя на **void** это количество байтов не может быть определено из типа.

### Типичная ошибка программирования 5.9

Присваивание указателя одного типа указателю другого типа (отличного от **void\***) вызывает синтаксическую ошибку.

### Типичная ошибка программирования 5.10

Разыменование указателя на **void\***.

Указатели можно сравнивать, используя операции проверки равенства и отношения, но такое сравнение бессмысленно, если указатели не указывают на элементы одного и того же массива. При сравнении указателей сравниваются адреса, хранимые в указателях. Сравнение двух указателей, указывающих на один и тот же массив, может показать, например, что один указатель указывает на элемент массива с более высоким номером, чем другой указатель. Типичным использованием сравнения указателя является определение, равен ли указатель 0.

## 5.8. Взаимосвязи между указателями и массивами

Массивы и указатели в C++ тесно связаны и могут быть использованы почти эквивалентно. Имя массива можно понимать как константный указатель. Указатели можно использовать для выполнения любой операции, включая индексирование массива.

### Хороший стиль программирования 5.5

Используйте нотацию массивов, а не нотацию указателей при манипуляции массивами. Несмотря на то, что программа может оказаться несколько длиннее, она, вероятно, будет более понятной.

Предположим, что объявлены массив целых чисел **b[5]** и целая переменная указатель **bPtr**. Поскольку имя массива (без индекса) является указателем на первый элемент массива, мы можем задать указателю **bPtr** адрес первого элемента массива **b** с помощью оператора

```
bPtr = b;
```

Это эквивалентно присваиванию адреса первого элемента массива следующим образом

```
bPtr = &b[0];
```

Сославшись на элемент массива **b[3]** можно с помощью выражения указателя

```
* (bPtr + 3)
```

В приведенном выражении 3 является *смещением* указателя. Когда указатель указывает на начало массива, смещение показывает, на какой элемент массива должна быть ссылка, так что значение смещения эквивалентно индексу массива. Предыдущую запись называют *записью указатель-смещение*. Скобки необходимы, потому что приоритет \* выше, чем приоритет +. Без скобок верхнее выражение прибавило бы 3 к значению выражения \*bPtr (т.е. 3 было бы прибавлено к **b[0]** в предположении, что **bPtr** указывает на начало массива). Поскольку элемент массива может быть указан указателем выражением, адрес

```
&b[3]
```

может быть записан также выражением указателем

```
bPtr + 3
```

Сам массив можно рассматривать как указатель и использовать в арифметике указателей. Например, выражение

```
* (b + 3)
```

тоже ссылается на элемент массива **b[3]**. Вообще все выражения с индексами массива могли бы быть записаны с помощью указателей и смещений. В этом случае запись указатель-смещение применялась бы к имени массива как к указателю. Заметим, что предыдущий оператор никоим образом не модифицирует имя массива; **b** продолжает указывать на первый элемент массива.

Указатели можно индексировать точно так же, как и массивы. Например, выражение

```
bPtr[1]
```

ссылается на элемент массива `b[1]`; это выражение рассматривается как запись **указатель-индекс**.

Напомним, что имя массива, по существу, является постоянным указателем; оно всегда указывает на начало массива. Таким образом, выражение

```
b+= 3
```

не разрешено, потому что оно пытается модифицировать значение имени массива с помощью арифметической операции над указателем.

### **Типичная ошибка программирования 5.11**

Хотя имена массивов являются указателями на их начало, а указатели в арифметических выражениях можно модифицировать, имена массивов в этих выражениях модифицировать нельзя.

В программе на рис. 5.20 использованы четыре обсужденных нами метода ссылок на элементы массива (индексирование массива, указатель-смещение с именем массива как указателем, индексирование указателя и указатель-смещение с указателем) для печати четырех элементов массива целых чисел `b`.

Чтобы продолжить иллюстрацию взаимозаменяемости массивов и указателей, рассмотрим две функции копирования строк `copy1` и `copy2` в программе на рис. 5.21. Обе функции копируют строку в массив символов. При сравнении прототипов функций для `copy1` и `copy2` функции выглядят идентично. Они соответствуют одной и той же задаче, но выполняются по разному.

Функция `copy1` использует для копирования строки `s2` в массив символов `s1` нотацию индексов массива. Функция объявляет целую переменную счетчика `i`, используемую как индекс массива. Заголовок структуры `for` полностью выполняет операцию копирования — ее тело является пустым оператором. Заголовок указывает, что `i` получает нулевое начальное значение и увеличивается на единицу в каждой итерации цикла. Условие `(s1[i] = s2[i]) != '\0'` в `for` выполняет операцию копирования символ за символом из `s2` в `s1`. Когда в `s2` встретится нулевой символ, он присваивается `s1`, и цикл завершается, потому что нулевой символ равен `'\0'`. Напомним, что значением оператора присваивания является значение, присвоенное левому аргументу.

Функция `copy2` использует для копирования строки `s2` в массив символов `s1` указатели и арифметику указателей. Опять же, заголовок структуры `for` полностью выполняет операцию копирования. Заголовок не содержит никаких операций задания начальных условий. Так же, как в функции `copy1`, условие `(*s1 = *s2) != '\0'` выполняет операцию копирования. Указатель `s2` разыменовывается и результатирующий символ присваивается разыменованному указателю `s1`. После присваивания в условии указатели увеличиваются, чтобы указать на следующий элемент массива `s1` и следующий символ строки `s2` соответственно. Когда `s2` указывает на нулевой символ, он присваивается разыменованному указателю `s1` и цикл заканчивается.

```

// Использование нотаций индексирования и указателей
// при работе с массивами

#include <iostream.h>

main()
{
 int b[] = {10, 20, 30, 40};
 int *bPtr = b; // задание bPtr как указателя
 // на массив b

 cout << "Массив b печатается с использованием:" << endl
 | << "Нотации индексов массива" << endl;

 for (int i = 0; i <= 3; i++)
 cout << "b[" << i << "] = " << b[i] << endl;

 cout << endl << "Записи указатель/смещение, в которой "
 << endl
 << "указатель является именем массива" << endl;

 for (int offset = 0; offset <= 3; offset++)
 cout << "*(" << b + offset << ") ="
 << *(b + offset) << endl;

 cout << endl << "Нотации индекса указателя" << endl;

 for (i = 0; i <= 3; i++)
 cout << "bPtr[" << i << "] = " << bPtr[i] << endl;

 cout << endl << "Нотации указатель/смещение" << endl;

 for (offset = 0; offset <= 3; offset++)
 cout << "*(" << bPtr + offset << ") ="
 << *(bPtr + offset) << endl;

 return 0;
}

```

**Рис. 5.20.** Использование четырех методов ссылки на элементы массива (часть 1 из 2)

Заметим, что первый аргумент и в `copy1`, и в `copy2` должен быть достаточно большим массивом, чтобы вмешать строку, содержащуюся во втором аргументе. В противном случае может произойти ошибка из-за попытки записи в ячейки памяти, выходящие за границы массива. Отметим также, что второй параметр каждой функции объявляется как `const char *` (константа строка). В обеих функциях второй аргумент копируется в первый — символы копируются из второго аргумента один за одним, но никогда не модифицируются. Поэтому второй параметр объявляется для указания на константное значение, чтобы реализовать принцип наименьших привилегий. Ни для одной из функций не требуется возможность модифицировать второй аргумент и поэтому ни одна из них не обеспечена этой возможностью.

**Массив b печатается с использованием:**

**Нотации индексов массива**

```
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40
```

**Записи указатель/смещение, в которой  
указатель является именем массива**

```
* (b + 0) = 10
* (b + 1) = 20
* (b + 2) = 30
* (b + 3) = 40
```

**Нотации индекса указателя**

```
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40
```

**Нотации указатель/смещение**

```
* (bPtr + 0) = 10
* (bPtr + 1) = 20
* (bPtr + 2) = 30
* (bPtr + 3) = 40
```

**Рис. 5.20.** Использование четырех методов ссылки на элементы массива (часть 2 из 2)

```
//Копирование строки с использованием нотации массивов
// и указателей

#include <iostream.h>

void copy1(char *, const char *);
void copy2(char *, const char *);

main()
{
 char string1[10], *string2 = "Привет",
 string3[10], string4[] = "До свидания";

 copy1(string1, string2);
 cout << "string1 = " << string1 << endl;
 .
 copy2(string3, string4);
 cout << "string3 = " << string3 << endl;

 return 0;
}
```

**Рис. 5.21.** Копирование строки с использованием нотации массивов и указателей (часть 1 из 2)

```

//Копирование s2 в s1 с использованием нотации массивов
void copy1(char *s1, const char *s2)
{
 for (int i = 0; (s1[i] = s2[i]) != '\0'; i++)
 ;
 //пустое тело
}
//Копирование s2 в s1 с использованием нотации указателей
void copy2(char *s1, const char *s2)
{
 for (; (*s1 = *s2) != '\0'; s1++, s2++)
 ;
 //пустое тело
}

string1 = Привет
string3 = До свидания

```

Рис. 5.21. Копирование строки с использованием нотации массивов и указателей (часть 2 из 2)

## 5.9. Массивы указателей

Массивы могут содержать указатели. Типичным использованием такой структуры данных является формирование *массива строк*. Каждый элемент такого массива — строка, но в C++ строка является, по существу, указателем на ее первый символ. Таким образом, каждый элемент в массиве строк в действительности является указателем на первый символ строки. Рассмотрим объявление массива строк *suit*, который может быть полезным для представления колоды карт.

```
char *suit[4] = {"Черви", "Бубны", "Трефы", "Пики"};
```

Элемент объявления *suit[4]* указывает массив из 4 элементов. Элемент объявления *char \** указывает, что тип каждого элемента массива *suit* — «указатель на *char*». Четыре значения, размещаемые в массиве — это «Черви», «Бубны», «Трефы» и «Пики». Каждое из них хранится в памяти как строка, завершающаяся нулевым символом, которая на один символ длиннее, чем число символов текста, указанного в кавычках. Эти четыре строки имеют длину 6, 6, 6 и 5 символов соответственно. Хотя это выглядит так, словно эти строки помещены в массив *suit*, на самом деле в массиве хранятся лишь указатели (Рис. 5.22). Каждый из них указывает на первый символ соответствующей ему строки. Таким образом, хотя размер массива *suit* фиксирован, он обеспечивает доступ к строкам символов любой длины. Эта гибкость — один из примеров мощных возможностей структурирования данных в C++.

Строки символов могли бы быть размещены в двумерном массиве, в котором каждая строка представляет одну масть, а каждый столбец представляет одну из букв имени масти. Такая структура данных должна иметь фиксированное количество столбцов на строку и это количество должно быть таким большим, как самая длинная строка. Поэтому затраты памяти на хранение большого количества строк, большинство из которых короче, чем самая длинная строка, будут значительными. Мы используем массивы строк для представления колоды карт в следующем разделе.

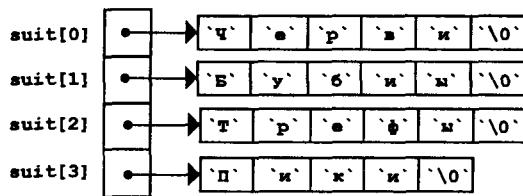


Рис. 5.22. Графическое представление массива suit

## 5.10. Учебный пример: моделирование тасования и раздачи карт

В этом разделе мы используем генерацию случайных чисел для разработки программы моделирования тасования и раздачи карт. Эта программа может затем быть использована для разработки программ, которые играют в различные карточные игры. Чтобы показать некоторые тонкости проблем эффективности, мы умышленно использовали неоптимальные алгоритмы тасования и раздачи. В упражнениях и далее в тексте мы разовьем более эффективные алгоритмы.

Используя нисходящую разработку с пошаговой детализацией, мы создадим программу, которая будет тасовать колоду из 52 игральных карт и затем раздавать их. Нисходящий подход особенно полезен при решении больших задач, более сложных, чем рассмотренные в предыдущих главах.

Для представления колоды игральных карт мы используем двумерный массив deck размером 4 на 13 (рис. 5.23). Строки соответствуют мастям — строка 0 соответствует червям, строка 1 — бубнам, строка 2 — трефам и строка 3 — пикам. Столбцы соответствуют значениям лицевых сторон (фигур) карт — столбцы от 0 до 9 соответствуют фигурам от туза до десятки соответственно, а столбцы от 10 до 12 соответствуют валету, даме и королю. Мы загрузим в массив строк suit строки символов, представляющие четыре масти, а в массив строк face символные строки, представляющие тринадцать значений фигур.

Представление колоды карт двумерным массивом deck[4][13].

|       |   | Туз | Двойка | Тройка | Четверка | Пятерка | Шестерка | Семерка | Восьмерка | Девятка | Десятка | Валет | Дама | Король |
|-------|---|-----|--------|--------|----------|---------|----------|---------|-----------|---------|---------|-------|------|--------|
|       |   | 0   | 1      | 2      | 3        | 4       | 5        | 6       | 7         | 8       | 9       | 10    | 11   | 12     |
| Черви | 0 |     |        |        |          |         |          |         |           |         |         |       |      |        |
| Бубны | 1 |     |        |        |          |         |          |         |           |         |         |       |      |        |
| Трефы | 2 |     |        |        |          |         |          |         |           |         |         |       |      |        |
| Пики  | 3 |     |        |        |          |         |          |         |           |         |         |       |      |        |

deck[2][12] представляет короля треф

Трефы      Король

Рис. 5.23. Представление колоды карт двумерным массивом

Эта смоделированная колода карт может тасоваться следующим образом. Сначала массив `deck` обнуляется. Затем строка `row` 0–3 и столбец `column` 0–12 выбираются как случайные числа. В элемент массива `deck[row][column]` вставляется число 1, которое указывает, что эта карта будет сдана из тасуемой колоды первой. Этот процесс продолжается с числами 2, 3, ..., 52, которые вставляются в массив `deck` и показывают, какие карты займут второе, третье, ..., пятьдесят второе место в тасуемой колоде. Во время заполнения массива `deck` номерами карт, может оказаться, что одна и та же карта должна попасть в него дважды, т.е. после случайного выбора карты `deck[row][column]` окажется ненулевым. Такой выбор просто игнорируется и повторно случайно выбираются другие строки и столбцы до тех пор, пока не будет найдена не выбиравшаяся ранее карта. В конечном счете числа с 1 по 52 займут 52 позиции массива `deck`. В этот момент колода карт полностью перетасована.

Этот алгоритм тасования может выполняться неопределенno долго, если карты, которые уже растасованы, случайно выбираются повторно. Это явление известно как *неопределенная отсрочка*. В упражнениях обсуждается лучший тасующий алгоритм, который исключает возможность неопределенной отсрочки.

### Совет по повышению эффективности 5.2

Иногда в алгоритме, который кажется «естественным», могут таиться такие тонкие проблемы эффективности, как неопределенная отсрочка. Ищите алгоритмы, которые лишены проблемы неопределенной отсрочки.

Чтобы сдать первую карту, мы отыскиваем в массиве `deck[row][column] = 1`. Это соответствует вложенной структуре `for`, которая варьирует `row` от 0 до 3 и `column` от 0 до 12. Какая карта соответствует этой позиции массива? В массив `suit` предварительно были загружены четыре масти, так что для получения масти мы печатаем символьную строку `suit[row]`. Аналогично, чтобы получить значение фигуры на карте, мы печатаем символьную строку `face[column]`. Мы печатаем также символьную строку «масти». Печать этой информации в нужной последовательности дает нам возможность напечатать каждую карту в виде «король масти трефы», «туз масти бубны» и т.д.

Приступим к процессу нисходящей пошаговой детализации. Вершина очевидна:

*Тасовать и раздать 52 карты*

Наша первая детализация дает:

*Задать начальные условия массиву suit (масть)*

*Задать начальные условия массиву face (фигура)*

*Задать начальные условия массиву deck (колода)*

*Тасовать колоду*

*Раздать 52 карты*

Предложение «Тасовать колоду» может быть расширено следующим образом:

*Цикл для каждой из 52 карт*

*Поместить номер карты в случайно выбранную незанятую позицию колоды*

Предложение «Раздать 52 карты» может быть расширено следующим образом:

*Цикл для каждой из 52 карт*

*Найти номер карты в массиве deck и напечатать фигуру и масть карты*

Объединение этих расширений дает нам полную вторую детализацию:

*Задать начальные условия массиву suit (масть)*

*Задать начальные условия массиву face (фигура)*

*Задать начальные условия массиву deck (колода)*

*ЦИКЛ для каждой из 52 карт*

*Поместить номер карты в случайно выбранную незанятую позицию колоды*

*ЦИКЛ для каждой из 52 карт*

*Найти номер карты в массиве deck и напечатать фигуру и масть карты*

Предложение «Поместить номер карты в случайно выбранную незанятую позицию колоды» может быть развернуто следующим образом:

*Выбрать случайным образом позицию в колоде*

*ПОКА выбранная позиция в колоде оказывается уже выбранной раньше*

*Выбрать случайным образом позицию в колоде*

*Поместить номер карты в выбранную позицию колоды*

Предложение «Найти номер карты в массиве deck и напечатать фигуру и масть карты» может быть развернуто следующим образом:

*ЦИКЛ для каждой позиции массива deck*

*ЕСЛИ значение позиции равно номеру карты*

*Напечатать фигуру и масть карты*

Объединение этих расширений дает нам третью детализацию:

*Задать начальные условия массиву suit (масть)*

*Задать начальные условия массиву face (фигура)*

*Задать начальные условия массиву deck (колода)*

*ЦИКЛ для каждой из 52 карт*

*Выбрать случайным образом позицию в колоде*

*ПОКА выбранная позиция в колоде оказывается уже выбранной раньше*

*Выбрать случайным образом позицию в колоде*

*Поместить номер карты в выбранную позицию колоды*

*ЦИКЛ для каждой из 52 карт*

*ЦИКЛ для каждой позиции массива deck*

*ЕСЛИ значение позиции равно номеру карты*

*Напечатать фигуру и масть карты*

Этим процесс детализаций завершается. Заметим, что программа станет более эффективной, если части алгоритма «тасование» и «раздача» скомбинировать так, чтобы каждая карта сдавалась, как только она помещается в

колоду. Мы выбрали раздельное программирование этих операций, потому что обычно карты раздаются после того, как они растасованы (а не во время тасования).

Программа тасования и раздачи карт представлена на рис. 5.24, а пример ее выполнения — на рис. 5.25. Заметим, что в функции `deal` использовано форматирование выходных данных:

```

cout << setw(9) << setiosflags(ios::right)
<< wFace[column] << " масти "
<< setw(5) << setiosflags(ios::left)
<< wSuit[row] << (card % 2 == 0 ? '\n' : '\t');

// Программа раздачи карт (часть1).
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

void shuffle(int [] [13]);
void deal(const int [] [13], const char *[], const char *[]);

main ()
{
 char *suit[4] = {"Черви", "Бубны", "Трефы", "Пики" };
 char *face[13] = {"Туз", "Двойка", "Тройка", "Четверка",
 "Пятерка", "Шестерка", "Семерка",
 "Восьмерка", "Девятка", "Десятка",
 "Валет", "Дама", "Король"};
 int deck[4][13] = {0};

 srand(time(NULL));

 shuffle(deck);
 deal(deck, face, suit);

 return 0;
}

void shuffle(int wDeck[] [13])
{
 int row, column;

 for (int card = 1; card <= 52; card++) {
 row = rand() % 4;
 column = rand() % 13;

 while(wDeck[row][column] != 0) {
 row = rand() % 4;
 column = rand() % 13;
 }

 wDeck[row][column] = card;
 }
}

```

**Рис. 5.24.** Программа раздачи карт (часть1 из 2)

Предыдущий оператор вывода выводит название фигуры в поле размером 9 символов с выравниванием вправо, а название масти — в поле размером 5 символов с выравниванием влево. Выходные данные печатаются в две колонки. Если карта выводится в первой колонке, выводится символ табуляции и перемещает вывод во вторую колонку, в противном случае выводится символ новой строки.

```
void deal(const int wDeck[] [13], const char *wFace[],
 const char *wSuit[])
{
 for (int card = 1; card <= 52; card++)
 for (int row = 0; row <= 3; row++)
 for (int column = 0; column <= 12; column++)
 if (wDeck[row] [column] == card)
 cout << setw(9) << setiosflags(ios::right)
 << wFace[column] << " масти "
 << setw(5) << setiosflags(ios::left)
 << wSuit[row]
 << (card % 2 == 0 ? '\n' : '\t');
}
```

Рис. 5.24. Программа раздачи карт (часть 2 из 2)

|           |             |           |             |
|-----------|-------------|-----------|-------------|
| Валет     | масти Бубны | Двойка    | масти Трефы |
| Шестерка  | масти Трефы | Семерка   | масти Трефы |
| Король    | масти Черви | Семерка   | масти Бубны |
| Десятка   | масти Пики  | Девятка   | масти Пики  |
| Дама      | масти Бубны | Четверка  | масти Бубны |
| Четверка  | масти Пики  | Дама      | масти Трефы |
| Туз       | масти Пики  | Двойка    | масти Черви |
| Девятка   | масти Бубны | Валет     | масти Пики  |
| Валет     | масти Трефы | Шестерка  | масти Пики  |
| Восьмерка | масти Черви | Дама      | масти Пики  |
| Тройка    | масти Черви | Дама      | масти Черви |
| Четверка  | масти Трефы | Восьмерка | масти Пики  |
| Король    | масти Трефы | Десятка   | масти Трефы |
| Тройка    | масти Трефы | Девятка   | масти Трефы |
| Туз       | масти Черви | Пятерка   | масти Бубны |
| Валет     | масти Черви | Восьмерка | масти Бубны |
| Четверка  | масти Черви | Семерка   | масти Черви |
| Тройка    | масти Бубны | Король    | масти Бубны |
| Двойка    | масти Бубны | Десятка   | масти Черви |
| Туз       | масти Бубны | Король    | масти Пики  |
| Двойка    | масти Пики  | Пятерка   | масти Черви |
| Туз       | масти Трефы | Восьмерка | масти Трефы |
| Десятка   | масти Бубны | Пятерка   | масти Пики  |
| Девятка   | масти Черви | Пятерка   | масти Трефы |
| Шестерка  | масти Черви | Тройка    | масти Пики  |
| Семерка   | масти Пики  | Шестерка  | масти Бубны |

Рис. 5.25. Пример работы программы раздачи карт

В алгоритме раздачи имеется слабое место. Когда соответствующая карта найдена, даже если она найдена с первой попытки, две внутренние структуры `for` продолжают просмотр оставшихся элементов `deck`. В упражнениях и в учебном примере в главе 16 мы исправим этот недостаток.

## 5.11. Указатели на функции

Указатель на функцию содержит адрес функции в памяти. В главе 4 мы видели, что имя массива на самом деле является адресом первого элемента массива. Аналогично, имя функции на самом деле — начальный адрес ее кода. Указатели на функции можно передавать функциям, возвращать из функций, хранить в массивах и присваивать другим указателям на функции.

Чтобы проиллюстрировать использование указателей на функции, мы модифицировали программу пузырьковой сортировки на рис. 5.15 и сформировали программу, приведенную на рис. 5.26.

```
// Программа многоцелевой сортировки, использующая указатели на функции
#include <iostream.h>
#include <iomanip.h>

void bubble(int *, const int, int (*) (int, int));
int ascending (const int, const int);
int descending(const int, const int);

main()
{
 const int arraySize = 10;
 int order, a[arraySize] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

 cout << "Введите 1 для сортировки в возрастающем порядке," << endl
 << "Введите 2 для сортировки в убывающем порядке: ";
 cin >> order;
 cout << endl << "Элементы данных в исходном порядке" << endl;

 for (int counter = 0; counter < arraySize; counter++)
 cout << setw(4) << a[counter];

 if (order == 1) {
 bubble(a, arraySize, ascending);
 cout << endl << "Элементы данных в возрастающем порядке"
 << endl;
 }
 else {
 bubble(a, arraySize, descending);
 cout << endl << "Элементы данных в убывающем порядке"
 << endl;
 }
 for (counter = 0; counter < arraySize; counter++)
 cout << setw(4) << a[counter];

 cout << endl;
 return 0;
}
```

Рис. 5.26. Программа многоцелевой сортировки, использующая указатели на функции (часть 1 из 2)

```
void bubble(int *work, const int size, int (*compare)(int, int))
{
 void swap(int *, int *);

 for (int pass = 1; pass < size; pass++)

 for (int count = 0; count < size - 1; count++)

 if ((*compare)(work[count], work[count + 1]))
 swap(&work[count], &work[count + 1]);
}

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. 5.26. Программа многоцелевой сортировки, использующая указатели на функции (часть 2 из 2)

Наша новая программа содержит `main` и функции `bubble`, `swap`, `ascending` и `descending`. Функция `bubbleSorting` принимает указатель на функцию — либо функцию `ascending`, либо функцию `descending` — как аргумент и, кроме того, аргументы вида массива целых чисел и размера этого массива. Программа предлагает пользователю выбрать, в каком порядке должен быть отсортирован массив — возрастающем или убывающем. Если пользователь вводит 1, то в функцию `bubble` передается указатель на функцию `ascending`, приводящую к сортировке массива в возрастающем порядке. Если пользователь вводит 2, то в функцию `bubble` передается указатель на функцию `descending`, приводящую к сортировке массива в возрастающем порядке. Выходные данные программы представлены на рис. 5.27.

В заголовке функции `bubble` появляется следующий параметр:

```
int (*compare) (int, int)
```

Это сообщает `bubble` о том, что она должна ждать параметр, являющийся указателем на функцию, которая принимает два целых параметра и возвращает целый результат. Скобки вокруг `*compare` нужны потому, что `*` имеет приоритет ниже, чем скобки, в которые заключены параметры функции.

```

Ведите 1 для сортировки в возрастающем порядке,
Ведите 2 для сортировки в убывающем порядке: 1
Элементы данных в исходном порядке
 2 6 4 8 10 12 89 68 45 37
Элементы данных в возрастающем порядке
 2 4 6 8 10 12 37 45 68 89

Ведите 1 для сортировки в возрастающем порядке,
Ведите 2 для сортировки в убывающем порядке: 2
Элементы данных в исходном порядке
 2 6 4 8 10 12 89 68 45 37
Элементы данных в убывающем порядке
 89 68 45 37 12 10 8 6 4 2

```

**Рис. 5.27.** Выходные данные программы пузырьковой сортировки на рис. 5.26

Если бы мы не включили скобки, объявление имело бы вид

```
int *compare(int, int)
```

что объявляло бы функцию, которая принимает два целых как параметры и возвращает указатель на целое.

Соответствующий параметр в прототипе функции **bubble** имеет вид

```
int (*) (int, int)
```

Заметим, что в прототип включены только типы, но для целей документирования можно включить и имена, которые компилятор будет игнорировать.

Функция, переданная **bubble**, вызывается в операторе **if** следующим образом

```
if ((*compare) (work[count], work[count+1]))
```

Так же, как указатель на переменную разыменовывается для доступа к значению переменной, указатель на функцию разыменовывается, чтобы использовать функцию.

, Вызов функции можно было бы выполнить и без разыменования указателя, как в выражении:

```
if ((compare) (work[count], work[count+1]))
```

которое использует указатель непосредственно как имя функции. Мы предполагаем первый метод вызова функции посредством указателя, потому что он явно показывает, что **compare** является указателем на функцию, который разыменовывается, чтобы вызвать функцию. Второй метод вызова функции посредством указателя представляет это так, будто **compare** является подлинной функцией. Это может смутить пользователя программы, который захотел бы увидеть определение функции **compare** и найти в файле то, что в нем не определено.

Типичным применением указателей на функцию являются так называемые системы, управляемые меню. Пользователю предлагается выбрать позицию меню (например, от 1 до 5). Каждая позиция обслуживается своей определенной функцией. Указатели на каждую функцию хранятся в массиве указателей на функции. Выбор пользователя используется как индекс массива, а указатель в массиве используется для вызова функции.

Программа на рис. 5.28 представляет обобщающий пример механизма объявления и использования массива указателей на функции. Определены три функции — `function1`, `function2` и `function3` — каждая из которых принимает целый аргумент и ничего не возвращает. Указатели на эти три функции хранятся в массиве `f`, который объявлен следующим образом

```
void (*f[3])(int) = {function1, function2, function3};

//Демонстрация массива указателей на функции.
#include <iostream.h>

void function1(int);
void function2(int);
void function3(int);

main()
{
 void (*f[3])(int) = {function1, function2, function3};
 int choice;

 cout << "Введите число между 0 и 2, 3 - окончание: ";
 cin >> choice;

 while (choice >= 0 && choice < 3) {
 (*f[choice])(choice);
 cout << "Введите число между 0 и 2, 3 - окончание: ";
 cin >> choice;
 }
 cout << "Вы ввели 3 для окончания" << endl;
 return 0;
}
void function1(int a)
{
 cout << "Вы ввели " << a << ", поэтому была вызвана функция 1"
 << endl << endl;
}

void function2(int b)
{
 cout << "Вы ввели " << b << ", поэтому была вызвана функция 2"
 << endl << endl;
}

void function3(int c)
{
 cout << "Вы ввели " << c << ", поэтому была вызвана функция 3"
 << endl << endl;
}
```

Рис. 5.28. Демонстрация массива указателей на функции (часть 1 из 2)

Ведите число между 0 и 2, 3 - окончание: 0  
Вы ввели 0, поэтому была вызвана функция 1

Ведите число между 0 и 2, 3 - окончание: 1  
Вы ввели 1, поэтому была вызвана функция 2

Ведите число между 0 и 2, 3 - окончание: 2  
Вы ввели 2, поэтому была вызвана функция 3

Ведите число между 0 и 2, 3 - окончание: 3  
Вы ввели 3 для окончания

**Рис. 5.28.** Демонстрация массива указателей на функции (часть 2 из 2)

Это объявление читается так: «`f` является массивом трех указателей на функции, которые берут аргумент `int` и возвращают `void`». Массив получает в качестве начальных значений имена трех функций. Когда пользователь вводит значения между 0 и 2, это значение используется в качестве индекса в массиве указателей на функции. Вызов функции выполняется следующим образом,

```
(*f[choice])(choice);
```

В этом вызове `f[choice]` выделяет указатель, расположенный в элементе массива с индексом `choice`. Указатель разыменовывается, чтобы вызвать функцию, и `choice` передается функции как аргумент. Каждая функция печатает значения своих аргументов и имя функции, чтобы показать, что функция вызывается правильно. Мы будем разрабатывать далее системы, управляемые меню, в упражнениях.

## 5.12. Введение в обработку символов и строк

В этом разделе мы познакомимся с некоторыми типичными функциями стандартной библиотеки С, которые обеспечивают обработку строк. Обсуждаемая здесь техника подходит для разработки текстовых редакторов, лингвистических процессоров, программ верстки, систем компьютеризированного набора и других программных систем обработки текстов.

### 5.12.1. Основы теории символов и строк

Символы — это фундаментальные стандартные блоки исходных программ на С++. Каждая программа составлена из последовательностей символов, которые, если их объединение в группу имеет смысл, интерпретируются компьютером как последовательности инструкций, используемых для выполнения задачи. Программа может содержать *символьные константы*. Символьная константа — это целое значение, представленное как символ в одинарных кавычках. Значение символьной константы — это целочисленное значение в наборе машинных символов. Например, 'z' представляет собой целое значение z, а '\n' представляет собой целое значение символа перехода на новую строку.

Строка — это последовательность символов, обрабатываемая как единый модуль. Страна может включать буквы, цифры и разнообразные специальные

символы, такие как +, -, \*, /, \$ и другие. *Литеральные константы* или *строковые константы* записываются в C++ в двойных кавычках следующим образом:

|                          |                  |
|--------------------------|------------------|
| "John Q. Doe"            | (имя)            |
| "9999 Main Street"       | (адрес улицы)    |
| "Waltham, Massachusetts" | (город и штат)   |
| "(201) 555-1212"         | (номер телефона) |

Строка в C++ — это массив символов, оканчивающийся *нулевым символом* ('\0'). Страна доступна через указатель на первый символ в строке. Значением строки является адрес ее первого символа. Таким образом можно сказать, что в C++ строка является *указателем* — указателем на первый символ строки. В этом смысле строки подобны массивам, потому что массив тоже является указателем на свой первый элемент.

Строка может быть объявлена либо как массив символов, либо как переменная типа **char\***. Каждое из объявлений

```
char color[] = "синий";
char *colorPtr = "синий";
```

присваивает переменной строке начальное значение «синий». Первое объявление создает массив из 6 элементов **color** содержащий символы 'с', 'и', 'н', 'и', 'й' и '\0'. Второе объявление создает переменную указатель **colorPtr**, который указывает на строку «синий» где-то в памяти.

### Замечание по мобильности 5.4

Когда переменная типа **char\*** получает в качестве начального значения строковую константу, некоторые компиляторы могут поместить строку в такое место в памяти, где строка не может быть модифицирована. Если у вас может возникнуть необходимость модифицировать строковую константу, она должна храниться в массиве символов, чтобы обеспечить возможность ее модификации во всех системах.

- Объявление **char color[] = {"синий"}** может быть записано  
`char color[] = {'с', 'и', 'н', 'и', 'й', '\0'};`

Когда объявляется массив символов, содержащий строку, он должен быть достаточно большим, чтобы хранить строку и ее завершающий нулевой символ. Предыдущее объявление определяет размер массива автоматически, основываясь на количестве начальных значений в списке.

### Типичная ошибка программирования 5.12

Не выделяется достаточно места в массиве символов для хранения нулевого символа, завершающего строку.

### Типичная ошибка программирования 5.13

Создание или использование «строки», которая не содержит завершающего нулевого символа.

## Хороший стиль программирования 5.6

При хранении строки в массиве символов убедитесь, что массив достаточно велик, чтобы вместить наибольшую строку, которую потребуется хранить. В C++ допускается хранить строки любой длины. Если строка больше символьного массива, в котором она должна храниться, символы, выходящие за конец массива, будут изменять данные в разделах памяти, следующих за массивом.

Строку можно присвоить массиву, используя операцию `cin` — взять из потока. Например, строку можно присвоить символьному массиву `word[20]` следующим оператором:

```
cin >> word;
```

Вводимая пользователем строка хранится в `word`. Предыдущий оператор считывает символы до тех пор, пока не встретится пробел, символ табуляции, символ новой строки или указатель конца файла. Заметим, что строка не должна быть длиннее 19 символов, чтобы оставить место для завершающего нулевого символа. Манипулятор потока `setw`, с которым мы познакомились в главе 2, можно использовать для гарантии того, что строка, считанная в `word`, не превысит размер массива. Например, оператор

```
cin >> setw(20) >> word;
```

указывает, что `cin` должен считать максимум 19 символов в массив `word` и оставить 20-й символ в массиве для хранения завершающего нулевого символа. Манипулятор потока `setw` применяется только к следующему вводимому значению символа.

В некоторых случаях желательно вводить в массив полную строку текста. С этой целью C++ снабжен функцией `cin.getline`. Функция `cin.getline` требует три аргумента — массив символов, в котором должна храниться строка текста, длина и символ-ограничитель. Например, фрагмент программы

```
char sentence[80];
cin.getline(sentence, 80, '\n');
```

объявляет массив `sentence` из 80 символов, затем считывает строку текста с клавиатуры в этот массив. Функция прекращает считывание символов в случаях, если встречается символ-ограничитель '`\n`', если вводится указатель конца файла или если количество считанных символов оказывается на один меньше, чем указано во втором аргументе (последний символ в массиве резервируется для завершающего нулевого символа). Если встречается символ-ограничитель, он считывается и отбрасывается. Третий аргумент `cin.getline` имеет '`\n`' в качестве значения по умолчанию, так что предыдущий вызов функции мог бы быть написан в следующем виде:

```
cin.getline(sentence, 80);
```

В главе 11, «Потоки ввода-вывода», детально обсуждается функция `cin.getline` и другие функции ввода-вывода.

## Типичная ошибка программирования 5.14

Обработка одного символа как строки. Стока является указателем — это может быть достаточно большое целое. А символ — это небольшое целое (диапазон значений ASCII 0–255). Во многих системах это вызывает ошибку, потому что нижние адреса памяти зарезервированы для специальных целей, таких как обработчики прерываний операционной системы — так что происходит «несанкционированный доступ».

### **Типичная ошибка программирования 5.15**

Передача символа в качестве аргумента функции, которая ожидает строку.

### **Типичная ошибка программирования 5.16**

Передача строки в качестве аргумента функции, которая ожидает символ.

## **5.12.2. Функции работы со строками из библиотеки обработки строк**

Библиотека обработки строк обеспечивает много полезных функций для работы со строковыми данными, сравнения строк, поиска в строках символов и других подстрок, разметки строк (разделения строк на логические куски) и определения длины строк. В этом разделе представлены некоторые типовые функции работы со строками из библиотеки обработки строк (из стандартной библиотеки С). Сведения об этих функциях сведены в таблицу на рис. 5.29.

Заметим, что некоторые функции на рис. 5.29 имеют параметры с типом данных `size_t`. Этот тип определяется в заголовочном файле `stddef.h` (из стандартной библиотеки С) как беззнаковый целый тип, такой, как `unsigned int` или `unsigned long`.

### **Типичная ошибка программирования 5.17**

Забывают включить заголовочный файл `<string.h>` при использовании функций из библиотеки обработки строк.

Функция `strcpy` копирует свой второй аргумент — строку в свой первый аргумент — массив символов, который должен быть достаточно большим, чтобы хранить строку и ее завершающий нулевой символ, который также копируется. Функция `strncpy` эквивалентна `strcpy` за исключением того, что `strncpy` указывает количество символов, которое должно быть скопировано из строки в массив. Заметим, что функция `strncpy` не обязательно должна копировать завершающий нулевой символ своего второго аргумента; завершающий нулевой символ записывается только в том случае, если количество символов, которое должно быть скопировано, по крайней мере на один больше, чем длина строки. Например, если второй аргумент — "test", завершающий нулевой символ записывается только в случае, если третий аргумент `strncpy` по меньшей мере равен 5 (четыре символа в "test" плюс один завершающий нулевой символ). Если третий аргумент больше пяти, завершающий нулевой символ добавляется к массиву до тех пор, пока не будет записано общее количество символов, указанное третьим аргументом.

### **Типичная ошибка программирования 5.18**

Не добавляется завершающий нулевой символ к первому аргументу `strncpy`, когда третий аргумент меньше или равен длине строки во втором аргументе.

| Прототип функции                                      | Описание функции                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| char *strcpy(char *s1, const char *s2)                | Копирует строку <b>s2</b> в массив символов <b>s1</b> . Возвращает значение <b>s1</b> .                                                                                                                                                                                                                                                                                                                                                                                                                               |
| char *strncpy(char *s1, const char *s2, size_t n)     | Копирует не более <b>n</b> символов из строки <b>s2</b> в массив символов <b>s1</b> . Возвращает значение <b>s1</b> .                                                                                                                                                                                                                                                                                                                                                                                                 |
| char *strcat(char *s1, const char *s2)                | Добавляет строку <b>s2</b> к строке <b>s1</b> . Первый символ строки <b>s2</b> записывается поверх завершающего нулевого символа строки <b>s1</b> . Возвращает значение <b>s1</b> .                                                                                                                                                                                                                                                                                                                                   |
| char *strncat(char *s1, const char *s2, size_t n)     | Добавляет не более <b>n</b> символов строки <b>s2</b> в строку <b>s1</b> . Первый символ из <b>s2</b> записывается поверх завершающего нулевого символа в <b>s1</b> . Возвращает значение <b>s1</b> .                                                                                                                                                                                                                                                                                                                 |
| int strcmp(const char *s1, const char *s2)            | Сравнивает строки <b>s1</b> и <b>s2</b> . Функция возвращает значение 0, меньшее, чем 0 или большее, чем 0, если <b>s1</b> соответственно равна, меньше или больше, чем <b>s2</b> .                                                                                                                                                                                                                                                                                                                                   |
| int strncmp(const char *s1, const char *s2, size_t n) | Сравнивает до <b>n</b> символов строки <b>s1</b> со строкой <b>s2</b> . Функция возвращает значение 0, меньшее, чем 0 или большее, чем 0, если <b>s1</b> соответственно равна, меньше или больше, чем <b>s2</b> .                                                                                                                                                                                                                                                                                                     |
| char *strtok(char *s1, const char *s2)                | Последовательность вызовов <b>strtok</b> разбивает строку <b>s1</b> на «лексемы» — логические куски, такие, как слова в строке текста — разделенные символами, содержащимися в строке <b>s2</b> . Первый вызов содержит в качестве первого аргумента <b>s1</b> , а последующие вызовы для продолжения обработки той же строки, содержат в качестве первого аргумента <b>NULL</b> . При каждом вызове возвращается указатель на текущую лексему. Если при вызове функции лексем больше нет, возвращается <b>NULL</b> . |
| size_t strlen(const char *s)                          | Определяет длину строки <b>s</b> . Возвращает количество символов, предшествующих завершающему нулевому символу.                                                                                                                                                                                                                                                                                                                                                                                                      |

Рис. 5.29. Функции работы со строками из библиотеки обработки строк

В программе на рис. 5.30 **strcpy** используется для копирования полной строки в массиве **x** в массив **y** и **strncpy** — для копирования первых 14 символов массива **x** в массив **z**. Нулевой символ ('\0') добавляется в массив **z**, потому что вызов **strncpy** в программе не записывает завершающий нулевой символ (третий аргумент меньше, чем длина строки второго аргумента).

Функция **strcat** добавляет свой второй аргумент — строку к своему первому аргументу — массиву символов, содержащему строку. Первый символ второго аргумента замещает нулевой символ ('\0'), который завершал строку в первом аргументе. Программист должен быть уверен, что массив, используемый для хранения первой строки, достаточно велик для того, чтобы хранить комбинацию первой строки, второй строки и завершающего нулевого символа (скопированного из второй строки). Функция **strncat** добавляет ука-

занное количество символов из второй строки в первую. К результату добавляется завершающий нулевой символ. Программа на рис. 5.31 демонстрирует функцию **strcat** и функцию **strncat**.

```
//Использование strcpy и strncpy
#include <iostream.h>
#include <string.h>

main()
{
 char x[] = "Поздравляю Вас с днем рождения";
 char y[35], z[15];

 cout << "Строка в массиве x: " << x << endl
 << "Строка в массиве y: " << strcpy(x, y) << endl;
 strncpy(z, x, 14);
 z[14] = '\0';
 cout << "Строка в массиве z: " << z << endl;
 return 0;
}
```

---

```
Строка в массиве x: Поздравляю Вас с днем рождения
Строка в массиве y: Поздравляю Вас с днем рождения
Строка в массиве z: Поздравляю Вас
```

**Рис. 5.30.** Использование **strcpy** и **strncpy**

```
// Использование strcat и strncat.
#include <iostream.h>
#include <string.h>

main()
{
 char s1[25] = "Счастливого ";
 char s2[] = "Нового Года ";
 char s3[40] = "";

 cout << "s1 = " << s1 << endl << "s2 = " << s2 << endl;
 cout << "strcat(s1, s2) = " << strcat(s1, s2) << endl;
 cout << "strncat(s3, s1, 12) = " << strncat(s3, s1, 12)
 << endl;
 cout << "strcat(s3, s1) = " << strcat(s3, s1) << endl;

 return 0;
}
```

---

```
s1 = Счастливого
s2 = Нового Года
strcat(s1, s2) = Счастливого Нового Года
strncat(s3, s1, 12) = Счастливого
strcat(s3, s1) = Счастливого Счастливого Нового Года
```

**Рис. 5.31.** Использование **strcat** и **strncat**

Программа на рис. 5.32 сравнивает три строки, используя функции **strcmp** и **strncmp**. Функция **strcmp** сравнивает символ за символом строку в своем первом аргументе со строкой в своем втором аргументом. Функция возвращает 0, если строки равны, отрицательное значение, если первая строка меньше, чем вторая, и положительное значение, если первая строка больше, чем вторая. Функция **strncmp** эквивалентна **strcmp**, за исключением того, что **strncmp** проводит сравнение только до указанного количества символов. Функция **strncmp** не сравнивает символы, следующие за нулевым символом в строке. Программа печатает целое значение, возвращаемое при каждом вызове функции.

### Типичная ошибка программирования 5.19

Предположение, что **strcmp** и **strncmp** возвращают 1, если их аргументы равны. При равенстве обе функции возвращают 0 (значение «ложь» в C++). Поэтому при проверке двух строк на равенство результаты функции **strcmp** или **strncmp** должны для определения равенства строк сравниваться с 0.

Чтобы понять, что означает, что одна строка «больше» или «меньше», чем другая строка, рассмотрим процесс расстановки имен по алфавиту. Читатель, без сомнения, поставил бы «Jones» перед «Smith», потому что в алфавите первая буква имени «Jones» стоит раньше первой буквы имени «Smith». Но алфавит — это больше, чем просто список из 26 букв — он упорядочивает список символов. Каждая буква занимает внутри списка определенную позицию. «Z» — это больше, чем просто буква алфавита; «Z» — это двадцать шестая буква алфавита.

```
// Использование strcmp и strncmp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

main()
{
 char *s1 = "Счастливого Нового Года";
 char *s2 = "Счастливого Нового Года";
 char *s3 = "Счастливого Праздника";

 cout << "s1 = " << s1 << endl << "s2 = " << s2 << endl
 << "s3 = " << s3 << endl << endl << "strcmp(s1, s2) = "
 << setw(2) << strcmp(s1, s2) << endl << "strcmp(s1, s3) = "
 << setw(2) << strcmp(s1, s3) << endl << "strcmp(s3, s1) = "
 << setw(2) << strcmp(s3, s1) << endl << endl;

 cout << "strncmp(s1, s3, 12) = " << setw(2)
 << strncmp(s1, s3, 12) << endl
 << "strcmp(s1, s3, 13) = " << setw(2)
 << strcmp(s1, s3, 13) << endl
 << "strcmp(s3, s1, 13) = " << setw(2)
 << strcmp(s3, s1, 13) << endl;
 return 0;
}
```

Рис. 5.32. Использование **strcmp** и **strncmp** (часть 1 из 2)

```
s1 = Счастливого Нового Года
s2 = Счастливого Нового Года
s3 = Счастливого Праздника

strcmp(s1, s2) = 0
strcmp(s1, s3) = -2
strcmp(s3, s1) = 2

strncmp(s1, s3, 12) = 0
strncmp(s1, s3, 13) = -2
strncmp(s3, s1, 13) = 2
```

Рис. 5.32. Использование `strcmp` и `strncmp` (часть 2 из 2)

Как компьютер узнает о порядке следования букв? Все символы представляются внутри компьютера как численные коды; когда компьютер сравнивает две строки, он на самом деле сравнивает численные коды символов в строке. (Замечание: коды символов упорядочены по алфавиту только для латинских букв, а к кириллице это, к сожалению, не относится.)

### Замечание по мобильности 5.5

Внутренний численный код, используемый для представления символов, может быть различным для разных компьютеров (особенно это относится к символам кириллицы).

В попытке стандартизации представления символов большинство производителей компьютеров спроектировало свои машины так, чтобы использовать одну из двух популярных кодирующих схем — ASCII или EBCDIC. ASCII означает «Американский стандартный код для информационного обмена» («American Standard Code for Information Interchange»), а EBCDIC означает «Расширенный двоичный код закодированного десятичного обмена» («Extended Binary Coded Decimal Interchange Code»). Существуют и другие схемы кодирования, но эти две наиболее популярны.

ASCII и EBCDIC называются *символьными кодами* или *символьными наборами*. Манипуляции со строками и символами на самом деле подразумевают манипуляцию с соответствующими численными кодами, а не с самими символами. Это объясняет взаимозаменяемость символов и малых целых в C++. Так как имеет смысл говорить, что один численный код больше, меньше или равен другому численному коду, стало возможным сопоставлять различные строки и символы друг с другом путем ссылки на коды символов. Приложение В содержит список символьных кодов ASCII.

Функция `strtok` используется для превращения строки в последовательность лексем. Лексема — это последовательность символов, отделенная *символами разделителям* (обычно пробелами или знаками пунктуации). Например, в строке текста каждое слово может рассматриваться как лексема, а пробелы, отделяющие слова друг от друга, можно рассматривать как разделители.

Для разбиения строки на лексемы требуется несколько вызовов функции `strtok` (при условии, что строка содержит больше одной лексемы). Первый вызов `strtok` содержит два аргумента: строку, которую нужно разбить на

лексемы, и строку, которая содержит символы, разделяющие лексемы (т.е. разделители). В программе на рис. 5.33 оператор

```
tokenPtr = strtok(string, " ");
```

присваивает `tokenPtr` указатель на первую лексему в `string`. Второй аргумент `strtok`, `" "` указывает, что лексемы в `string` разделяются пробелами. Функция `strtok` отыскивает первый символ в `string`, не являющийся разделителем (пробелом). Это начало первой лексемы. Затем функция находит следующий разделительный символ в строке и заменяет его нулевым символом ('`\0`'). Этим заканчивается текущая лексема. Функция `strtok` сохраняет указатель на следующий символ, стоящий в `string` за данной лексемой, и возвращает указатель на текущую лексему.

```
//Использование strtok
#include <iostream.h>
#include <string.h>

main()
{
 char string[] = "Это предложение содержит пять лексем";
 char *tokenPtr;

 cout << "Строка, разбиваемая на лексемы:" << endl << string
 << endl << endl << "Лексемы:" << endl;

 tokenPtr = strtok(string, " ");
 while (tokenPtr != NULL) {
 cout << tokenPtr << endl;
 tokenPtr = strtok(NULL, " ");
 }
 return 0;
}
```

**Строка, разбиваемая на лексемы:**  
**Это предложение содержит пять лексем**

**Лексемы:**  
**Это**  
**предложение**  
**содержит**  
**пять**  
**лексем**

**Рис. 5.33.** Использование `strtok`

Последующие вызовы `strtok` для продолжения разбиения `string` на лексемы содержат в качестве первого аргумента `NULL`. Аргумент `NULL` указывает, что вызов `strtok` должен продолжать разбиение на лексемы, начиная с ячейки в `string`, сохраненной последним вызовом `strtok`. Если лексем при вызове `strtok` больше не оказалось, `strtok` возвращает `NULL`. Программа на рис. 5.33 использует `strtok` для разбиения на лексемы строки «Это предложение содержит пять лексем». Каждая лексема печатается отдельно. Заметим, что `strtok` модифицирует входную строку; поэтому, если строка после вызова `strtok` будет снова использоваться в программе, необходимо сделать копию строки.

### Типичная ошибка программирования 5.20

Непонимание того, что **strtok** изменяет разбиваемую на лексемы строку, и последующая попытка использовать эту строку, как если бы она была исходной неизмененной строкой.

Функция **strlen** получает в качестве аргумента строку и возвращает количество символов в строке — завершающий нулевой символ в длину не включается. Программа на рис. 5.34 демонстрирует функцию **strlen**.

```
//Использование strlen
#include <iostream.h>
#include <string.h>

main()
{
 char *string1 = "abcdefghijklmnopqrstuvwxyz";
 char *string2 = "три";
 char *string3 = "Бостон";

 cout << "Длина строки \""
 << string1
 << "\" - " << strlen(string1) << endl
 << "Длина строки \""
 << string2
 << "\" - " << strlen(string2) << endl
 << "Длина строки \""
 << string3
 << "\" - " << strlen(string3) << endl;
 return 0;
}
```

---

```
Длина строки "abcdefghijklmnopqrstuvwxyz" - 26
Длина строки "три" - 3
Длина строки "Бостон" - 6
```

Рис. 5.34. Использование **strlen**

## 5.13. Размышления об объектах: взаимодействие объектов

Это последнее из наших заданий на объектно-ориентированное проектирование, перед тем как мы начнем изучение объектно-ориентированного программирования на C++ в главе 6. После того как вы завершите это задание и закончите чтение главы 6, вы будете готовы (и, возможно, будете полны страстного желания) начать кодирование вашей модели лифта. Чтобы завершить модель лифта, определенную в главе 2, вам необходима технология C++, обсуждаемая в главах 6, 7, и 8. Затем мы предложим модификации модели, которые потребуют применения наследования и полиморфизма, обсуждаемых в главах 9 и 10.

В этом разделе мы займемся взаимодействием объектов. Мы надеемся, что раздел поможет вам «связать все вместе». Возможно, вы пополните списки объектов, их атрибутов и вариантов поведения, разработанные вами в лабораторных заданиях глав 2, 3 и 4.

Мы узнаем, что большинство объектов в C++ работают отнюдь не спонтанно. Правильнее сказать, что объекты отзываются на стимулы в форме сообщений, которые на самом деле являются вызовами функций-элементов объектов.

Рассмотрим некоторые взаимодействия объектов при моделировании лифта. Постановка задачи гласит: «Пассажир нажимает на этаже кнопку «вниз» или кнопку «вверх». «Субъект» этого предложения — пассажир, а объект — кнопка. Это пример взаимодействия между объектами. Объект-пассажир посыпает сообщение объекту-кнопке. Мы назвали это сообщение `pushButton` (нажатие кнопки). В последней главе мы сделали это сообщение функцией-элементом объекта кнопки.

Теперь в разделах *Другие Факты* для каждого объекта в вашей модели вы должны указать взаимодействия между объектами. Некоторые из них легко увидеть. Но рассмотрим утверждение

#### *«пассажир ждет открытия дверей лифта»*

В последней главе мы рассмотрели два варианта поведения дверей лифта, а именно, `openDoor` (открывание двери) и `closeDoor` (закрывание двери). Но теперь мы хотим определить, какие объекты посыпают соответствующие сообщения. Это неявно указано в предыдущем утверждении, заключенном в кавычки. Немного подумав, мы сделаем вывод, что лифт сам посыпает эти сообщения дверям. Подобные взаимодействия объектов неявно подразумеваются в формулировке задачи.

Теперь продолжите совершенствование разделов *Другие Факты* для каждого объекта в вашей модели лифта. Эти разделы теперь должны содержать большинство взаимодействий объектов.

Представим каждое из них в виде

1. объект-отправитель
2. посыпает конкретное сообщение
3. конкретному объекту-получателю.

Для каждого объекта добавим раздел *Сообщения, посыпаемые другим объектам* и внесем в него соответствующие взаимодействия, например, для объекта «пассажир» внесем запись

*Пассажир посыпает сообщение `pushButton` кнопке на этом этаже*

Под объектом кнопка в раздел *Сообщения, посыпаемые другим объектам* поместим сообщение

*Кнопка посыпает сообщение `comeGetMe` (иди ко мне) лифту*

Когда вы сделали эти записи, вы можете добавить вашим объектам атрибуты и варианты поведения. Это совершенно естественно.

После того, как вы завершите это лабораторное задание, вы будете иметь вполне приемлемое перечисление объектов, которые вам нужны для реализации вашей модели лифта. Для каждого объекта вы получите вполне приемлемое перечисление атрибутов, вариантов поведения объекта и сообщений, которые данный объект посыпает другим объектам.

В следующей главе мы начнем изучение объектно-ориентированного программирования на C++. Вы научитесь создавать новые классы. Когда вы прочтете главу 6, вы будете готовы написать существенную часть модели лифта на C++. После освоения глав 7 и 8 вы будете достаточно обучены, чтобы реализовать действующую модель лифта.

Давайте вспомним основные моменты процесса объектно-ориентированного проектирования, рассмотренного в главах 2–5.

1. Постановка задачи записывается в текстовый файл.
2. Исключается малосущественный текст.
3. Извлекаются все факты. Факты размещаются по одному в строке в файле фактов.
4. Просматриваются все факты, описанные существительными; они образуют наиболее вероятное множество необходимых вам объектов. Создается по одному элементу верхнего уровня схемы для каждого объекта.
5. Остальные факты размещаются в качестве элементов второго уровня ниже соответствующего объекта. Если факт относится к нескольким объектам (как получается для многих фактов), он помещается ниже каждого объекта, к которому он относится.
6. Теперь совершенствуется множество фактов, относящихся к каждому объекту. Ниже каждого объекта создаются три списка с подзаголовками *Атрибуты*, *Варианты поведения* и *Сообщения, посылаемые другим объектам*.
7. В списке *Атрибуты* перечисляются данные, связанные с каждым объектом.
8. В списке *Варианты поведения* перечисляются действия объектов, которые производятся в ответ на принятые сообщения. Каждый вариант поведения является функцией элементом объекта.
9. В списке *Сообщения, посылаемые другим объектам* перечисляются сообщения, посылаемые данным объектом другим объектам, а также объекты, которые принимают эти сообщения.
10. К этому моменту ваш проект еще имеет, возможно, некоторые белые пятна. Они, вероятно, станут проявляться по мере того, как вы будете продолжать реализацию вашей системы на C++ после чтения главы 6.

## Резюме

- Указатели — это переменные, которые содержат в качестве своих значений адреса других переменных.
- Указатели должны быть объявлены до того, как они могут быть использованы.
- Объявление

```
int *ptr;
```

объявляет `ptr` указателем на объект типа `int` и читается как «`ptr` — это указатель на `int`». Символ `*` в объявлении указывает, что переменная является указателем.

- Для присваивания указателю начального значения можно использовать три значения: **0**, **NULL** или адрес объекта того же типа. Присваивание указателю в качестве начального значения **0** или **NULL** эквивалентно.
- Единственным целым, которое может быть присвоено указателю, является **0**.
- Операция адресации **&** возвращает адрес своего операнда.
- Операнд операции адресации должен быть переменной; операция адресации не применима к константам, выражениям, которые не возвращают ссылку, и переменным, объявленным с классом памяти **register**.
- Операция **\***, называемая операцией косвенной адресации или операцией разыменования, возвращает значение объекта, на который указывает ее operand. Это называется разыменованием указателя.
- Если аргумент вызываемой функции должен ею изменяться, можно в качестве параметра передавать адрес аргумента. Вызванная функция затем может модифицировать значение аргумента в вызывающей функции, используя операцию разыменования **\***.
- Функция, принимающая адрес в качестве аргумента, должна в качестве соответствующего формального параметра иметь указатель.
- Нет необходимости включать в прототипы функций имена указателей; единственное, что нужно включить, — это типы указателей. Имена указателей можно включать с целью документирования, но компилятор их игнорирует.
- Спецификатор **const** дает программисту возможность информировать компилятор о том, что значение данной переменной не должно модифицироваться.
- Компилятор отлавливает попытки модифицировать значения, объявленные как **const**, и выдает либо предупреждение, либо сообщение об ошибке в зависимости от типа компилятора.
- Существуют четыре способа передачи в функцию указателя: неконстантный указатель на неконстантные данные, константный указатель на неконстантные данные, неконстантный указатель на константные данные и константный указатель на константные данные.
- Массивы автоматически передаются ссылкой, использующей указатели, потому что значение имени массива является адресом массива.
- Чтобы передать по ссылке с использованием указателей один элемент массива, нужно передать адрес этого элемента.
- В C++ имеется специальная унарная операция **sizeof** для определения размера массива (или любого другого типа данных) в байтах во время компиляции программы.
- Если операция **sizeof** применяется к имени массива, она возвращает как целое общее количество байтов в массиве.
- Операцию **sizeof** можно применять к любым именам переменных, типам или константам.

- К числу арифметических операций, которые можно выполнять над указателями, относятся инкремент (`++`) указателя, декремент указателя (`--`), сложение (`+` или `+=`) указателя и целого, вычитание (`-` или `=-`) указателя и целого и вычитание одного указателя из другого.
- При сложении или вычитании указателя и целого указатель увеличивается или уменьшается на величину, равную произведению этого целого на размер указанного объекта.
- Арифметические операции могут выполняться только с указателями, указывающими на смежные участки памяти, такие, как массив. Все элементы массива хранятся в памяти непосредственно друг за другом.
- При выполнении арифметических операций над указателем на массив символов результаты не отличаются от обычной арифметики, потому что каждый символ хранится в одном байте памяти.
- Указатели можно присваивать один другому, если оба указателя одного и того же типа. В противном случае нужно использовать приведение типов. Исключением из этого правила является указатель на `void`, который является общим указателем, способным представлять указатели любого типа. Указателю на `void` можно присваивать все типы указателей без приведения типа. Однако указатель на `void` может быть присвоен указателю другого типа только явным приведением к типу соответствующего указателя.
- Указатель на `void` не может быть разыменован.
- Указатели можно сравнивать, используя операции проверки на равенство и отношения. Сравнение указателей имеет смысл только в случае, если они указывают на элементы одного и того же массива.
- Указатели можно индексировать точно так же, как имена массивов.
- Имя массива эквивалентно указателю на первый элемент массива.
- В записи `указатель-смещение` `смещение` — это то же самое, что индекс массива.
- Все выражения с индексами массива можно записать с помощью указателя и смещения, используя либо имя массива как указатель, либо отдельный указатель, указывающий на массив.
- Имя массива — это постоянный указатель, который всегда указывает на одну и ту же ячейку памяти. Но в отличие от обычных указателей имена массивов нельзя изменять.
- Можно использовать массивы указателей.
- Можно использовать указатели на функции.
- Указатель на функцию — это адрес, по которому расположен код функции.
- Указатели на функции можно передавать функциям, возвращать из функций, хранить в массивах и присваивать другим указателям.
- Типичным применением указателей на функции являются так называемые системы, управляемые меню. Указатели на функции используются для вызова функций, выбираемых с помощью отдельных разделов меню.

- Функция `strcpy` копирует свой второй аргумент — строку в свой первый аргумент — массив символов. Программист должен быть уверен, что массив достаточно велик, чтобы хранить строку и ее завершающий нулевой символ.
- Функция `strncpy` эквивалентна `strcpy`, за исключением того, что в вызове `strncpy` указывается количество символов, которое должно быть скопировано из строки в массив. Завершающий нулевой символ будет копироваться, только если количество символов, которое должно быть скопировано, по крайней мере на один больше длины строки.
- Функция `strcat` добавляет свой второй аргумент — строку к своему первому аргументу — массиву символов, содержащему строку. Первый символ второго аргумента замещает нулевой символ ('\0'), который завершал строку в первом аргументе. Программист должен быть уверен, что массив, используемый для хранения первой строки, достаточно велик для того, чтобы хранить комбинацию первой строки и второй строки.
- Функция `strncat` добавляет указанное количество символов из второй строки в первую строку. К результату добавляется завершающий нулевой символ.
- Функция `strcmp` сравнивает символ за символом строку в своем первом аргументе со строкой в своем втором аргументом. Функция возвращает 0, если строки равны, отрицательное значение, если первая строка меньше, чем вторая, и положительное значение, если первая строка больше, чем вторая.
- Функция `strncmp` эквивалентна `strcmp`, за исключением того, что `strncmp` сравнивает указанное количество символов. Если количество символов одной из строк меньше чем указанное количество символов, `strncmp` сравнивает символы до тех пор, пока не встретится нулевой символ в более короткой строке.
- Последовательные вызовы функции `strtok` разбивают строку на лексемы, разделенные символами, содержащимися во втором аргументе — строке. При первом вызове в качестве первого аргумента передается строка, разбиваемая на лексемы, а при последующих вызовах, продолжающих разбиение на лексемы той же самой строки, в качестве первого аргумента передается NULL. При каждом вызове возвращается указатель на текущую лексему. Если при вызове `strtok` лексем больше нет, возвращается NULL.
- Функция `strlen` получает в качестве аргумента строку, а возвращает количество символов в строке; завершающий нулевой символ в длину строки не входит.

## Терминология

|        |          |
|--------|----------|
| ASCII  | strcmp   |
| const  | strcpy   |
| EBCDIC | strihg.h |
| sizeof | strlen   |
| strcat | strncat  |

|                                   |                                                 |
|-----------------------------------|-------------------------------------------------|
| strcmp                            | неконстантный указатель на константные данные   |
| strncpy                           | неконстантный указатель на неконстантные данные |
| strtok                            | неопределенная отсрочка                         |
| void * (указатель на void)        | обработка строк                                 |
| арифметические операции           | операция адресации (&)                          |
| с указателями                     | операция косвенной адресации (*)                |
| вызов по значению                 | операция разыменования (*)                      |
| вызов по ссылке                   | представление символа численным кодом           |
| выражение с указателями           | принцип наименьших привилегий                   |
| вычитание двух указателей         | присваивание указателей                         |
| вычитание целого из указателя     | присваивание указателям начальных значений      |
| декремент указателя               | прямая ссылка на переменную                     |
| длина строки                      | разбиение строки на лексемы                     |
| добавление строки к другой строке | разделитель                                     |
| запись указатель-смещение         | связный список                                  |
| индексирование указателей         | символ разделитель                              |
| инкремент указателя               | символьная константа                            |
| код символа                       | сложение указателя и целого                     |
| константный указатель             | смещение                                        |
| константный указатель             | соединение (конкатенация) строк                 |
| на константные данные             | сравнение строк                                 |
| константный указатель             | сравнение указателей                            |
| на неконстантные данные           | строка                                          |
| копирование строк                 | строковая константа                             |
| косвенная адресация               | типы указателей                                 |
| косвенная ссылка                  | указатель                                       |
| на переменную                     | указатель NULL                                  |
| лексема                           | указатель на void (void *)                      |
| лингвистический процессор         | указатель на символ                             |
| литеральная константа             | указатель на функцию                            |
| массив строк                      |                                                 |
| массив указателей                 |                                                 |
| моделируемый вызов по ссылке      |                                                 |
| набор символов                    |                                                 |

## Типичные ошибки программирования

- 5.1. Операция \* косвенной адресации не распространяется на все имена переменных в объявлении. Каждый указатель должен быть объявлен с помощью символа \*, стоящего перед именем.
- 5.2. Разыменование указателя, который не был должным образом иницирован или которому не присвоено указание на конкретное место в памяти. Это может вызвать неисправимую ошибку выполнения или может неожиданно изменить важные данные и программа завершится неверными результатами.
- 5.3. Не разыменовывается указатель, когда это необходимо сделать, чтобы получить значение, на которое указывает этот указатель.
- 5.4. Отсутствие присваивания начального значения указателю, который объявлен как const, приводит к ошибке компиляции.
- 5.5. Пропуск скобок в операции sizeof, когда операндом является имя типа, вызывает синтаксическую ошибку.

- 5.6. Использование арифметических действий с указателями, не ссылающимися на элементы массива.
- 5.7. Вычитание или сравнение двух указателей, которые не ссылаются на элементы одного и того же массива.
- 5.8. Выход за пределы массива при использовании арифметических действий с указателями.
- 5.9. Присваивание указателя одного типа указателю другого типа (отличного от `void*`) вызывает синтаксическую ошибку.
- 5.10. Разыменование указателя на `void*`.
- 5.11. Хотя имена массивов являются указателями на их начало, а указатели в арифметических выражениях можно модифицировать, имена массивов в этих выражениях модифицировать нельзя.
- 5.12. Не выделяется достаточно места в массиве символов для хранения нулевого символа, завершающего строку.
- 5.13. Создание или использование «строки», которая не содержит завершающего нулевого символа.
- 5.14. Обработка одного символа как строки. Стока является указателем — это может быть достаточно большое целое. А символ — это небольшое целое (диапазон значений ASCII 0–255). Во многих системах это вызывает ошибку, потому что нижние адреса памяти зарезервированы для специальных целей, таких как обработчики прерываний операционной системы — так что происходит «нсанкционированный доступ».
- 5.15. Передача символа в качестве аргумента функции, которая ожидает строку.
- 5.16. Передача строки в качестве аргумента функции, которая ожидает символ.
- 5.17. Забывают включить заголовочный файл `<string.h>` при использовании функций из библиотеки обработки строк.
- 5.18. Не добавляется завершающий нулевой символ к первому аргументу `strncpy`, когда третий аргумент меньше или равен длине строки во втором аргументе.
- 5.19. Предположение, что `strcmp` и `strncmp` возвращают 1, если их аргументы равны. При равенстве обе функции возвращают 0 (значение «ложь» в C++). Поэтому при проверке двух строк на равенство результаты функций `strcmp` или `strncmp` должны для определения равенства строк сравниваться с 0.
- 5.20. Непонимание того, что `strtok` изменяет разбиваемую на лексемы строку, и последующая попытка использовать эту строку, как если бы она была исходной неизмененной строкой.

## Хороший стиль программирования

- 5.1. Хотя это и не обязательно, включайте буквы `Ptr` в имена переменных указателей, чтобы было ясно, что эти переменные являются указателями и требуют соответствующей обработки.
- 5.2. Присваивайте начальные значения указателям во избежание неожиданных результатов.
- 5.3. Используйте передачу по значению аргументов функции до тех пор, пока оператор вызова явно не требует, чтобы вызываемая функция модифицировала значение переменной аргумента в окружении вызывающего оператора. Это еще один пример принципа наименьших привилегий.
- 5.4. Прежде, чем использовать функцию, проверьте прототип функции, чтобы определить, имеет ли она возможность модифицировать передаваемые ей значения.
- 5.5. Используйте нотацию массивов, а не нотацию указателей при манипуляции массивами. Несмотря на то, что программа может оказаться несколько длиннее, она, вероятно, будет более понятной.
- 5.6. При хранении строки в массиве символов убедитесь, что массив достаточно велик, чтобы вместить наибольшую строку, которую потребуется хранить. В C++ допускается хранить строки любой длины. Если строка больше символьного массива, в котором она должна храниться, символы, выходящие за конец массива, будут изменять данные в разделах памяти, следующих за массивом.

## Советы по повышению эффективности

- 5.1. Передавайте большие объекты, такие, как структуры, используя указатели на константные данные или ссылки на константные данные, чтобы получить выигрыш по производительности, даваемый передачей по ссылке, и защиту информации, даваемую передачей по значению.
- 5.2. Иногда в алгоритме, который кажется «естественнным», могут таиться такие тонкие проблемы эффективности, как неопределенная отсрочка. Ищите алгоритмы, которые лишены проблемы неопределенной отсрочки.

## Замечания по мобильности

- 5.1. Хотя спецификация `const` полностью определена в С ANSI, в некоторых компиляторах она не реализована.
- 5.2. Количество байтов, используемое для хранения отдельных типов данных, может быть различным для разных систем. При написании программ, которые зависят от размеров типа данных и которые будут выполняться на различных компьютерных системах, используйте `sizeof` для определения количества байтов, применяемых для хранения различных типов данных.

- 5.3. Большинство современных компьютеров имеет 2-х или 4-х байтовые целые. Некоторые из более новых машин имеют 8-байтовые целые. Поскольку результат арифметических действий с указателями зависит от размера объектов, на которые указывает указатель, арифметические действия с указателями являются машиннозависимыми.
- 5.4. Когда переменная типа `char*` получает в качестве начального значения строковую константу, некоторые компиляторы могут поместить строку в такое место в памяти, где строка не может быть модифицирована. Если у вас может возникнуть необходимость модифицировать строковую константу, она должна храниться в массиве символов, чтобы обеспечить возможность ее модификации во всех системах.
- 5.5. Внутренний численный код, используемый для представления символов, может быть различным для разных компьютеров (особенно это относится к символам кириллицы).

### Замечания по технике программирования

- 5.1. Спецификацию `const` можно использовать для реализации принципа минимизации привилегий. Использование должным образом принципа минимизации привилегий при проектировании программного обеспечения весьма существенно сокращает время отладки и количество нежелательных побочных эффектов, упрощает модификацию и сопровождение программы.
- 5.2. Если значение не изменяется (или не должно изменяться) в теле функции, которой оно передается, параметр должен быть объявлен как `const`, чтобы иметь гарантию от неожиданных изменений его.
- 5.3. При использовании вызова по значению в вызываемой функции можно изменять только одно значение. Это значение должно быть задано как возвращаемое в операторе `return`. Для модификации множества значений в вызываемой функции нужно использовать передачу параметров по ссылке.
- 5.4. Включение прототипов функций в определения других функций реализует принцип наименьших привилегий путем ограничения вызовов соответствующих функций только теми функциями, в которых появляются соответствующие прототипы.
- 5.5. При передаче массива в функции передавайте также и размер массива (это предпочтительнее, чем встраивание в функцию знаний о размере массива). Такой подход позволяет создавать функции более общего вида. Функции общего вида часто могут повторно использоваться во многих программах.

### Упражнения для самопроверки

- 5.1. Заполнить пробелы в следующих утверждениях:
  - а) Указатель — это переменная, которая содержит в качестве своего значения \_\_\_\_\_ другой переменной.

- b) Для присвоения указателю начального значения можно использовать три значения: \_\_\_\_\_, \_\_\_\_\_ или \_\_\_\_\_.
- c) Единственным целым, которое может быть присвоено указателю, является \_\_\_\_\_.
- 5.2. Укажите, справедливы или нет следующие утверждения. Если они ошибочны, укажите почему.
- Операция адресации может быть применима только к константам, к выражениям не имеющим результат, на который можно сослаться, и к переменным, объявленным с классом хранения `register`.
  - Указатель, объявленный как `void`, может быть разыменован.
  - Указатели разных типов нельзя присваивать друг другу без операции приведения типов.
- 5.3. Выполните следующее упражнение предполагая, что числа с одинарной точностью с плавающей запятой хранятся в 4 байтах и что начальный адрес массива — 1002500. Каждая часть упражнения использует соответствующие результаты предыдущих частей.
- Объявите массив типа `float` с именем `numbers` с 10 элементами и присвойте элементам начальные значения 0.0, 1.1, 2.2, ..., 9.9. Предполагайте, что определена символьская константа `SIZE`, равная 10.
  - Объявите указатель `nPtr`, который указывает на объект типа `float`.
  - Напечатайте элементы массива `numbers`, используя запись индексов массива. Используйте структуру `for` и предполагайте, что была объявлена целая управляющая переменная `i`. Напечатайте каждое число с одной значащей цифрой справа от десятичной точки.
  - Приведите два различных оператора, которые присваивают начальный адрес массива `numbers` переменной указателю `nPtr`.
  - Напечатайте элементы массива `numbers`, используя указатель `nPtr`.
  - Напечатайте элементы массива `numbers`, используя запись указатель-смещение с именем массива как указателем.
  - Напечатайте элементы массива `numbers`, используя индексацию указателя `nPtr`.
  - Сошлитесь на элемент 4 массива `number`, используя запись индекса массива, запись указатель-смещение с именем массива как указателем, запись индекса указателя `nPtr` и запись указатель-смещение с `nPtr`.
  - Предполагая, что `nPtr` указывает на начало массива `numbers`, определите, на какой адрес ссылается выражение `nPtr + 8`? Какое значение хранится по этому адресу?
  - Предполагая, что `nPtr` указывает на `numbers[5]`, определите, на какой адрес будет ссылаться `nPtr` после выполнения оператора `nPtr -= 4`. Какое значение хранится по этому адресу?
- 5.4. Для каждого из следующих пунктов напишите один оператор, который выполняет указанное задание. Предполагайте, что переменные с плавающей запятой `number1` и `number2` уже объявлены и

что `number1` получила начальное значение 7.3. Кроме того, предполагайте, что переменная `ptr` имеет тип `char*`, а массивы `s1[100]` и `s2[100]` — тип `char`.

- Объявите переменную `fPtr` как указатель на объект типа `float`.
- Присвойте адрес переменной `number1` указателю переменной `fPtr`.
- Напечатайте значение объекта, на который указывает `fPtr`.
- Присвойте значение объекта, на который указывает `fPtr`, переменной `number2`.
- Напечатайте значение `number2`.
- Напечатайте адрес `number1`.
- Напечатайте адрес, хранимый в `fPtr`. Совпадает ли напечатанное значение с адресом `number1`?
- Скопируйте строку, хранимую в массиве `s2`, в массив `s1`.
- Сравните строку `s1` со строкой `s2`.
- Добавьте 10 символов из строки `s2` в строку `s1`.
- Определите длину строку `s1`. Напечатайте результат.
- Присвойте `ptr` позицию первой лексемы в `s2`. Лексемы в `s2` разделены запятыми (,).

#### 5.5. Проделайте указанное в каждом пункте:

- Напишите заголовок функции `exchange`, которая получает в качестве параметров два указателя на числа с плавающей запятой `x` и `y` и не возвращает никакого значения.
- Напишите прототип функции из пункта а).
- Напишите заголовок функции `evaluate`, которая возвращает целое и которая получает в качестве параметров целое и указатель на функцию `poly`. Функция `poly` получает целый параметр и возвращает целое.
- Напишите прототип функции из пункта с).
- Покажите два разных метода присваивания в качестве начальных условий массиву символов `vowel` строки гласных «AEIOU».

#### 5.6. Найдите ошибку в каждом из следующих фрагментов программ. Полагайте, что

```

int *zPtr; // zPtr будет ссылаться 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
 // значения массива
 number = zPtr;
c) // присваивание переменной number значения элемента
 // 2 массива (значение 3)
 number = *zPtr[2];

```

- d) // печать всего массива z  
     for (i = 0; i <= 5; i++)  
         cout << zPtr[ i ] << endl;
- e) // присваивание значения, указываемого sPtr,  
     // переменной number  
     number = \*sPtr;
- f) ++z;
- g) char s[10];  
     cout << strncpy(s, "hello", 5) << endl;
- h) char s[12];  
     strcpy(s, "Welcome Home");
- i) if ( strcmp(string1, string2) )  
         cout << "Строки равны" << endl;

- 5.7. Что печатается (если что-либо печатается) при выполнении каждого из следующих операторов? Если оператор содержит ошибку, опишите ее и укажите, как ее исправить. Предполагайте следующие объявления переменных:

```
char s1[50] = "jack", s2[50] = "jill", s3[50], *sptr;
a) cout << strcpy(s3, s2) << endl;
b) cout << strcat(strcat(strcpy(s3, s1), " and "), s2)
 << endl;
c) cout << strlen(s1) + strlen(s2) << endl;
d) cout << strlen(s3) << endl;
```

### Ответы на упражнения для самопроверки

- 5.1. a) адрес. b) 0, NULL, адрес. c) 0.
- 5.2. a) Ошибка. Операция адресации может быть применена только к переменным и не может быть применена к константам, выражениям или переменным, объявленным с классом хранения register.  
     б) Ошибка. Указатель на void не может быть разыменован, потому что не существует способа узнать точно, сколько байтов памяти должно быть разыменовано.  
     в) Ошибка. Указателю на void можно присваивать все типы указателей без приведения типа. Однако указатель на void может быть присвоен указателю другого типа только явным приведением к типу соответствующего указателя.
- 5.3. a) float numbers[SIZE] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5,
                                6.6, 7.7, 8.8, 9.9, };
- б) float \*nPtr;
- в) cout << setiosflags(ios::fixed | ios::showpoint)
           << setprecision(1);
         for (i = 0; i < SIZE; i++)
           cout << numbers[i] << ' ';
- д) nPtr = numbers;
         nPtr = &numbers[0];

- e) cout << setiosflags(ios::fixed | ios::showpoint)  
     << setprecision(1);  
     for (i = 0; i < SIZE; i++)  
         cout << \*(nPtr + i) << ' ';
- f) cout << setiosflags(ios::fixed | ios::showpoint)  
     << setprecision(1);  
     for (i = 0; i < SIZE; i++)  
         cout << \*(numbers + i) << ' ';
- g) cout << setiosflags(ios::fixed | ios::showpoint)  
     << setprecision(1);  
     for (i = 0; i < SIZE; i++)  
         cout << nPtr[ i ] << ' ';
- h) number[4]  
     \*(numbers + 4)  
     nPtr[4]  
     \*(nPtr + 4)
- i) Адрес равен  $1002500 + 8 * 4 = 1002532$ . Значение равно 8.8.
- j) Адрес **number [5]** равен  $1002500 + 5 * 4 = 1002520$ .  
 Адрес **nPtr - 4** равен  $1002520 - 4 * 4 = 1002504$ .  
 Значение равно 1.1.

- 5.4.** a) float \*fPtr;  
 b) fPtr = &number1;  
 c) cout << "Значение \*fPtr равно" << \*fPtr << endl;  
 d) number2 = \*fPtr;  
 e) cout << "Значение number2 равно" << number2 << endl;  
 f) cout << "Адрес number1 равен" << &number1 << endl;  
 g) cout << "Адрес, хранимый в fPtr, равен" << fPtr << endl;  
 Да, значение то же самое.
- h) strcpy(s1, s2);  
 i) cout << "strcmp(s1, s2) = " << strcmp(s1, s2) << endl;  
 j) strncat(s1, s2, 10);  
 k) cout << "strlen(s1) = " << strlen(s1) << endl;  
 l) ptr = strtok(s2, ",");
- 5.5.** 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));  
 e) char vowe[ ] = "AEIOU";  
     char vowe[] = {'A', 'E', 'I', 'O', 'U', '\0'};
- 5.6.** a) Ошибка: zPtr не получил начального значения.  
 Исправление: присвоить zPtr начальное значение zPtr = z;  
 b) Ошибка: указатель не разыменован.  
 Исправление: изменить оператор на number = \*zPtr;

- с) Ошибка: `zPtr[2]` не указатель и не может быть разыменован.  
Исправление: изменить `*zPtr[2]` на `zPtr[2]`.
- д) Ошибка: ссылка с помощью индексации указателя на элемент массива, находящийся вне его границ.  
Исправление: изменить операцию отношения в структуре `for` на `<`, чтобы избежать выхода за конец массива.
- е) Ошибка: разыменование указателя `void`.  
Исправление: чтобы разыменовать указатель, он сначала должен быть приведен к целому указателю. Измените оператор на `number = *(int *)sPtr;`
- ф) Ошибка: попытка изменить имя массива с помощью арифметической операции над указателем.  
Исправление: или для выполнения арифметических действий с указателями используйте переменную указатель вместо имени массива, или индексируйте имя массива, чтобы сослаться на отдельный элемент.
- г) Ошибка: функция `strncpy` не записывает завершающий нулевой символ в массив `s`, потому что ее третий аргумент равен длине строки `"hello"`.  
Исправление: сделайте третий аргумент `strncpy` равным 6 или присвойте `s[5]` значение `'\0'`, чтобы быть уверенным, что завершающий нулевой символ добавляется к строке.
- х) Ошибка: массив символов `s` недостаточно велик для хранения завершающего нулевого символа.  
Исправление: Объявите массив с большим числом элементов.
- и) Ошибка: функция `strcmp` возвратит 0, если строки равны, поэтому условие в структуре `if` будет ложным и оператор вывода данных не будет выполнен.  
Исправление: явно сравнить результат `strcmp` с 0 в условии структуры `if`.

- 5.7. a) jill  
b) jack and jill  
c) 8  
d) 13

## Упражнения

- 5.8. Определите, являются ли следующие утверждения правильными. Если нет, то объясните почему.
- а) Бессмысленно сравнивать два указателя, указывающие на разные массивы.
- б) Поскольку имя массива является указателем на его первый элемент, именами массивов можно манипулировать точно так же, как указателями.
- 5.9. Решите следующие задачи. Полагайте, что беззнаковые целые хранятся в 2 байтах, и что начальный адрес массива находится в ячейке памяти 1002500.

- a) Объявите массив `values` типа `unsigned int` с 5 элементами и присвойте элементам начальные значения в виде четных чисел от 2 до 10. Полагайте, что определена символьская константа `SIZE`, равная 5.
- b) Объявите указатель `vPtr`, который указывает на объект типа `unsigned int`.
- c) Напечатайте элементы массива `values`, используя запись с индексом массива. Используйте структуру `for` и считайте, что целая управляющая переменная `i` уже была объявлена.
- d) Укажите два разных оператора, которые присваивают начальный адрес массива `address` переменной указателю `vPtr`.
- e) Напечатайте элементы массива `values`, используя запись указатель-смещение.
- f) Напечатайте элементы массива `values`, используя запись указатель-смещение с именем массива как указателем.
- g) Напечатайте элементы массива `values`, используя индексацию указателя на массив.
- h) Скопируйтесь на элемент 5 массива `values`, используя запись с индексом массива, запись с именем массива как указателем, запись с индексом указателя и запись указатель-смещение.
- i) На какой адрес ссылается выражение `vPtr + 3`? Какое значение хранится в этой ячейке?
- j) Предполагая, что `vPtr` указывает на `values[4]`, укажите, на какой адрес ссылается выражение `vPtr - 4`. Какое значение хранится в этой ячейке?

**5.10.** В каждом из следующих заданий напишите один оператор, который выполняет указанное действие. Считайте, что длинные целые переменные `value1` и `value2` уже объявлены и что переменной `value1` присвоено начальное значение 200000.

- a) Объявите переменную `lPtr` указателем на объект типа `long`.
- b) Присвойте адрес переменной `value1` переменной указателю `lPtr`.
- c) Напечатайте значение объекта, на который указывает `lPtr`.
- d) Присвойте значение объекта, на который указывает `lPtr`, переменной `value2`.
- e) Напечатайте значение `value2`.
- f) Напечатайте адрес `value2`.
- g) Напечатайте адрес, который хранится в `lPtr`. Совпадает ли напечатанное значение с адресом `value1`.

**5.11.** Выполните следующее:

- a) Напишите заголовок функции `zero`, которая получает в качестве параметра массив длинных целых значений `bigIntegers` и не возвращает никакого значения.
- b) Напишите прототип функции пункта а).

- с) Напишите заголовок функции `add1Andsumzero`, которая получает в качестве параметра массив целых `oneTooSmall` и возвращает целое.  
д) Напишите прототип функции, описанной в пункте с).

*Замечание: упражнения с 5.12 по 5.15 достаточно необычны. Когда вы решите эти задачи, вы будете в состоянии легко разрабатывать программы для самых популярных карточных игр.*

- 5.12. Модифицируйте программу на рис. 5.24 так, чтобы функция раздачи карт раздавала на руки пятикарточный покер. Затем напишите следующие дополнительные функции:
- Определите, находится ли на руках пара.
  - Определите, находятся ли на руках две пары.
  - Определите, находятся ли на руках три валета.
  - Определите, находятся ли на руках четыре туза.
  - Определите, находится ли на руках флеш (все пять карт одной масти).
  - Определите, находятся ли на руках пять карт с последовательными фигурами.
- 5.13. Используйте разработки упражнения 5.12 для написания программы, которая раздает два пятикарточных покера, оценивает раздачу в каждые руки и определяет, какая из них лучше.
- 5.14. Модифицируйте программу, созданную в упражнении 5.13, так, чтобы она могла моделировать сдающего. Пять карт сдающего сдаются рубашкой вверх, так что игрок не может видеть фигуры на картах. Программа должна затем оценить карты на руках сдающего и, основываясь на их качестве, сдающий должен засветить дополнительно одну, две или три дополнительные карты, чтобы заменить соответствующее количество ненужных карт в исходной раздаче. Затем программа должна оценить заново раздачу на руки. (*Предостережение: это трудная задача!*)
- 5.15. Модифицируйте программу, разработанную в упражнении 5.14, так, чтобы она могла автоматически обрабатывать раздачу на руки сдающего, но позволяла игроку решать, какие карты из имеющихся на руках заменять. Программа должна затем оценивать оба набора карт на руках и определять победителя. Теперь используйте эту новую программу, чтобы сыграть 20 партий против компьютера. Кто выиграл больше, вы или компьютер? Дайте сыграть 20 партий против компьютера одному из ваших друзей. Кто выиграл больше? Основываясь на результатах этих игр, введите в программу соответствующие модификации для улучшения ее работы (это тоже сложная задача). Сыграйте еще 20 партий. Улучшилась ли игра вашей модифицированной программы?
- 5.16. В программе тасования и раздачи карт на рис. 5.24 мы активно использовали неэффективный алгоритм тасования, допускающий возможность неопределенной отсрочки. В данной задаче вы должны

будете создать высокоэффективный алгоритм тасования, который позволяет избежать неопределенной отсрочки.

Модифицируйте программу на рис. 5.24 следующим образом. Начните с задания начальных значений массиву `deck`, как показано на рис. 5.35. Модифицируйте функцию `shuffle`, чтобы в цикле обрабатывать строка за строкой и столбец за столбцом поочередно каждый элемент массива. Каждый элемент должен переставляться со случайно выбранным элементом массива.

Напечатайте результирующий массив, чтобы определить, удовлетворительно ли перетасована колода (например, как на рис. 5.36). У вас может появиться желание вызвать функцию `shuffle` несколько раз, чтобы быть уверенным в удовлетворительном тасовании.

|   | 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. 5.35. Нетасованный массив `deck`

|   | 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. 5.36. Пример тасованного массива `deck`

Заметим, что хотя рассмотренный в этой задаче подход улучшает алгоритм тасования, алгоритм раздачи все еще требует поиска в массиве `deck` карты 1, затем карты 2, потом карты 3 и т.д. Хуже того, даже после того как алгоритм раздачи раздаст карты, он продолжает поиск в остатке колоды. Модифицируйте программу на рис. 5.24 так, чтобы, как только карта сдана, не было в дальнейшем попытка найти этот номер карты и программа немедленно сдавала бы следующую карту. В главе 16 мы разработаем алгоритм раздачи карт, который требует всего одну операцию на карту.

**5.17. (Моделирование: черепаха и заяц).** В этой задаче мы воссоздадим один из истинно великих моментов истории, а именно, классическую гонку черепахи и зайца. Вы будете использовать генерацию случайных чисел для разработки модели этого памятного события.

Наши соперники начинают гонку в первой клетке из 70. Каждая клетка представляет собой возможную позицию вдоль трассы гонки. Первый из соперников, достигший или миновавший клетку 70, награждается ведром моркови и салата. Трасса вьется по склону скользкой горы, так что соперникам случается падать на землю.

Имеются часы, которые отбивают такт раз в секунду. С каждым тактом ваша программа должна устанавливать позицию животных согласно следующим правилам:

| Животное | Тип движения         | Процент времени | Перемещение     |
|----------|----------------------|-----------------|-----------------|
| Черепаха | Быстрое ползанье     | 50%             | 3 клетки вправо |
|          | Скольжение           | 20%             | 6 клеток влево  |
|          | Медленное ползанье   | 30%             | 1 клетка вправо |
| Заяц     | Сон                  | 20%             | Движения нет    |
|          | Большой прыжок       | 20%             | 9 клеток вправо |
|          | Сильное скольжение   | 10%             | 12 клеток влево |
|          | Маленький прыжок     | 30%             | 1 клетка вправо |
|          | Маленькое скольжение | 20%             | 2 клетки влево  |

Используйте переменные для отслеживания позиций животного (позиции с номерами 1–70). Каждое животное стартует с позиции 1 («стартовые ворота»). Если животное скользит налево и должно оказаться перед клеткой 1, движение начинается заново с клетки 1.

Используйте проценты в приведенной таблице, вырабатывая случайные целые числа  $i$  в диапазоне от 1 до 10 включительно. Для черепахи при  $i$  от 0 до 5 — быстрое ползанье, при  $i$  от 6 до 7 — скольжение, при  $i$  от 8 до 10 — медленное ползанье. Аналогичный прием используйте для движения зайца.

Начните движение печатью

**ГОНГ !!!!!**

**И ОНИ ПОШЛИ !!!!!**

Затем на каждом такте (т.е. при каждом повторении цикла) печатайте 70-позиционную строку, показывающую букву Ч в позиции черепахи и букву З в позиции зайца. Может случиться, что оба животных окажутся на одной и той же клетке. В этом случае черепахакусает зайца и ваша программа должна печатать **ОХ!!!**, начиная с этой позиции. Все печатаемые позиции, отличающиеся от Ч, З или ОХ!!!, должны быть пробелами.

После печати каждой строки проверяйте, достигло или миновало животное клетку 70. Если да, то печатайте, кто победитель, и заканчивайте моделирование. Если победила черепаха, печатайте **ПОБЕДИЛА ЧЕРЕПАХА!!! УРА!!!** Если победил заяц, печатайте: **Победил заяц. Эх.** Если оба животных победили одновременно, вы можете отдать предпочтение черепахе или напечатать: **Равный счет.** Если ни одно животное не победило, повторяйте цикл моделирования для следующего отрезка времени. Если вы готовы запустить программу на выполнение, пригласите группу болельщиков. Вы будете изумлены реакцией вашей аудитории!

### Специальный раздел: построение вашего собственного компьютера

В следующих нескольких задачах мы покинем мир программирования на языке высокого уровня. Мы «разднем донага» компьютер и посмотрим его внутреннюю структуру. Мы познакомимся с ма-

шинным языком программирования и напишем несколько программ на машинном языке. Чтобы закрепить этот особенно значимый опыт, мы построим затем компьютер (основываясь на технике программного моделирования), на котором вы сможете выполнять программы на машинном языке!

**5.18. (Программирование на машинном языке)** Создадим компьютер, который назовем Простотрон. В соответствии со своим именем это простая машина, но, как мы вскоре увидим, достаточно мощная. Простотрон выполняет программы, написанные на единственном понимаемом им языке — языке машины Простотрон, или, для краткости, ЯМП.

Простотрон содержит *аккумулятор* — специальный регистр, в котором находится информация перед тем, как она будет использована Простотроном в вычислениях или специальным образом исследована. Вся информация в Простотроне обрабатывается в терминах слов. Слово — это знаковое четырехзначное десятичное число, например, +3364, -1293, +0007, -0001 и т.д. Простотрон имеет память на 100 слов и эти слова доступны по номерам их ячеек 00, 01, ..., 99.

Перед запуском программы на ЯМП мы должны загрузить ее или разместить в памяти. Первая команда (или оператор) каждой программы на ЯМП всегда помещается в ячейку 00. Моделирующая программа начинает выполнение с этой ячейки.

Каждая команда, написанная на ЯМП, занимает в памяти Простотрона одно слово (и, следовательно, команды имеют знаковые четырехзначные десятичные номера). Мы будем считать, что знак команды ЯМП всегда плюс, но знак слова данных может быть плюсом или минусом. Каждая ячейка в памяти Простотрона может содержать либо команду, либо значение данных, используемых программой, либо соответствовать неиспользуемой (и, следовательно, неопределенной) области памяти. Первые две цифры каждой команды ЯМП служат кодом операции, который указывает, какая операция выполняется. Коды операций ЯМП приведены на рис. 5.37.

Последние две цифры в команде ЯМП являются *операндом* и представляют собой адрес ячейки памяти, содержащей слово, к которому относится операция. Теперь рассмотрим несколько простых программ на ЯМП.

Первая программа на ЯМП (Пример 1) считывает два числа с клавиатуры, складывает их и печатает их сумму. Команда +1007 считывает первое число с клавиатуры и помещает его в ячейку 07 (которая имеет нулевое начальное значение). Затем команда +1008 считывает следующее число в ячейку 08. Команда загрузки +2007 помещает первое число в аккумулятор, а команда сложения +3008 прибавляет второе число к числу в аккумуляторе. *Все арифметические команды ЯМП оставляют результаты в аккумуляторе.* Команда сохранения +2109 помещает результат обратно в память в ячейку 09, из которой команда печати +1109 берет число и печатает его (как знаковое четырехзначное десятичное число). Команда останова +4300 завершает выполнение.

| Код операции                         | Значение                                                                                              |
|--------------------------------------|-------------------------------------------------------------------------------------------------------|
| <i>Операции ввода/вывода:</i>        |                                                                                                       |
| #define READ 10                      | Считать слово с терминала в указанную ячейку памяти                                                   |
| #define WRITE 11                     | Напечатать слово из указанной ячейки памяти на терминале                                              |
| <i>Операции загрузки/хранения:</i>   |                                                                                                       |
| #define LOAD 20                      | Загрузить слово из указанной ячейки памяти в аккумулятор                                              |
| #define STORE 21                     | Сохранить слово из аккумулятора в указанной ячейке памяти                                             |
| <i>Арифметические операции:</i>      |                                                                                                       |
| #define ADD 30                       | Сложить слово из указанной ячейки памяти со словом в аккумуляторе (результат оставить в аккумуляторе) |
| #define SUBTRACT 31                  | Вычесть слово в указанной ячейке памяти из слова в аккумуляторе (результат оставить в аккумуляторе)   |
| #define DIVIDE 32                    | Разделить слово в указанной ячейке памяти на слово в аккумуляторе (результат оставить в аккумуляторе) |
| #define MULTIPLY 33                  | Умножить слово в указанной ячейке памяти на слово в аккумуляторе (результат оставить в аккумуляторе)  |
| <i>Операции передачи управления:</i> |                                                                                                       |
| #define BRANCH 40                    | Передать управление указанной ячейке памяти                                                           |
| #define BRANCHNEG 41                 | Передать управление указанной ячейке памяти, если значение в аккумуляторе отрицательное               |
| #define BRANCHZERO 42                | Передать управление указанной ячейке памяти, если значение в аккумуляторе равно нулю                  |
| #define HALT 43                      | Останов, т.е. программа заканчивает выполнение задания                                                |

Рис. 5.37. Коды операций на языке машины Простотрон (ЯМП)

| Пример1<br>Ячейка | Номер | Команда        |
|-------------------|-------|----------------|
| 00                | +1007 | (Прочитать А)  |
| 01                | +1008 | (Прочитать В)  |
| 02                | +2007 | (Загрузить А)  |
| 03                | +3008 | (Прибавить В)  |
| 04                | +2109 | (Сохранить С)  |
| 05                | +1109 | (Печать С)     |
| 06                | +4300 | (Останов)      |
| 07                | +0000 | (Переменная А) |
| 08                | +0000 | (Переменная В) |
| 09                | +0000 | (Резульят С)   |

| Пример2<br>Ячейка | Номер | Команда                                                 |
|-------------------|-------|---------------------------------------------------------|
| 00                | +1009 | (Прочитать А)                                           |
| 01                | +1010 | (Прочитать В)                                           |
| 02                | +2009 | (Загрузить А)                                           |
| 03                | +3110 | (Вычесть В)                                             |
| 04                | +4107 | (Передать управление в 07 при отрицательном результате) |
| 05                | +1109 | (Печать А)                                              |
| 06                | +4300 | (Останов)                                               |
| 07                | +1110 | (Печать В)                                              |
| 08                | +4300 | (Останов)                                               |
| 09                | +0000 | (Переменная А)                                          |
| 10                | +0000 | (Переменная В)                                          |

Эта программа на ЯМП считывает два числа с клавиатуры, определяет и печатает большее значение. Заметим, что использование команды **+4107** как условной передачи управления очень похоже на оператор **if** в C++. Теперь напишите программы на ЯМП, которые выполняют следующие задачи:

- Используйте цикл, управляемый меткой, для чтения 10 положительных чисел, вычисления их суммы и печати ее.
- Используйте цикл, управляемый счетчиком, для чтения семи чисел, часть из которых положительные, а часть — отрицательные, вычисления их среднего значения и печати его.
- Прочтите последовательность чисел, определите и напечатайте наибольшее из них. Первое прочитанное число указывает количество чисел, которое должно быть обработано.

**5.19. (Программа, моделирующая компьютер)** Может быть это покажется на первый взгляд нахальством, но в этой задаче вы должны будете построить свой собственный компьютер. Нет, вы не будете паять его из отдельных компонентов. Вы будете использовать мощную технику *программного моделирования* для создания *программной модели* Простотрана. Вы не будете разочарованы. Ваша программа, моделирующая Простотрон, будет играть роль этого компьютера и вы на самом деле сможете выполнять, проверять и отлаживать на нем программы на ЯМП, которые вы написали в упражнении 5.18.

Когда вы запустите свою программу, моделирующую Простотрон, она должна начать с печати:

\*\*\* Добро пожаловать в Простотрон! \*\*\*  
 \*\*\* Пожалуйста, вводите по одной команде (или слову \*\*\*  
 \*\*\* данных) вашей программы. Я буду печатать номера \*\*\*  
 \*\*\* ячеек и знак вопроса (?). После этого Вы можете \*\*\*  
 \*\*\* ввести слово в эту ячейку. Введите метку -99999 \*\*\*  
 \*\*\* в конце ввода программы. \*\*\*

Моделируйте память Простотроном одномерным массивом `memory`, содержащим 100 элементов. Теперь предположите, что программа работает, и давайте рассмотрим пример диалога, соответствующий программы примера 2 из упражнения 5.18:

```
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 закончена ***
*** Начинается выполнение программы ***
```

Программа на ЯМП размещена (загружена) в массив `memory`. Теперь Простотрон выполняет вашу программу. Выполнение начинается с команды в ячейке 00 и, как в C++, продолжается последовательно, пока управление не будет передано другой части программы операцией передачи управления.

Используйте переменную `accumulator` для представления регистра аккумулятора. Используйте переменную `instructionCounter` (счетчик команд), чтобы отслеживать ячейку памяти, которая содержит выполняемую команду. Используйте переменную `operationCode` (код операции), содержащую текущую выполняемую команду, т.е. две левые цифры в слове команды. Используйте переменную `operand` (операнд), содержащую номер ячейки памяти, с которой оперирует текущая команда. Таким образом, `operand` — это две крайние правые цифры выполняемой в данный момент команды. Не выполняйте команды непосредственно из памяти. Лучше передайте следующую команду, которая должна выполняться, из памяти в переменную по имени `instructionRegister` (регистр команд). Затем отделите две цифры слева и поместите их в `operationCode`, отделите две цифры справа и поместите их в `operand`.

Когда Простотрон начнет выполнение, отдельные регистры содержат следующие начальные значения:

|                                  |       |
|----------------------------------|-------|
| <code>accumulator</code>         | +0000 |
| <code>instructionCounter</code>  | 00    |
| <code>instructionRegister</code> | +0000 |
| <code>operationCode</code>       | 00    |
| <code>operand</code>             | 00    |

Теперь рассмотрим в деталях выполнение первой команды ЯМП — `+1009` расположенной в ячейке памяти 00. Это называется *циклом выполнения команды*.

`InstructionCounter` сообщает нам ячейку следующей команды, которая будет выполняться. Мы считываем содержимое этой ячейки из `memory`, используя оператор C++

```
instructionRegister = memory[instructionCounter];
```

Код операции и operand извлекаются из регистра команд с помощью операторов

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Теперь Простотрон должен определить, какой код операции *прочитан* (говорят также *записан*, *загружен* и т.д.). Это делает структура **switch**, различающая 12 операций ЯМП.

В этой структуре **switch** моделируется выполнение команд ЯМП следующим образом (остальные команды мы оставляем читателю для самостоятельной разработки):

|                                                                           |                                                                                    |
|---------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <b>чтение:</b>                                                            | <code>cin &gt;&gt; memory[operand];</code>                                         |
| <b>загрузка:</b>                                                          | <code>accumulator = memory[operand];</code>                                        |
| <b>сложение:</b>                                                          | <code>accumulator += memory[operand];</code>                                       |
| <b>Различные команды передачи управления:</b> их мы обсудим кратко позже. |                                                                                    |
| <b>останов:</b>                                                           | <code>Эта команда печатает сообщение<br/>*** Простотрон закончил работу ***</code> |

После выполнения команды программа печатает ее имя, содержимое каждого регистра и полное содержимое памяти. Такой вывод на печать часто называют *дамп компьютера* (распечатка содержимого памяти). Чтобы помочь вам в создании соответствующей функции, на рис. 5.38 показан пример формата дампа. Заметим, что дамп после выполнения программы Простотроном должен показывать истинные значения команд и данных на момент окончания выполнения программы.

Продолжим рассмотрение выполнения первой команды нашей программы, а именно **+1009** в ячейке **00**. Как мы указывали, оператор **switch** моделирует это выполнением оператора C++

```
cin >> memory[operand];
```

Знак вопроса (?) должен отображаться на экране до того, как будет выполняться `cin`, в качестве приглашения пользователя к вводу. Простотрон ждет, когда пользователь напечатает значение и нажмет клавишу ввода *Enter*. После этого значение считывается в ячейку **09**.

На этом моделирование первой команды заканчивается. Все, что остается сделать, — это подготовить Простотрон к выполнению следующей команды. Поскольку выполненная команда не передавала управление, нам нужно просто увеличить регистр счетчика команд следующим образом:

```
++instructionCounter;
```

Это завершает моделирование выполнения первой команды. Весь процесс (т.е. цикл выполнения команды) начинается заново со считывания следующей выполняемой команды.

Теперь рассмотрим, как моделируются команды передачи управления. Все, что нам нужно сделать, — это установить соответствующее значение в счетчике команд. Поэтому команда безусловной передачи управления (**40**) моделируется внутри **switch** следующим образом:

```
instructionCounter = operand;
```

Команда условной передачи управления «передать управление, если аккумулятор равен 0» моделируется следующим образом:

```
if (accumulator == 0)
 instructionCounter = operand;
```

Теперь вы должны реализовать вашу программу моделирования Простотрона и запустить на выполнение все программы на ЯМП, написанные вами в упражнении 5.18. Вы можете украсить ЯМП дополнительными возможностями и реализовать их в вашей программе моделирования.

#### РЕГИСТРЫ

|                     |       |
|---------------------|-------|
| 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. 5.38. Пример распечатки содержимого памяти

Ваша моделирующая программа должна проверять различные типы ошибок. На этапе загрузки, например, каждое число, которое пользователь вводит в массив *memory* вашего Простотрона, должно находиться в диапазоне от -9999 до +9999. Ваша программа должна использовать цикл *while* для проверки, находится ли каждое вводимое число в этом диапазоне, и если нет, предлагать пользователю повторить ввод, до тех пор, пока не будет введено правильное число.

На этапе выполнения ваша программа должна проверять различные серьезные ошибки, такие как попытки деления на 0, попытки выполнения неправильных кодов операции, переполнение аккумулятора (т.е. когда результат арифметических операций больше чем +9999 или меньше чем -9999) и тому подобное. Такие серьезные ошибки называются *неисправимыми ошибками*. Если обнаружена неисправимая ошибка, ваша программа должна печатать сообщение об ошибке, например, такое:

\*\*\* Попытка деления на 0 \*\*\*

\*\*\* Выполнение Простотрона завершено ненормально \*\*\*

и должна напечатать полный дамп в формате, который мы уже ранее обсуждали. Это поможет пользователю локализовать место ошибки в программе.

## Дополнительные упражнения на указатели

**5.20.** Модифицируйте программу тасования и раздачи карт на рис. 5.24 так, чтобы операции тасования и раздачи выполнялись одной и той же функцией (*shuffleAndDeal*). Функция должна содержать вложенную структуру цикла, которая аналогична функции *shuffle* на рис. 5.24.

**5.21.** Что делает эта программа?

```
#include <iostream.h>

void mystery1(char *, const char *);

main()
{
 char string1[80], string2[80];

 cout << "Введите две строки: ";
 cin >> string1 >> string2;
 mystery1(string1, string2);
 cout << string1 << endl;
 return 0;
}
void mystery1(char *s1, const char *s2)
{
 while (*s1 != '\0')
 ++s1;

 for (; *s1 = *s2; s1++, s2++)
 ;
 // пустой оператор
}
```

**5.22.** Что делает эта программа?

```
#include <iostream.h>
int mystery2(const char *);
main()
{
 char string[80];

 cout << "Введите строку: ";
 cin >> string;
 cout << mystery2(string) << endl;
 return 0;
}

int mystery2(const char *s)
{
 for (int x = 0; *s != '\0'; s++)
 ++x;

 return x;
}
```

**5.23.** Найдите ошибку в каждом из следующих фрагментов программ. Если ошибка может быть исправлена, объясните как.

- a) int \*number;  
cout << number << endl;
- b) float \*realPtr;  
long \*integerPtr;  
integerPtr = realPtr;
- c) int \*x, y;  
x = y;
- d) char s[ ] = "это массив символов";  
for ( ; \*s != '\0'; s++)  
 cout << \*s << ' ';
- e) short \*numPtr, result;  
void \*genericPtr = numPtr;  
result = \*genericPtr + 7;
- f) float x = 19.34;  
float xPtr = &x;  
cout << xPtr << endl;
- g) char \*s;  
cout << s << endl;

**5.24. (Быстрая сортировка)** В примерах и упражнениях главы 4 мы обсуждали технику различных видов сортировок — пузырьковой, блочной, выборочной. Теперь рассмотрим технику рекурсивной сортировки, называемой «быстрая сортировка». Основной алгоритм для одномерного массива значений выглядит следующим образом:

- 1) *Шаг декомпозиции*: возьмите первый элемент несортированного массива и определите его окончательную позицию в сортированном массиве. Эта позиция соответствует тому, что все значения слева от элемента в массиве меньше, чем элемент, а все значения справа от элемента в массиве больше, чем элемент. Теперь мы имеем один элемент в соответствующей ему позиции и два несортированных подмассива.
- 2) *Шаг рекурсии* : выполняем шаг 1 для каждого из несортированных подмассивов.

Каждый раз после применения шага 1 к подмассиву еще один элемент помещается в свою окончательную позицию в сортированном массиве и появляются два несортированных массива. Когда подмассив содержит только один элемент, он отсортирован, поэтому этот элемент находится в своей окончательной позиции.

Основной алгоритм представляется достаточно простым, но как определить окончательную позицию первого элемента каждого подмассива? В качестве примера рассмотрим следующее множество значений (выделенный жирным шрифтом элемент — это разделяющий элемент — его мы помещаем в окончательную позицию в сортированном массиве):

37 2 6 4 89 8 10 12 68 45

1) Начиная с самого правого элемента массива, сравниваем каждый элемент с 37 до тех пор, пока не будет найден элемент, меньший, чем 37; тогда переставляем этот элемент и 37. Первый элемент, меньший, чем 37, — число 12, так что переставляем местами 12 и 37. Получаем новый массив:

12 2 6 4 89 8 10 37 68 45

Элемент **12** выделен курсивом, чтобы показать, что он переставлен с **37**.

2) Начиная с левого конца массива, но со следующего после элемента **12**, сравниваем каждый элемент с **37** до тех пор, пока не будет найден элемент, больший, чем **37**; тогда переставляем **37** и этот элемент. Первый элемент, больший, чем **37**, — число **89**, так что переставляем местами **89** и **37**. Получаем новый массив:

**12 2 6 4 37 8 10 89 68 45**

3) Начиная справа, но с первого элемента перед **89**, сравниваем каждый элемент с **37** до тех пор, пока не будет найден элемент, меньший, чем **37**; тогда переставляем **37** и этот элемент. Первый элемент, меньший, чем **37**, — число **10**, так что переставляем местами **10** и **37**. Получаем новый массив:

**12 2 6 4 10 8 37 89 68 45**

4) Начиная слева, но со следующего после элемента **10**, сравниваем каждый элемент с **37** до тех пор, пока не будет найден элемент, больший, чем **37**; тогда переставляем **37** и этот элемент. Поскольку элементов, больших, чем **37**, не оказалось, то при сравнении **37** с самим собой нам становится ясно, что **37** занимает свою окончательную позицию в отсортированном массиве.

Когда выполнена декомпозиция массива, появляются два несортированных массива. Подмассив со значениями, меньшими, чем **37**, содержит числа **12, 2, 6, 4, 10** и **8**. Подмассив со значениями, большими, чем **37**, содержит числа **89, 68** и **45**. Сортировка продолжается и оба подмассива подвергаются декомпозиции тем же способом, что и исходный массив.

Основываясь на приведенных рассуждениях, напишите рекурсивную функцию **quickSort** для сортировки одномерного массива целых. Функция должна принимать в качестве аргументов массив целых, начальный массив и окончательный массив. Функция **partition** должна вызываться функцией **quickSort** для выполнения шага декомпозиции.

**5.25. (Блуждание по лабиринту)** Приведенная ниже сетка из значков **#** и точек **(.)** является двумерным массивом, представляющим лабиринт.

```
#
. . . #
. . # . # . # # . #
. # . . . # .
. . . . # # . # . .
. # . # . # .
. . # . # . # . # .
. # . # . # .
. # .
. # # .
. # . . .
#
```

В приведенном двумерном массиве **#** представляют собой стены лабиринта, а точки представляют собой возможные пути по лабиринту.

Движение возможно только по тем позициям в массиве, которые содержат точки.

Существует простой алгоритм прохождения лабиринта, который гарантирует нахождение выхода (если он существует). Если выхода нет, вы возвращаетесь опять к исходной позиции. Положите вашу правую руку на стену справа и начните двигаться вперед. Никогда не отрывайте руку от стены. Если лабиринт поворачивает направо, следуйте за стеной направо. В конечном счете вы дойдете до выхода из лабиринта. Могут быть пути короче выбранного вами, но, следуя этому алгоритму, вы гарантированно выйдете из лабиринта.

Напишите рекурсивную функцию `mazeTraverse` для прохождения через лабиринт. Функция должна получать в качестве аргумента двумерный массив символов 12 на 12, представляющий лабиринт, и начальную позицию в лабиринте. Функция `mazeTraverse` пытается найти путь к выходу из лабиринта, она должна помечать символом X каждую клетку этого пути. Функция должна отображать лабиринт после каждого перемещения так, чтобы пользователь мог наблюдать решение задачи.

- 5.26. (*Случайная генерация лабиринта*) Напишите функцию `mazeGenerator`, которая получает в качестве аргумента двумерный массив символов 12 на 12 и создает лабиринт случайным образом. Функция должна также выдавать начальную и конечную позиции в лабиринте. Поработайте с вашей функцией `mazeTraverse` из упражнения 5.25, используя несколько случайно сгенерированных лабиринтов.
- 5.27. (*Лабиринты любого размера*) Обобщите функции `mazeTraverse` и `mazeGenerator` из упражнений 5.25 и 5.26 так, чтобы можно было работать с лабиринтами любой ширины и высоты.

- 5.28. (*Массивы указателей на функцию*) Перепишите программу на рис. 4.23 так, чтобы использовать интерфейс, управляемый меню. Программа должна предлагать пользователю меню из четырех позиций, как показано ниже (эти строки должны отображаться на экране):

Выберите:

- 0 Напечатать массив оценок
- 1 Найти низшую оценку
- 2 Найти высшую оценку
- 3 Напечатать среднее значение по всем тестам для каждого студента
- 4 Завершить программу

Одним из ограничений на применение массивов указателей на функции является то, что все указатели должны иметь один и тот же тип. Указатели должны быть на функции, имеющие одинаковый возвращаемый тип и тип принимаемых аргументов. По этой причине функции на рис. 4.23 должны быть модифицированы так, чтобы они возвращали одинаковый тип и принимали одинаковые по типу параметры. Модифицируйте функции `minimum` и `maximum`, чтобы они печатали минимальное и максимальное значение и ничего не возвращали. Для позиции 3 модифицируйте функцию `average` на рис. 4.23, чтобы выводить среднее значение для каждого студента (а не указанного студента). Функция `average` ничего не должна воз-

вращать и должна получать те же самые параметры, что и функции `printArray`, `minimum` и `maximum`. Храните указатели на четыре функции в массиве `processGrades` и используйте выбор, сделанный пользователем, в качестве индексов массива для вызова каждой функции.

**5.29. (Модификации программы моделирования Простотрона)** В упражнении 5.19 вы написали моделирующую программу компьютера, который выполняет программы, написанные на Языке Машины Простотрон (ЯМП). В данном упражнении мы предлагаем несколько модификаций и усовершенствований программы моделирования Простотрона. Мы предлагаем построить компилятор, который преобразует программы, написанные на языке высокого уровня (варианте языка Бейсик) в программы на языке ЯМП. Некоторые из следующих модификаций и усовершенствований могут потребоваться для выполнения программ, создаваемых компилятором.

- Расширьте память модели Простотрона до 1000 ячеек, чтобы Простотрон был в состоянии обрабатывать большие программы.
- Предоставьте возможность модели Простотрона выполнять возведение в степень. Это потребует новых команд ЯМП.
- Предоставьте возможность модели Простотрона выполнять вычисления экспонент. Это потребует новых команд ЯМП.
- Модифицируйте моделирующую программу так, чтобы она могла использовать для представления команд ЯМП вместо целых значений шестнадцатиричные.
- Модифицируйте моделирующую программу так, чтобы позволить ей выводить символ перевода строки. Это потребует новых команд ЯМП.
- Модифицируйте моделирующую программу так, чтобы она могла работать со значениями с плавающей запятой в дополнение к работе с целыми значениями.
- Модифицируйте имитатор так, чтобы он мог обрабатывать ввод строки. *Подсказка:* каждое слово Простотрона можно разделить на две части, каждая из которых содержит двухразрядное целое. Каждое двухразрядное целое представляет десятичный эквивалент ASCII символа. Добавьте команду машинного языка, которая будет вводить строку и хранить ее, начиная с указанной ячейки памяти Простотрона. Первая половина слова в этой ячейке будет равна числу символов в строке (т.е. длине строки). Каждое последующее полуслово содержит один ASCII символ, выражаемый двумя десятичными цифрами. Команда машинного языка преобразует каждый символ в его эквивалент ASCII и присваивает его полуслову.
- Модифицируйте моделирующую программу так, чтобы она могла обрабатывать вывод строки, хранимой в формате пункта g). *Подсказка:* добавьте команду машинного языка, которая будет печатать строку, начиная с указанной ячейки памяти Простотрона. Первая половина слова в этой ячейке — число символов в строке (т.е. длина строки). Каждое последующее полуслово содержит один символ ASCII, выражаемый двумя десятичными цифрами. Команда машинного языка проверяет длину и печатает строку, переводя каждое двухразрядное число в эквивалентный ему символ.

**5.30. Что делает эта программа?**

```
#include <iostream.h>

int mystery3(const char *, const char *);

main()
{
 char string1[80], string2[80];

 cout << " Введите две строки: ";
 cin >> string1 >> string2;
 cout << " Результат равен: "
 << mystery3(string1, string2) << endl;
 return 0;
}

int mystery3(const char *s1, const char *s2)
{
 for (; *s1 != '\0' && *s2 != '\0'; s1++, s2++)
 if (*s1 != *s2)
 return 0;
 return 1;
}
```

**Упражнения на работу со строками**

- 5.31.** Напишите программу, которая использует функцию `strcmp` для сравнения двух строк, вводимых пользователем. Программа должна определять, первая строка меньше, равна или больше второй строки.
- 5.32.** Напишите программу, которая использует функцию `strcmp` для сравнения двух строк, вводимых пользователем. Программа должна вводить количество сравниваемых символов. Программа должна определять, первая строка меньше, равна или больше второй строки.
- 5.33.** Напишите программу, которая использует генерацию случайных чисел для создания предложений. Программа должна использовать четыре массива указателей на `char`, называемые `article` (артикль), `noun` (существительные), `verb` (глаголы) и `preposition` (предлоги). Программа должна создавать предложение, случайно выбирая слова из каждого массива в следующем порядке: `article`, `noun`, `verb`, `preposition`, `article` и `noun`. Как только слово выбрано, оно должно быть подсоединенено к предыдущему слову в массиве, который достаточно велик для того, чтобы вместить все предложение. Слова должны быть разделены пробелами. При выводе окончательного предложения оно должно начинаться с заглавной буквы и заканчиваться точкой. Программа должна генерировать 20 таких предложений. (Замечание: к сожалению, для русского языка подобную программу сделать сложнее из-за падежных окончаний имен существительных.)
- Массивы должны быть заполнены следующим образом: массив `article` должен содержать артикли «`the`», «`a`», «`one`» «`some`» и «`any`»; массив `noun` должен содержать существительные: «`boy`», «`girl`»,

«dog», «town» и «car»; массив `verb` должен содержать глаголы: «drove», «jumped», «ran», «walked» и «skipped»; массив `preposition` должен содержать предлоги «to», «from», «over», «under» и «on».

После того, как вы написали предыдущую программу и она заработала, модифицируйте ее, чтобы она могла создавать короткие рассказы, состоящие из нескольких таких предложений.

- 5.34.** (*Лимерики*) Лимерик — это шуточное пятистрочное стихотворение, в котором первая и вторая строчки rhymeются с пятой, а третья rhymeется с четвертой. Используя технику, подобную разработанной в упражнении 5.33, напишите программу на C++, которая создает случайным способом лимерики. Совершенствование этой программы с целью создания хороших лимериков — головоломная задача, но результат стоит усилий!

- 5.35.** Напишите программу, которая кодирует фразы английского языка псевдолатынью. Псевдолатынь — это форма кодированного языка, часто используемая для развлечения. Существует много вариантов формирования фраз на псевдолатыни. Для простоты используйте следующий алгоритм:

Чтобы сформировать фразу на псевдолатыни из фразы на английском языке, разбейте фразу на слова с помощью функции `strtok`. Для перевода каждого английского слова на язык псевдолатыни поместите первую букву английского слова в конец английского слова и прибавьте буквы «ay». Таким образом слово «jump» превратится в «jmpay», слово «the» превратится в «hetay», а слово «computer» превратится в «omputercay». Пропуски между словами сохраняются. Примем следующие допущения: английская фраза состоит из слов, разделенных пробелами, пунктуация отсутствует, все слова состоят не менее чем из двух букв. Функция `printLatinWord` должна отображать каждое слово. Подсказка: каждая найденная с помощью функции `strtok` лексема должна передавать указатель на лексему функцию `printLatinWord`, которая печатает слово на псевдолатыни.

- 5.36.** Напишите программу, которая вводит в виде строки телефонный номер в форме (555) 555-5555. Программа должна использовать функцию `strtok` для извлечения в виде лексем кода места (указывается в скобках), первых трех цифр и последних четырех цифр телефонного номера. Семь цифр телефонного номера должны соединяться друг с другом в одну строку. Программа должна преобразовывать в `int` строку кода места, и в `long` — строку номера телефона. И код места, и номер телефона должны быть напечатаны.

- 5.37.** Напишите программу, которая вводит строку текста, разбивает ее на лексемы с помощью функции `strtok` и выводит лексемы в обратном порядке.

- 5.38.** Используйте функцию сравнения строк, обсужденную в разделе 5.12.2, и технику сортировки массивов, разработанную в главе 4, для написания программы, которая располагает список строк в алфавитном порядке. Используйте названия 10 или 15 городов в вашей местности в качестве данных для вашей программы.

- 5.39. Напишите по два варианта функций копирования строк и объединения строк, повторяющих функции, приведенные на рис. 5.29. Первый вариант должен использовать индексацию массива, а второй вариант должен использовать указатели и арифметические операции с ними.
- 5.40. Напишите два варианта функций сравнения строк, повторяющих функции, приведенные на рис. 5.29. Первый вариант должен использовать индексацию массива, а второй вариант должен использовать указатели и арифметические операции с ними.
- 5.41. Напишите два варианта функции `strlen`. Первый вариант должен использовать индексацию массива, а второй вариант должен использовать указатели и арифметические операции с ними.

### Специальный раздел: упражнения повышенной сложности на работу со строками

Предыдущие упражнения были базовыми и предназначались для проверки понимания читателем основных принципов работы со строками. Данный раздел включает группу упражнений средней и повышенной сложности на работу со строками. Читатель получит удовольствие от головоломности этих задач. Задачи весьма трудны. Некоторые требуют для своей реализации одного — двух часов работы. Другие полезны для лабораторных занятий и могут требовать двух или трех недель. Некоторые находятся на уровне проектов.

- 5.42. (*Анализ текстов*) Способность компьютеров обрабатывать строки позволяет реализовать некоторые довольно интересные подходы к анализу произведений великих писателей. Много внимания было уделено в свое время вопросу о том, существовал ли когда-либо Вильям Шекспир. Некоторые филологи полагают, что имеются существенные обстоятельства, указывающие на то, что шедевры, приписываемые Шекспиру, написали на самом деле Кристофер Марло или другие авторы. Исследователи использовали компьютеры для поиска сходства в творениях этих двух авторов. В данном упражнении рассматриваются три способа компьютерного анализа текстов.

- a) Напишите программу, которая считывает с клавиатуры несколько строк текста и печатает таблицу, показывающую, сколько раз в тексте встречается каждая буква алфавита. Например, фраза  
`To be, or not to be: that is the question;`  
содержит одну букву «а», две «б», ни одной «с» и т.д.
- b) Напишите программу, которая читает несколько строк текста и печатает таблицу, показывающую, сколько раз в тексте встречаются однобуквенные слова, двухбуквенные слова, трехбуквенные слова и т.д. Например, фраза  
`Whether 'tis nobler in the mind to suffer`  
содержит

| Длина слова | Сколько раз встречается |
|-------------|-------------------------|
| 1           | 0                       |
| 2           | 2                       |
| 3           | 2                       |
| 4           | 2 (включая 'tis)        |
| 5           | 0                       |
| 6           | 2                       |
| 7           | 1                       |

с) Напишите программу, которая считывает несколько строк текста и печатает таблицу, показывающую, сколько раз в тексте встречаются одинаковые слова. Первый вариант вашей программы должен включать слова в таблицу в том порядке, в котором они встречаются в тексте. Например, строки

To be, or not to be: that is the question;  
Whether 'tis nobler in the mind to suffer

содержат слово «to» три раза, слово «be» два раза, слово «or» один раз и т.д. Более интересна и полезна распечатка, в которой слова будут рассортированы в алфавитном порядке.

**5.43. (Обработка слов)** Важной функцией текстовых редакторов является *выравнивание текста* — выравнивание слов по левому и правому полю страницы. Это создает профессионально выглядящий документ, вид которого более похож на выполненный в типографии, чем подготовленный на пишущей машинке. Выравнивание текста можно выполнить на компьютерных системах, вставляя пробелы между словами в строке так, чтобы крайнее правое слово совпадало с правым полем.

Напишите программу, которая считывает несколько строк текста и печатает этот текст с выровненными полями. Будем считать, что текст напечатан на бумаге шириной 8.5 дюймов и что с правой и левой стороны напечатанной страницы допускаются поля шириной в 1 дюйм. Предполагайте, что компьютер печатает 10 символов на одном дюйме. Следовательно, ваша программа должна печатать 6.5 дюймов текста или 65 символов на строку.

**5.44. (Печать дат в различных форматах)** Обычно даты в деловой корреспонденции печатаются в нескольких различных форматах. Наиболее типичными из них являются два:

07/21/55 и Июль 21, 1955

Напишите программу, которая считывает даты в первом формате и печатает их во втором формате.

**5.45. (Защита чеков)** Компьютеры часто используются в системах выписывания чеков, платежных ведомостей, счетов на оплату. Циркулирует множество странных историй, связанных с ошибочной печатью платежных чеков на суммы свыше 1 миллиона долларов. Роковые счета печатаются компьютерными системами документи-

рования чеков из-за человеческих ошибок или машинных сбоев. Разработчики систем используют встроенный контроль для предотвращения выпуска таких ошибочных чеков.

Другой серьезной проблемой является интенсивная подделка чеков теми, кто намеревается мошеннически получить по ним деньги. Чтобы защитить написанные суммы счета от подделки, большинство компьютерных систем документирования чеков используют технику, называемую *защита чеков*.

Чеки, спроектированные для печати компьютером, содержат фиксированное количество пробелов, в которых компьютер может напечатать сумму. Предположим, что платежный чек содержит восемь пробелов, в которых компьютер должен напечатать сумму недельной оплаты. Если сумма большая, заполняются все восемь пробелов, например:

1,230.60 (сумма чека)  
-----  
12345678 (номера позиций)

С другой стороны, если сумма меньше чем 1000 долларов, то несколько пробелов с левой стороны бланка останутся незаполненными. Например,

99.87  
-----  
12345678

содержит три пустых пробела. Если чек напечатан с пустыми пробелами, сумму чека легко подделать. Чтобы защитить чек от подделки, многие системы документирования чеков вставляют начальные звездочки для защиты счета, как например:

\*\*\*99.87  
-----  
12345678

Напишите программу, которая вводит сумму в долларах, которая должна быть напечатана в чеке, и затем печатает ее в защищенном формате с начальными звездочками. Примите, что для печати суммы выделяется девять пробелов.

**5.46. (Написание словесного эквивалента суммы чека)** Продолжая обсуждение предыдущего примера, мы еще раз отметим важность проектирования систем документирования чеков, предотвращающих подделки чековых сумм. Один из наиболее типовых способов защиты требует, чтобы чековая сумма была написана как цифрами, так и словами. Даже если кому-либо удастся подделать сумму в цифрах, чрезвычайно сложно изменить сумму, выраженную словами.

Многие компьютерные системы документирования чеков не печатают сумму чека словами. Возможно, главной причиной этого недостатка является тот факт, что большинство языков высокого уровня, используемых в коммерческих приложениях, не содержат адекватных средств работы со строками. Другой причиной является то, что логика для написания словесных эквивалентов чековых сумм довольно запутана.

Напишите программу на C++, которая вводит сумму в виде числа и печатает ее словесный эквивалент. Например, сумма 112.43 должна быть напечатана так:  
сто двенадцать и 43/100

**5.47. (Азбука Морзе)** Возможно, наиболее известной из всех систем кодирования является азбука Морзе, разработанная Самуэлем Морзе в 1832 году для использования в телеграфе. Азбука Морзе обозначает каждую букву алфавита, каждую цифру и несколько специальных символов (таких как точка, запятая, двоеточие и точка с запятой) последовательностью точек и тире. В звукоориентированных системах точка представляется коротким звуком, а тире — длинным. В светоориентированных системах и системах сигнальных флагов используются другие представления.

Промежутки между словами указываются пробелами, или совсем просто — отсутствием точки или тире. В звукоориентированных системах пробел представляется коротким периодом времени, в течение которого звук отсутствует. Международная версия азбуки Морзе приведена на рис. 5.39.

Напишите программу, которая считывает фразы на английском языке и кодирует их азбукой Морзе. Напишите также программу, которая считывает фразы на языке азбуки Морзе и переводит их в на английский язык. Используйте один пробел между буквами, закодированными азбукой Морзе, и три пробела между закодированными словами.

| Символ | Код     | Символ | Код     | Символ | Код    |
|--------|---------|--------|---------|--------|--------|
| A      | . -     | N      | - .     | Цифры  |        |
| B      | - ...   | O      | ---     | 1      | .----  |
| C      | - -. .  | P      | . -- .  | 2      | ..---- |
| D      | - ..    | Q      | --- . - | 3      | ....-  |
| E      | .       | R      | . - .   | 4      | .....- |
| F      | ... - . | S      | ...     | 5      | .....  |
| G      | -- .    | T      | -       | 6      | -..... |
| H      | ....    | U      | . - -   | 7      | ---... |
| I      | ..      | V      | ... - - | 8      | -----. |
| J      | --- -   | W      | . --    | 9      | -----. |
| K      | - . -   | X      | - . . - | 0      | -----  |
| L      | - . . . | Y      | - . - - |        |        |
| M      | --      | Z      | -- . .  |        |        |

**Рис. 5.39.** Буквы алфавита, выраженные международной азбукой Морзе

**5.48. (Программа преобразования метрической системы мер)** Напишите программу, которая будет помогать пользователю преобразовывать метрическую систему мер. Ваша программа должна позволять пользователю указывать названия единиц измерения в виде строк (например, сантиметры, литры, граммы и т.д. для метрической системы и дюймы, кварты, фунты и т.д. для английской системы мер) и должна отвечать на такие простые вопросы, как

«Сколько дюймов в 2 метрах?»  
«Сколько литров в 10 квартах?»

Ваша программа должна распознавать неправильные преобразования. Например, вопрос

«Сколько футов в 5 килограммах?»

бессмысленен, поскольку «фут» — это мера длины, тогда как «килограмм» — мера веса.

### Головоломный проект работы со строками

**5.49. (Генератор кроссвордов)** Большинство людей имело дело с кроссвордами, но немногие пытались когда-либо придумывать их сами. Генерация кроссвордов — сложная проблема. Она предлагается здесь в виде проекта по работе со строками, требующего существенной изощренности и усилий. Существует много спорных вопросов, которые программист должен разрешить, чтобы получить работающую программу генерации даже простейших кроссвордов. Например, как представить сетку кроссворда в компьютере. Использовать последовательность строк, или двумерный массив? Программисту нужен источник слов (т.е. компьютерный словарь), который можно было бы вызывать непосредственно из программы. В какой форме хранить слова, чтобы облегчить требуемые программой сложные манипуляции со строками? По настоящему честолюбивый читатель захочет генерировать и ту часть головоломки, в которой печатаются краткие подсказки для «горизонтальных» и «вертикальных» слов. Даже просто печать незаполненного варианта кроссворда сама по себе не простая проблема.

г л а в а

---

# 6

## Классы и абстрагирование данных



### Ц е л и

- Понять принципы инкапсуляции и скрытия данных при разработке программного обеспечения.
- Понять идеи абстракции данных и абстрактных типов данных (АТД).
- Научиться создавать АТД С++, а именно, классы.
- Понять, как создаются, используются и разрушаются объекты классов.
- Научиться управлять доступом к данным-элементам и функциям-элементам.
- Начать ценить значение объектной ориентации.

## План

- 6.1. Введение
- 6.2. Определения структур
- 6.3. Доступ к элементам структуры
- 6.4. Использование определенного пользователем типа *Time* с помощью *Struct*
- 6.5. Использование абстрактного типа данных *Time* с помощью класса
- 6.6. Область действия класс и доступ к элементам класса
- 6.7. Отделение интерфейса от реализации
- 6.8. Управление доступом к элементам
- 6.9. Функции доступа и обслуживающие функции-утилиты
- 6.10.Инициализация объектов класса: конструкторы
- 6.11. Использование конструкторов с аргументами по умолчанию
- 6.12. Использование деструкторов
- 6.13. Когда вызываются конструкторы и деструкторы
- 6.14. Использование данных-элементов и функций-элементов
- 6.15. Тонкий момент: возвращение ссылки на закрытые данные-элементы
- 6.16. Присваивание побитовым копированием по умолчанию
- 6.17. Повторное использование программного обеспечения
- 6.18. Размышления об объектах: программирование классов для моделирования лифта.

*Резюме* • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

## 6.1. Введение

Теперь мы начнем знакомиться с объектной ориентацией в C++. Почему мы отложили объектно-ориентированное программирование (ООП) на C++ до главы 6? Дело в том, что объекты, которые мы будем строить, будут составлены частично из структурированных фрагментов программ, так что сначала нам нужно было определить основы структурного программирования.

В разделах «Размышления об объектах» в конце глав с 1 по 5 мы познакомились с основными принципами (т.е. с «объектным мышлением») и с терминологией (т.е. с «объектным языком») объектно-ориентированного программирования на C++. В этих специальных разделах мы также обсудили технику *объектно-ориентированного проектирования*: мы проанализировали типичную постановку задачи, которая требовала построения системы (например, модели лифта), определили, какие объекты необходимы для реализации системы, определили, какие атрибуты должны иметь объекты, определили, какими вариантами поведения должны обладать эти объекты, и указали, как объекты должны взаимодействовать друг с другом для достижения глобальной цели системы.

Давайте проведем краткий обзор некоторых ключевых принципов и терминологии объектной ориентации. ООП *инкапсулирует* данные (атрибуты) и функции (варианты поведения) в совокупности, называемые *объектами*; данные и функции объекта тесно связаны друг с другом. Объекты обладают свойством *скрытия информации*. Это значит, что хотя объекты могут знать, как связываться друг с другом посредством хорошо определенного *интерфейса*, им обычно не позволено знать, как реализуются другие объекты — детали реализации спрятаны внутри самих объектов. Несомненно, можно ездить на автомобиле, не зная технических деталей его внутреннего функционирования — трансмиссии, выхлопной трубы и др. Мы увидим, почему скрытие информации так важно для разработки хорошего программного обеспечения.

В С и других *процедурно-ориентированных языках* программирование стремится быть *ориентированным на действия*, тогда как в идеале программирование на C++ *объектно-ориентированное*. В С единицей программирования является функция. В C++ единицей программирования является *класс*, на основе которого в конечном счете создаются объекты.

Программисты на С основное внимание уделяют написанию функций. Группы действий, выполняющие некоторую задачу, объединяются в функции, а функции группируются, чтобы образовать программу. Данные несомненно важны в С, но современная точка зрения состоит в том, что данные существуют в первую очередь для поддержки действий, выполняемых функциями. Глаголы в описании проектируемой системы помогают программисту на С определить множество функций, которые будут совместно работать для реализации системы.

Программисты на C++ основное внимание уделяют созданию своих собственных *определеняемых пользователем типов*, называемых *классами*. Классы — это типы, определяемые программистом. Каждый класс содержит данные и набор функций, манипулирующих с этими данными. Компоненты-данные класса называются *данными-элементами*. Компоненты-функции класса называются *функциями-элементами*. Подобно тому как сущность встроенного типа, такого, как `int`, называется переменной, сущность определяемого пользователем типа (т.е. класса) называется *объектом*. Центром внимания в C++ являются не функции, а объекты. *Имена существительные* в описании проектируемой системы помогают программисту на C++ определить множество классов. Эти классы используются для создания объектов, которые будут совместно работать для реализации системы.

Классы в C++ являются естественным продолжением структуры `struct` в С. Прежде чем рассматривать специфику разработки классов на C++, мы обсудим структуры и построим определенный пользователем тип, основанный

на структуре. Слабости, которые мы выявим в этом подходе, помогут объяснить запись класса.

## 6.2. Определения структур

Структуры — это составные типы данных, построенные с использованием других типов. Рассмотрим следующее определение структуры:

```
struct Time {
 int hour; // 0-23
 int minute; // 0-59
 int second; // 0-59
};
```

Ключевое слово **struct** начинает определение структуры. Идентификатор **Time** — тег (обозначение, имя-этикетка) структуры. Тэг структуры используется при объявлении переменных структур данного типа. В этом примере имя нового типа — **Time**. Имена, объявленные в фигурных скобках описания структуры — это элементы структуры. Элементы одной и той же структуры должны иметь уникальные имена, но две разные структуры могут содержать не конфликтующие элементы с одинаковыми именами. Каждое определение структуры должно заканчиваться точкой с запятой. Приведенное объяснение, как мы вскоре увидим, верно и для классов.

Определение **Time** содержит три элемента типа **int** — **hour**, **minute** и **second** (часы, минуты и секунды). Элементы структуры могут быть любого типа и одна структура может содержать элементы многих разных типов. Структура не может, однако, содержать экземпляры самой себя. Например, элемент типа **Time** не может быть объявлен в определении структуры **Time**. Однако, может быть включен указатель на другую структуру **Time**. Структура, содержащая элемент, который является указателем на такой же структурный тип, называется *структурой с самоадресацией*. Структуры с самоадресацией полезны для формирования связанных структур данных (см. главу 15).

Предыдущее определение структуры данных не резервирует никакого пространства в памяти; определение только создает новый тип данных, который используется для объявления переменных. Переменные структуры объявляются так же, как переменные других типов. Объявление

```
Time timeObject, timeArray[10], *timePtr;
```

объявляет **timeObject** переменной типа **Time**, **timeArray** — массивом с 10 элементами типа **Time**, а **timePtr** — указателем на объект типа **Time**.

## 6.3. Доступ к элементам структуры

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) по имени переменной объекта или по ссылке на объект. Например, чтобы напечатать элемент **hour** структуры **timeObject** используется оператор

```
cout << timeObject.hour;
```

Операция стрелка, состоящая из знака минус (**-**) и знака больше (**>**), записанных без пробела, обеспечивает доступ к элементу структуры (или

класса) через указатель на объект. Допустим, что указатель `timePtr` был уже объявлен как указывающий на объект типа `Time` и что адрес структуры `timeObject` был уже присвоен `timePtr`. Тогда, чтобы напечатать элемент `hour` структуры `timeObject` с указателем `timePtr`, можно использовать оператор

```
cout << timePtr->hour;
```

Выражение `timePtr->hour`; эквивалентно `(*timePtr).hour`, которое разыменовывает указатель и делает доступным элемент `hour` через операцию точка. Скобки нужны здесь потому, что операция точки имеет более высокий приоритет, чем операция разыменования указателя (\*). Операции стрелка и точка наряду с круглыми и квадратными скобками имеют второй наивысший приоритет (после операции разрешения области действия, введенной в главе 3) и ассоциативность слева направо.

## 6.4. Использование определенного пользователем типа `Time` с помощью `Struct`

Программа на рис. 6.1 создает определенный пользователем тип структуры `Time` с тремя целыми элементами: `hour`, `minute` и `second`. Программа определяет единственную структуру типа `Time`, названную `dinnerTime`, и использует операцию точки для присвоения элементам структуры начальных значений 18 для `hour`, 30 для `minute` и 0 для `second`. Затем программа печатает время в военном (24-часовом) и стандартном (12-часовом) форматах. Заметим, что функции печати принимают ссылки на постоянные структуры типа `Time`. Это является причиной того, что структуры `Time` передаются печатающим функциям по ссылке — этим исключаются накладные расходы на копирование, связанные с передачей структур функциям по значению, а использование `const` предотвращает изменение структуры типа `Time` функциями печати. В главе 7 мы обсудим объекты `const` и функции-элементы `const`.

### Совет по повышению эффективности 6.1

Обычно структуры передаются вызовом по значению. Чтобы избежать накладных расходов на копирование структуры, передавайте структуры вызовом по ссылке.

### Совет по повышению эффективности 6.2

Чтобы избежать накладных расходов вызова по значению и вдобавок получить выгоды защиты исходных данных от изменения, передавайте аргументы большого размера как ссылки `const`.

Существуют препятствия созданию новых типов данных указанным способом с помощью структур. Поскольку инициализация структур специально не требуется, можно иметь данные без начальных значений и вытекающие отсюда проблемы. Даже если данные получили начальные значения, возможно, это было сделано неверно. Неправильные значения могут быть присвоены элементам структуры (как мы сделали на рис. 6.1), потому что программа имеет прямой доступ к данным. Программа присвоила плохие значения всем трем элементам объекта `dinnerTime` типа `Time`. Если реализация `struct` изменится (например, время теперь будет представляться как

число секунд после полуночи), то все программы, которые используют `struct`, нужно будет изменить. Не существует никакого «интерфейса», гарантирующего, что программист правильно использует тип данных и что данные являются непротиворечивыми.

### **Замечание по технике программирования 6.1**

Важно писать программы, которые легко понимать и поддерживать. Изменения являются скорее правилом, чем исключением. Программисты должны предвидеть, что их коды будут изменяться. Как мы увидим, классы способствуют модифицируемости программ.

```
// FIG6_1.CPP
// Создание структуры, задание и печать ее элементов.
#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

 // задание элементам правильных значений
 dinnerTime.hour = 18;
 dinnerTime.minute = 30;
 dinnerTime.second = 0;

 cout << "Обед состоится в ";
 printMilitary(dinnerTime);
 cout << " по военному времени," << endl
 << "что соответствует ";
 printStandard(dinnerTime);
 cout << " по стандартному времени." << endl;

 // задание элементам неправильных значений
 dinnerTime.hour = 29;
 dinnerTime.minute = 73;
 dinnerTime.second = 103;

 cout << endl << "Время с неправильными значениями: ";
 printMilitary (dinnerTime);
 cout << endl;
 return 0;
}
```

**Рис. 6.1.** Создание структуры, задание и печать ее элементов (часть 1 из 2)

```

// Печать времени 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;
}

// Печать времени 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");
}

```

Обед состоится в 18:30:00 по военному времени,  
что соответствует 6:30:00 PM по стандартному времени.

Время с неправильными значениями: 29:73:103

**Рис. 6.1.** Создание структуры, задание и печать ее элементов (часть 2 из 2)

Существуют и другие проблемы, связанные со структурами в стиле C. В С структуры не могут быть напечатаны как единое целое, только по одному элементу с соответствующим форматированием каждого. Для печати элементов структуры в каком-либо подходящем формате должна быть написана функция. Глава 8, «Перегрузка операций» покажет, как перегрузить операцию <<, чтобы предоставить возможность простой печати объектов типа структура (C++ расширяет понятие структуры) или типа класс. В С структуры нельзя сравнивать в целом; их нужно сравнивать элемент за элементом. Глава 8 покажет, как перегрузить операции проверки равенства и отношения, чтобы можно было в Си++ сравнивать объекты типов структура и класс.

В следующем разделе мы вновь используем нашу структуру Time, но уже как класс, и продемонстрируем некоторые преимущества создания таких так называемых *абстрактных типов данных*, как классы. Мы увидим, что классы и структуры в C++ можно использовать почти одинаково. Различие между ними состоит в доступности по умолчанию элементов каждого из этих типов. Это будет более детально объяснено позже.

## 6.5. Использование абстрактного типа данных Time с помощью класса

Классы предоставляют программисту возможность моделировать объекты, которые имеют *атрибуты* (представленные как *данные-элементы*) и *варианты поведения* или *операции* (представленные как *функции-элементы*). Типы, содержащие данные-элементы и функции-элементы, обычно определяются в C++ с помощью ключевого слова *class*.

Функции-элементы иногда в других объектно-ориентированных языках называют *методами*, они вызываются в ответ на *сообщения*, посылаемые объекту. Сообщение соответствует вызову функции-элемента.

Когда класс определен, имя класса может быть использовано для объявления объекта этого класса. Рис. 6.2 содержит простое определение класса Time.

Определение нашего класса Time начинается с ключевого слова `class`. Тело определения класса заключается в фигурные скобки (`{ }` ). Определение класса заканчивается точкой с запятой. Определение нашего класса Time, как и нашей структуры Time, содержит три целых элемента `hour`, `minute` и `second`.

### **Типичная ошибка программирования 6.1**

Забывается точка с запятой в конце определения класса (или структуры).

```
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. 6.2.** Простое определение класса Time

Остальные части определения класса — новые. Метки `public:` (открытая) и `private:` (закрытая) называются *спецификаторами доступа к элементам*. Любые данные-элементы и функции-элементы, объявленные после спецификатора доступа к элементам `public:` (и до следующего спецификатора доступа к элементам), доступны при любом обращении программы к объекту класса Time. Любые данные-элементы и функции-элементы, объявленные после спецификатора доступа к элементам `private:` (и до следующего спецификатора доступа к элементам), доступны только функциям-элементам этого класса. Спецификаторы доступа к элементам всегда заканчиваются двоеточием (`:`) и могут появляться в определении класса много раз и в любом порядке. В дальнейшем в тексте нашей книги мы будем использовать записи спецификаторов доступа к элементам в виде `public` и `private` (без двоеточия).

### **Хороший стиль программирования 6.1**

Используйте при определении класса каждый спецификатор доступа к элементам только один раз, что сделает программу более ясной и простой для чтения. Размещайте первыми элементы `public`, являющиеся общедоступными.

Определение класса в нашей программе содержит после спецификатора доступа к элементам `public` прототипы следующих четырех функций-элементов: `Time`, `setTime`, `printMilitary` и `printStandard`. Это — *открытые функции-элементы* или *открытый интерфейс услуг класса*. Эти функции будут

использоваться *клиентами* класса (т.е. частями программы, играющими роль пользователей) для манипуляций с данными этого класса.

Обратите внимание на функцию-элемент с тем же именем, что и класс. Она называется *конструктором* этого класса. Конструктор — это специальная функция-элемент, которая инициализирует данные-элементы объекта этого класса. Конструктор класса вызывается автоматически при создании объекта этого класса. Мы увидим, что обычно класс имеет несколько конструкторов; это достигается посредством перегрузки функции.

После спецификатора доступа к элементам **private** следуют три целых элемента. Это говорит о том, что эти данные-элементы класса являются доступными только функциям-элементам класса и, как мы увидим в следующей главе, «друзьям» класса. Таким образом, данные-элементы могут быть доступны только четырем функциям, прототипы которых включены в определение этого класса (или друзей этого класса). Обычно данные-элементы перечисляются в части **private**, а функции-элементы — в части **public**. Как мы увидим далее, можно иметь функции-элементы **private** и данные **public**; последнее не типично и считается в программировании дурным вкусом.

Когда класс определен, его можно использовать в качестве типа в объявлениях, например, следующим образом:

```
Time sunset, // объект типа Time
arrayOfTimes[5], // массив объектов типа Time
*pointerToTime, // указатель на объект типа Time
&dinnerTime = sunset; // ссылка на объект типа Time
```

Имя класса становится новым спецификатором типа. Может существовать множество объектов класса как и множество переменных типа, например, такого, как **int**. Программист по мере необходимости может создавать новые типы классов. Это одна из многих причин, по которым C++ является расширяемым языком.

Программа на рис. 6.3 использует класс **Time**. Эта программа создает единственный объект класса **Time**, названный **t**. Когда объект создается, автоматически вызывается конструктор **Time**, который явно присваивает нулевые начальные значения всем данным-элементам закрытой части **private**. Затем печатается время в военном и стандартном форматах, чтобы подтвердить, что элементы получили правильные начальные значения. После этого с помощью функции-элемента **setTime** устанавливается время и оно снова печатается в обоих форматах. Затем функция-элемент **setTime** пытается дать данным-элементам неправильные значения и время снова печатается в обоих форматах.

Снова отметим, что данные-элементы **hour**, **minute** и **second** предваряются спецификатором доступа к элементам **private**. Эти закрытые данные-элементы класса обычно недоступны вне класса (но, как мы увидим в главе 7, друзья класса могут иметь доступ к закрытым элементам класса.) Глубокий смысл такого подхода заключается в том, что истинное представление данных внутри класса не касается клиентов класса. Например, было бы вполне возможно изменить внутреннее представление и представлять, например, время внутри класса как число секунд после полуночи. Клиенты могли бы использовать те же самые открытые функции-элементы и получать те же самые результаты, даже не осознавая произведенных изменений. В этом смысле, говорят, что реализация класса скрыта от клиентов. Такое скрытие информации способствует модифицируемости программ и упрощает восприятие класса клиентами.

```

// FIG6_3.CPP
// Класс Time.
#include <iostream.h>

// Определение абстрактного типа данных (АТД) Time
class Time{
public:
 Time(); // конструктор
 void setTime(int, int, int); // установка часов, минут
 // и секунд
 void printMilitary(); // времени в военном формате
 void printStandard(); // печать времени
 // в стандартном формате

private:
 int hour; // 0 - 23
 int minute; // 0 - 59
 int second; // 0 - 59
};

// Конструктор Time присваивает нулевые начальные значения
// каждому элементу данных. Обеспечивает согласованное
// начальное состояние всех объектов Time
Time::Time() { hour = minute = second = 0; }

// Задание нового значения Time в виде военного времени.
// Проверка правильности значений данных.
// Обнуление неверных значений.
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;
}

// Печать времени в военном формате
void Time:: printMilitary ()
{
 cout << (hour < 10 ? "0" "") << hour << ":"
 << (minute < 10 ? "0": "") << minute << ":"
 << (second < 10 ? "0" : "") << second;
}

// Печать времени 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. 6.3.** Использование абстрактного типа данных **Time** как класса (часть 1 из 2)

```
// Формирование проверки простого класса Time
main()
{
 Time t; // определение экземпляра объекта t класса Time
 cout << "Начальное значение военного времени равно ";
 t.printMilitary();
 cout << endl
 << "Начальное значение стандартного времени равно ";
 t.printStandard();

 t.setTime(13, 27, 6);
 cout << endl << endl << "Военное время после setTime равно ";
 t.printMilitary();
 cout << endl << "Стандартное время после setTime равно ";
 t.printStandard();

 t.setTime(99, 99, 99); // попытка установить
 // неправильные значения
 cout << endl << endl
 << "После попытки неправильной установки: "
 << endl << "Военное время: ";
 t.printMilitary();
 cout << endl << "Стандартное время: ";
 t.printStandard();
 cout << endl;
 return 0;
}
```

---

```
Начальное значение военного времени равно 00:00:00
Начальное значение стандартного времени равно 12:00:00 AM

Военное время после setTime равно 13:27:06
Стандартное время после setTime равно 1:27:06 PM

После попытки неправильной установки:
Военное время: 00:00:00
Стандартное время: 12:00:00 AM
```

Рис. 6.3. Использование абстрактного типа данных **Time** как класса (часть 2 из 2)

### Замечание по технике программирования 6.2

Клиенты класса используют класс, не зная внутренних деталей его реализации. Если реализация класса изменяется (например, с целью улучшения производительности), интерфейс класса остается неизменным и исходный код клиента класса не требует изменений. Это значительно упрощает модификацию систем.

В нашей программе конструктор **Time** просто присваивает начальные значения, равные 0, данным-элементам, (т.е. задает военное время, эквивалентное 12 AM). Это гарантирует, что объект при его создании находится в известном состоянии. Неправильные значения не могут храниться в данных-элементах объекта типа **Time**, поскольку конструктор автоматически вызывается при создании объекта типа **Time**, а все последующие попытки изменить данные-элементы тщательно рассматриваются функцией **setTime**.

### **Замечание по технике программирования 6.3**

Функции-элементы обычно короче, чем обычные функции в программах без объектной ориентации, потому что достоверность данных, хранимых в данных-элементах, идеально проверена конструктором и функциями-элементами, которые сохраняют новые данные.

Отметим, что данные-элементы класса не могут получать начальные значения в теле класса, где они объявляются. Эти данные-элементы должны получать начальные значения с помощью конструктора класса или им можно присваивать значения через функции.

### **Типичная ошибка программирования 6.2**

Попытка явно присвоить начальное значение данным-элементам в определении класса.

Функция с тем же именем, что и класс, но со стоящим перед ней символом тильда (~), называется *деструктором* этого класса (наш пример не включает деструктор). Деструктор производит «завершающие служебные действия» над каждым объектом класса перед тем, как память, отведенная под этот объект, будет повторно использована системой. Подробнее мы обсудим конструкторы и деструкторы позже в этой главе и в главе 7.

Заметим, что функции, которыми класс снабжает внешний мир, предваряются меткой `public`. Открытые функции реализуют все возможности класса, необходимые для его клиентов. Открытые функции класса называют *интерфейсом* класса или *открытым интерфейсом*.

### **Замечание по технике программирования 6.4**

Клиенты имеют доступ к интерфейсу класса, но не имеют доступа к реализации класса.

Объявление класса содержит объявления данных-элементов и функций-элементов класса. Объявления функций-элементов являются прототипами функций, которые мы обсуждали в предыдущих главах. Функции-элементы могут быть описаны внутри класса, но хороший стиль программирования заключается в описании функций вне определения класса.

### **Замечание по технике программирования 6.5**

Объявление функций-элементов внутри определения класса и описание функций-элементов вне этого определения отделяет интерфейс класса от его реализации. Это способствует высокому качеству разработки программного обеспечения.

Отметим использование *бинарной операции разрешения области действия* (::) в каждом определении функции-элемента, следующем за определением класса на рис. 6.3. После того, как класс определен и его функции-элементы объявлены, эти функции-элементы должны быть описаны. Каждая функция-элемент может быть описана прямо в теле класса (вместо включение прототипа функции класса) или после тела класса. Когда функция-элемент описывается после соответствующего определения класса, имя функции предваряется именем класса и бинарной операцией разрешения области действ-

вия (::). Поскольку разные классы могут иметь элементы с одинаковыми именами, операция разрешения области действия «привязывает» имя элемента к имени класса, чтобы однозначно идентифицировать функции-элементы данного класса.

Несмотря на то, что функция-элемент, объявленная в определении класса, может быть описана вне этого определения, эта функция-элемент все равно имеет *область действия класс*, т.е. ее имя известно только другим элементам класса пока к ней обращаются посредством объекта класса, ссылки на объект класса или указателя на объект класса. Об области действия класса мы более подробно еще поговорим позднее.

Если функция-элемент описана в определении класса, она автоматически встраивается *inline*. Функция-элемент, описанная вне определения класса, может быть сделана встраиваемой посредством явного использования ключевого слова *inline*. Напомним, что компилятор резервирует за собой право не встраивать никаких функций.

### Совет по повышению эффективности 6.3

Описание небольших функций-элементов внутри определения класса автоматически встраивает функцию-элемент *inline* (если компилятор решит делать это). Это может улучшить производительность, но не способствует улучшению качества проектирования программного обеспечения.

Интересно, что функции-элементы *printMilitary* и *printStandard* не получают никаких аргументов. Это происходит потому, что функции-элементы неявно знают, что они печатают данные-элементы определенного объекта типа *Time*, для которого они активизированы. Это делает вызовы функций-элементов более краткими, чем соответствующие вызовы функций в процедурном программировании. Это уменьшает также вероятность передачи неверных аргументов, неверных типов аргументов или неверного количества аргументов.

### Замечание по технике программирования 6.6

Использование принципов объектно-ориентированного программирования часто может упростить вызовы функции за счет уменьшения числа передаваемых параметров. Это достоинство объектно-ориентированного программирования проистекает из того факта, что инкапсуляция данных-элементов и функций-элементов внутри объекта дает функциям-элементам право прямого доступа к данным-элементам.

Классы упрощают программирование, потому что клиент (или пользователь объекта класса) имеет дело только с операциями, инкапсулированными или встроенными в объект. Такие операции обычно проектируются ориентированными именно на клиента, а не на удобную реализацию. Клиентам нет необходимости касаться реализации класса. Интерфейсы меняются, но не так часто, как реализации. При изменении реализации соответственно должны изменяться ориентированные на реализацию коды. А путем скрытия реализации мы исключаем возможность для других частей программы оказаться зависимыми от особенностей реализации класса.

Часто классы не создаются «на пустом месте». Обычно они являются производными от других классов, обеспечивающих новые классы необходимыми им операциями. Или классы могут включать объекты других классов

как элементы. Такое повторное использование программного обеспечения значительно увеличивает производительность программиста. Создание новых классов на основе уже существующих классов называется *наследованием* и подробно обсуждается в главе 9. Включение классов как элементов других классов называется *композицией* и обсуждается в главе 7.

## 6.6. Область действия класс и доступ к элементам класса

Данные-элементы класса (переменные, объявленные в определении класса) и функции-элементы (функции, объявленные в определении класса) имеют *областью действия класс*. Функции, не являющиеся элементом класса, имеют *областью действия файл*.

При области действия класс элементы класса непосредственно доступны всем функциям-элементам этого класса и на них можно ссылаться просто по имени. Вне области действия класс к элементам класса можно обращаться либо через имя объекта, либо ссылкой на объект, либо с помощью указателя на объект.

Функции-элементы класса можно перегружать, но только с помощью других функций-элементов класса. Для перегрузки функции-элемента просто обеспечьте в определении класса прототип для каждой версии перегруженной функции и снабдите каждую версию функции отдельным описанием.

### Типичная ошибка программирования 6.3

Попытка перегрузить функцию-элемент класса с помощью функции не из области действия этого класса.

Функции-элементы имеют внутри класса *область действия функцию*: переменные, определенные в функции-элементе, известны только этой функции. Если функция-элемент определяет переменную с тем же именем, что и переменная в области действия класс, последняя делается невидимой в области действия функция. Такая скрытая переменная может быть доступна посредством операции разрешения области действия с предшествующим этой операции именем класса. Невидимые глобальные переменные могут быть доступны с помощью унарной операции разрешения области действия (смотри главу 3).

Операции, использованные для доступа к элементам класса, аналогичны операциям, используемым для доступа к элементам структуры. *Операция выбора элемента точка (.)* комбинируется для доступа к элементам объекта с именем объекта или со ссылкой на объект. *Операция выбора элемента стрелка (->)* комбинируется для доступа к элементам объекта с указателем на объект.

Программа на рис. 6.4 использует простой класс, названный *Count*, с открытым элементом данных *x* типа *int* и открытой функцией-элементом *print*, чтобы проиллюстрировать доступ к элементам класса с помощью операций выбора элемента. Программа создает три экземпляра переменных типа *Count* — *counter*, *counterRef* (ссылка на объект типа *Count*) и *counterPtr* (указатель на объект типа *Count*). Переменная *counterRef* объявлена, чтобы

ссылаясь на `counter`, переменная `counterPtr` объявлена, чтобы указывать на `counter`. Важно отметить, что здесь элемент данных `x` сделан открытым просто для того, чтобы продемонстрировать способы доступа к открытым элементам. Как мы уже установили, данные обычно делаются закрытыми (`private`), чему мы и будем следовать в дальнейшем.

```
// FIG6_4.CPP
// Демонстрация операций доступа к элементам класса . и ->
//
// ПРЕДУПРЕЖДЕНИЕ: В ПОСЛЕДУЮЩИХ ПРИМЕРАХ МЫ
// БУДЕМ ИЗБЕГАЕМ ОТКРЫТЫХ ДАННЫХ!
#include <iostream.h>

// Простой класс Count
class Count {
public:
 int x;
 void print() { cout << x << endl; }
};

main ()
{
 Count counter, // создается объект counter
 *counterPtr = &counter , // указатель на counter
 &counterRef = counter; // ссылка на counter

 cout << "Присваивание x значения 7 и печать по имени объекта: ";
 counter.x = 7; // присваивание 7 элементу данных x
 counter.print(); // вызов функции-элемента для печати

 cout << "Присваивание x значения 8 и печать по ссылке: ";
 counterRef.x = 8; // присваивание 8 элементу данных x
 counterRef.print(); // вызов функции-элемента для печати

 cout << "Присваивание x значения 10 и печать по указателю: ";
 counterPtr->x = 10; // присваивание 10 элементу данных x
 counterPtr->print(); // вызов функции-элемента для печати
return 0;
}
```

---

```
Присваивание x значения 7 и печать по имени объекта: 7
Присваивание x значения 8 и печать по ссылке: 8
Присваивание x значения 10 и печать по указателю: 10
```

**Рис. 6.4.** Доступ к данным-элементам объекта и функциям-элементам посредством имени объекта, ссылки и указателя на объект

## 6.7. Отделение интерфейса от реализации

Один из наиболее фундаментальных принципов разработки хорошего программного обеспечения состоит в отделении интерфейса от реализации. Это облегчает модификацию программ. Что касается клиентов класса, то изменения в реализации класса не влияют на клиента до тех пор, пока интерфейс класса, изначально предназначенный для клиента, остается неизменным (функции класса можно расширять за пределы исходного интерфейса).

### **Замечание по технике программирования 6.7**

Помещайте объявление класса в заголовочный файл, чтобы оно было доступно любому клиенту, который захочет использовать класс. Это формирует открытый интерфейс класса. Помещайте определения функций-элементов класса в исходный файл. Это формирует реализацию класса.

### **Замечание по технике программирования 6.8**

Клиенты класса не нуждаются в доступе к исходному коду класса для того, чтобы использовать класс. Однако, клиенты должны иметь возможность связаться с объектным кодом класса.

Это стимулирует независимых продавцов программного обеспечения поставлять библиотеки классов для продажи или лицензирования. Продавцы поставляют в своих продуктах только заголовочные файлы и объектные модули. Не выдается никакой оригинальной, патентоспособной информации, как это было бы в случае поставки исходных кодов. Сообщество пользователей C++ извлекает выгоду, имея в своем распоряжении большинство библиотек классов, поставляемых независимыми продавцами.

На самом деле все выглядит не в таком розовом свете. Заголовочные файлы содержат некоторую часть реализации и краткие сведения о других частях реализации. Встраиваемые функции-элементы, например, должны находиться в заголовочном файле, так что когда компилятор компилирует клиента, клиент может включить определение встраиваемой функции *inline*. Закрытые элементы перечисляются в определении класса в заголовочном файле, так что эти элементы видимы для клиентов, несмотря на то, что клиенты не могут иметь к ним доступа.

### **Замечание по технике программирования 6.9**

Информация, важная для интерфейса класса, должна включаться в заголовочный файл. Информация, которая будет использоваться только внутри класса и которая не является необходимой для клиентов класса, должна включаться в неоглашаемый исходный файл. Это еще один пример принципа наименьших привилегий.

Рисунок 6.5 разбивает программу на рис. 6.3 на ряд файлов. При построении программы на C++ каждое определение класса обычно помещается в заголовочный файл, а определения функций-элементов этого класса помещаются в файлы исходных кодов с теми же базовыми именами. Заголовочные файлы включаются (посредством *#include*) в каждый файл, в котором используется класс, а файлы с исходными кодами компилируются и компонуются с файлом, содержащим главную программу. Посмотрите документацию на ваш компилятор, чтобы определить, как компилировать и компоновать программы, содержащие множество исходных файлов.

Программа на рис. 6.5 состоит из заголовочного файла *time1.h*, в котором объявляется класс *Time*, файла *time1.cpp*, в котором описываются функции-элементы класса *Time*, и файла *fig6\_5.cpp*, в котором описывается функция *main*. Выходные данные этой программы идентичны выходным данным программы на рис. 6.3.

```

// TIME1.H
// Объявление класса Time.
// Функции-элементы определены в TIME.CPP

// Предотвращение многократного включения заголовочного файла
#ifndef TIME1_H
#define TIME1_H

// Определение абстрактного типа данных Time

class Time {
public:
 Time(); // конструктор
 void setTime(int, int, int); // установка часов, минут,
 // секунд
 void printMilitary(); //печать времени в военном формате
 void printStandard(); //печать времени в стандартном формате

private:
 int hour; // 0 - 23
 int minute; // 0 - 59
 int second; // 0 - 59
};

#endif

```

Рис. 6.5. Заголовочный файл класса Time (часть 1 из 3)

```

// TIME1.CPP
// Определения функций-элементов для класса Time.

#include <iostream.h>
#include "time1.h"

// Конструктор Time присваивает нулевые начальные значения каждому
// элементу данных. Обеспечивает согласованное начальное состояние
// всех объектов Time
Time::Time() { hour = minute = second = 0; }

// Задание нового значения Time в виде военного времени.
// Проверка правильности значений данных. Обнуление неверных
// значений.
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;
}

// Печать времени в военном формате
void Time::printMilitary()
{
 cout << (hour < 10 ? "0" : "") << hour << ":"
 << (minute < 10 ? "0" : "") << minute << ":"
 << (second < 10 ? "0" : "") << second;
}

```

```
// Печать времени 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. 6.5.** Исходный файл определений функций-элементов класса Time (часть 2 из 3)

```
// FIG6_5.CPP
// Драйвер класса Time1
// ЗАМЕЧАНИЕ: Компилируется вместе с TIME1.CPP
#include <iostream.h>
#include "time1.h"

// Драйвер для проверки простого класса Time
main()
{
 Time t; // определение экземпляра объекта t класса Time

 cout << "Начальное значение военного времени равно ";
 t.printMilitary();
 cout << endl
 << "Начальное значение стандартного времени равно ";
 t.printStandard();

 t.setTime(13, 27, 6);
 cout << endl << endl << "Военное время после setTime равно ";
 t.printMilitary();
 cout << endl << "Стандартное время после setTime равно ";
 t.printStandard();

 t.setTime(99, 99, 99); // попытка установить
 // неправильные значения
 cout << endl << endl
 << "После попытки неправильной установки: "
 << endl << "Военное время: ";
 t.printMilitary();
 cout << endl << "Стандартное время: ";
 t.printStandard();
 cout << endl;
 return 0;
}
```

---

Начальное значение военного времени равно 00:00:00  
Начальное значение стандартного времени равно 12:00:00 AM

Военное время после setTime равно 13:27:06  
Стандартное время после setTime равно 1:27:06 PM

После попытки неправильной установки:  
Военное время: 00:00:00  
Стандартное время: 12:00:00 AM

**Рис. 6.5.** Программа драйвера класса Time (часть 3 из 3)

Заметим, что объявление класса заключено в следующие директивы препроцессора (смотри главу 17):

```
// Предотвращение многократного включения заголовочного файла
#ifndef TIME1_H
#define TIME1_H
...
#endif
```

При построении больших программ в заголовочные файлы будут помещаться также и другие определения и объявления. Приведенные выше директивы препроцессора предотвращают включение кода между `#ifndef` и `#endif`, если определено имя `TIME1_H`. Если заголовок еще не включался в файл, то имя `TIME1_H` определяется директивой `#define` и операторы заголовочного файла включаются в результирующий файл. Если же заголовок уже был включен ранее, `TIME1_H` уже определен и операторы заголовочного файла повторно не включаются. Попытки многократного включения заголовочного файла обычно случаются в больших программах с множеством заголовочных файлов, которые могут сами включать другие заголовочные файлы. Замечание: по негласному соглашению в приведенных выше директивах используется имя символической константы, представляющее собой просто имя заголовочного файла с символом подчеркивания вместо точки.

### Хороший стиль программирования 6.2

Используйте директивы препроцессора `#ifndef`, `#define` и `#endif` для того, чтобы избежать включения в программу заголовочных файлов более одного раза.

### Хороший стиль программирования 6.3

Используйте имя заголовочного файла с символом подчеркивания вместо точки в директивах препроцессора `#ifndef` и `#define` заголовочного файла.

## 6.8. Управление доступом к элементам

Спецификаторы доступа к элементу `public` и `private` (а также, как мы увидим в главе 9 «Наследование», `protected` — защищенные) используются для управления доступом к данным-элементам класса и функциям-элементам. По умолчанию режим доступа для классов — `private` (закрытый), так что все элементы после заголовка класса и до первого спецификатора доступа являются закрытыми. После каждого спецификатора режим доступа, определенный им, действует до следующего спецификатора или до завершающей правой скобки `()` определения класса. Спецификаторы `private`, `public` и `protected` могут быть повторены, но такое употребление редко и может привести к беспорядку.

Закрытые элементы класса могут быть доступны только для функций-элементов (и дружественных функций) этого класса. Открытые элементы класса могут быть доступны для любых функций в программе.

Основная задача открытых элементов состоит в том, чтобы дать клиентам класса представление о возможностях (*услугах*), которые обеспечивает класс. Этот набор услуг составляет *открытый интерфейс* класса. Клиентов класса не должно касаться, каким образом класс выполняет их задачи. Закрытые

элементы класса и описания открытых функций-элементов недоступны для клиентов класса. Эти компоненты составляют *реализацию* (*implementation*) класса.

### **Замечание по технике программирования 6.10**

C++ способствует созданию программ, не зависящих от реализации. Если, изменяется реализация класса, используемого программой, не зависящей от реализации, то код этой программы не требует изменения, но может потребоваться его перекомпиляция.

### **Типичная ошибка программирования 6.4**

Попытка с помощью функции, не являющейся элементом определенного класса (или другом этого класса) получить доступ к элементам этого класса.

Программа на рис. 6.6 демонстрирует, что закрытые элементы класса доступны только через открытый интерфейса класса, включающий открытые функции-элементы. Во время компиляции этой программы компилятор выдает две ошибки, объявляющие, что закрытые элементы, указанные в каждом операторе, недоступны. Программа на рис. 6.6 включает `time1.h` и компилируется вместе с файлом `time1.cpp`, описанном на рис. 6.5.

```
// FIG6_6.CPP
// Демонстрация ошибок вследствие попыток доступа
// к закрытым элементам класса
#include <iostream.h>
#include "time1.h"
main()
{
 Time t;

 // Ошибка: 'Time :: hour' недоступно
 t.hour = 7;

 // Ошибка: 'Time :: minute' недоступно
 cout << " минута = " << t.minute;

 return 0;
}

Compiling FIG6_6.CPP:
Error FIG6_6.CPP 12: 'Time :: hour' is not accessible
Error FIG6_6.CPP 15: 'Time :: minute' is not accessible
```

**Рис. 6.6.** Ошибочная попытка доступа к закрытым элементам класса

### **Хороший стиль программирования 6.4**

Если вы намереваетесь сначала перечислить в определении класса закрытые элементы, используйте явно метку **private**, несмотря на то, что **private** предполагается по умолчанию. Это облегчит чтение программы. Но мы предпочитаем первым в определении класса помещать список открытой части **public**.

### **Хороший стиль программирования 6.5**

Несмотря на то, что спецификаторы **public** и **private** могут повторяться и чередоваться, перечисляйте сначала все элементы класса открытой группы, а затем все элементы закрытой группы. Это концентрирует внимание клиентов класса в большей степени на его открытом интерфейсе, чем на реализации класса.

### **Замечание по технике программирования 6.11**

Делайте все данные-элементы класса закрытыми. Используйте открытые функции-элементы для задания и получения значений закрытых данных-элементов. Такая архитектура помогает скрыть реализацию класса от его клиентов, что снижает число ошибок и улучшает модифицируемость программ.

Клиент класса может быть функцией-элементом другого класса или глобальной функцией.

По умолчанию доступ к элементам класса — **private**. Доступ к элементам класса можно явно установить как **public**, **protected** (как мы увидим в главе 9) или **private**. В отличие от этого доступ к элементам структуры **struct** по умолчанию — **public**. Доступ к элементам структуры **struct** также может быть установлен явно как **public**, **protected** или **private**.

### **Замечание по технике программирования 6.12**

Разработчики классов используют доступ типа **public**, **protected** или **private**, чтобы обеспечить скрытие информации и принцип наименьших привилегий.

Заметим, что элементы класса по умолчанию являются закрытыми, так что никогда не возникает необходимости явно использовать спецификатор доступа к элементам **private**. Однако, многие программисты предпочитают сначала описывать интерфейс класса (т.е. открытые элементы класса), а затем перечислять закрытые элементы, откуда вытекает необходимость явного использования в определении класса спецификатора доступа к элементам **private**.

### **Хороший стиль программирования 6.6**

Использование спецификаторов доступа к элементам **public**, **protected** и **private** лишь по одному разу в любом определении класса позволяет избежать путаницы.

Из того, что данные класса закрытые, не следует, что клиенты не могут изменять эти данные. Данные могут быть изменены функциями-элементами или друзьями этого класса. Как мы увидим, эти функции должны быть спроектированы так, чтобы гарантировать целостность данных.

Доступ к закрытым данным класса должен тщательно контролироваться использованием функций-элементов, называемых *функциями доступа*. Например, чтобы разрешить клиентам прочитать закрытое значение данных, класс может иметь функцию «получить» (*«get»*). Чтобы дать клиентам возможность изменять закрытые данные, класс может иметь функцию «установить» (*«set»*). Казалось бы, подобные изменения противоречат смыслу закрытых данных. Но функция-элемент *set* (*«установить»*) может обеспечить проверку правильности данных (например, проверку диапазона) и дать уве-

ренность в том, что данные установлены верно. Функция *set* может также быть переводчиком между формой данных, используемой в интерфейсе, и формой, используемой в реализации. Функция *get* (*получить*) не требует представления данных в «сыром», необработанном виде; функция *get* может редактировать данные и ограничивать область данных, видимых клиенту.

#### Замечание по технике программирования 6.13

Разработчик класса не обязательно должен снабжать каждый элемент закрытых данных функциями *get* и *set*; такие возможности должны быть обеспечены, только тогда, когда это имеет смысл, и лишь после тщательного обдумывания разработчиком класса.

#### Замечание по технике программирования 6.14

Задание данных-элементов класса закрытыми, а функций-элементов класса открытыми облегчает отладку, так как проблемы с манипуляциями данных локализуются в рамках либо функций-элементов класса, либо друзей класса.

## 6.9. Функции доступа и обслуживающие функции-утилиты

Не все функции-элементы необходимо делать открытыми как часть интерфейса класса. Некоторые функции-элементы оставляются закрытыми и служат обслуживающими функциями-утилитами для других функций класса.

#### Замечание по технике программирования 6.15

Функции-элементы можно разбить на ряд категорий: функции, которые читают и возвращают значения закрытых данных-элементов; функции, которые устанавливают значения закрытых данных-элементов; функции, которые реализуют возможности класса; функции, которые выполняют для класса различные вспомогательные операции, такие, как задание начальных значений объектам класса, присваивания объектам класса, преобразования между классами и встроенными типами или между классами и другими классами, выделение памяти для объектов класса.

Функции доступа могут читать или отображать данные. Другим типичным применением функций доступа является проверка истинности или ложности условий — такие функции часто называют *предикатными функциями*. Примером предикатной функции могла бы быть функция *isEmpty* для любого класса контейнера — класса, способного содержать внутри себя много объектов, например, связного списка, стека или очереди. Программа проверяла бы функцию *isEmpty* прежде, чем пытаться прочесть очередной элемент из объекта контейнера. Предикатная функция *isFull* могла бы проверять объект класса контейнер, чтобы выяснить, имеется ли в нем дополнительное пространство.

Рис. 6.7 демонстрирует запись функции-утилиты. Функция-утилита не является частью интерфейса класса. Она является закрытой функцией-элементом, которая поддерживает работу открытых функций-элементов класса. Функции-утилиты не предназначены для использования клиентами класса.

Класс **SalesPerson** имеет массив, содержащий 12 сведений о месячных продажах, которым с помощью конструктора присвоены нулевые начальные значения и которым значения, задаваемые пользователем, устанавливаются с помощью функции **setSales**. Открытая функция-элемент **printAnnualSales** печатает сумму продаж за последние 12 месяцев. Функция-утилита **totalAnnualSales** суммирует сведения о продажах за 12 месяцев, обеспечивая работу **printAnnualSales**. Функция-элемент **printAnnualSales** редактирует сведения о продажах и переводит их в формат суммы долларов.

```
// SALESP.H
// Определение класса SalesPerson
// Функции-элементы, описаны в SALESP.CPP

#ifndef SALESP_H
#define SALESP_H

class SalesPerson {
public:
 SalesPerson(); // конструктор
 void setSales(int, double); // Установка введенных
 // пользователем сведений
 // о продажах за один месяц.
 void printAnnualSales();

private:
 double sales[12]; //сведения о продажах за 12 месяцев
 double totalAnnualSales(); //функция-утилита
};

#endif
```

Рис. 6.7. Использование функции-утилиты (часть1 из 3)

```
// SALESP.CPP
// Функции-элементы класса SalesPerson
#include <iostream.h>
#include <iomanip.h>
#include "salesp.h"

// Функция конструктор присваивает начальные значения массиву
SalesPerson::SalesPerson()
{
 for (int i = 0; i < 12; i++)
 sales[i] = 0.0;
}

// Функция установки сведений о продажах за 1 из 12 месяцев
void SalesPerson::setSales(int month, double amount)
{
 if (month >= 1 && month <= 12 && amount > 0)
 sales[month - 1] = amount;
 else
 cout << "Ошибочный месяц или сведения о продажах" << endl;
}

// Закрытая функция-утилита для суммирования продаж за год
double SalesPerson::totalAnnualSales()
{
```

```

 double total = 0.0;

 for (int i = 0; i < 12; i++)
total += sales[i];

 return total;
 }

// Печать суммы годовых продаж
void SalesPerson::printAnnualSales()
{
 cout << setprecision(2)
 << setiosflags(ios::fixed | ios::showpoint)
 << endl << "Сумма продаж за год: $"
 << totalAnnualSales() << endl;
}

```

**Рис. 6.7.** Использование функции-утилиты (часть2 из 3)

```

// FIG6_7.CPP
// Демонстрация функции-утилиты
// Компиляция совместно с SALESP.CPP
#include <iostream.h>
#include "salesp.h"

main()
{
 SalesPerson s; //создание объекта s класса SalesPerson
 double salesFigure;

 for (int i = 1; i <= 12; i++) (
 cout << "Введите объем продаж за месяц "
 << i << ": ";
 cin >> salesFigure;
 s.setSales(i, salesFigure);
 }

 s.printAnnualSales();

 return 0;
}

```

---

Введите объем продаж за месяц 1: 5314.76  
 Введите объем продаж за месяц 2: 4292.38  
 Введите объем продаж за месяц 3: 4589.83  
 Введите объем продаж за месяц 4: 5534.03  
 Введите объем продаж за месяц 5: 4376.34  
 Введите объем продаж за месяц 6: 5698.45  
 Введите объем продаж за месяц 7: 4439.22  
 Введите объем продаж за месяц 8: 5893.57  
 Введите объем продаж за месяц 9: 4909.67  
 Введите объем продаж за месяц 10: 5123.45  
 Введите объем продаж за месяц 11: 4024.97  
 Введите объем продаж за месяц 12: 5923.92

Сумма продаж за год: \$60120.59

**Рис. 6.7.** Использование функции-утилиты (часть 3 из 3)

## 6.10. Инициализация объектов класса: конструкторы

После создания объекта его элементы могут быть инициализированы с помощью функции *конструктор*. Конструктор — это функция-элемент класса с тем же именем, что и класс. Программист предусматривает конструктор, который затем автоматически вызывается при создании объекта (создании экземпляра класса). *Данные-элементы класса не могут получать начальные значения в определении класса.* Они либо должны получить эти значения в конструкторе класса, либо их значения можно установить позже, после создания объекта. Конструкторы не могут указывать типы возвращаемых значений или возвращать какие-то значения. Конструкторы можно перегружать, чтобы обеспечить множество начальных значений объектов класса.

### Типичная ошибка программирования 6.5

Попытка объявить тип возвращаемого значения для конструктора или возвратить значение из конструктора.

### Хороший стиль программирования 6.7

В соответствующих случаях (почти всегда) предусматривайте конструктор для уверенности в том, что каждый объект получил соответствующие, имеющие смысл начальные значения.

### Хороший стиль программирования 6.8

Каждая функция-элемент (или дружественная функция), которая изменяет исходные данные-элементы, должна гарантировать, что данные остаются в не противоречащем друг другу согласованном состоянии.

Когда объявляется объект класса, между именем объекта и точкой с запятой можно в скобках указать *список инициализации элементов*. Эти начальные значения передаются в конструктор класса. Скоро мы увидим несколько примеров подобных *вызовов конструкторов*.

## 6.11. Использование конструкторов с аргументами по умолчанию

Конструктор из `time1.cpp` (рис. 6.5) присваивает нулевые (т.е. соответствующие 12 часам по полуночи в военном формате времени) начальные значения переменным `hour`, `minute` и `second`. Конструктор может содержать значения аргументов по умолчанию. Программа на рис. 6.8 переопределяет функцию конструктор `Time` так, чтобы она включала нулевые значения аргументов по умолчанию для каждой переменной. Задание в конструкторе аргументов по умолчанию позволяет гарантировать, что объект будет находиться в непротиворечивом состоянии, даже если в вызове конструктора не указаны никакие значения. Созданный программистом конструктор, у которого все аргументы — аргументы по умолчанию (или который не требует никаких аргументов), называется *конструктором с умолчанием*, т.е. конструктором, который можно вызывать без указания каких-либо аргументов.

```

// TIME2.H
// Объявление класса Time.
// Функции-элементы определены в TIME.CPP,
// предотвращение неоднократного включения заголовочного файла
#ifndef TIME2_H
#define TIME2_H

class Time {
public:
 Time(int = 0, int = 0, int = 0); //конструктор с умолчанием
 void setTime(int, int, int);
 void printMilitary();
 void printStandard();
private:
 int hour;
 int minute;
 int second;
};

#endif

```

**Рис. 6.8.** Использование конструктора с аргументами по умолчанию (часть 1 из 4)

```

// TIME2.CPP
// Определения функций-элементов класса Time
#include <iostream.h>
#include "time2.h"

// Функция конструктор для задания начальных значений закрытых данных.
// По умолчанию значения равны 0 (смотри определение класса).
Time::Time(int hr, int min, int sec)
 { setTime(hr, min, sec); }

// Установка значений hour, minute и second.
// Неверные значения устанавливаются равными 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;
}

//Отображение времени в военном формате: HH:MM:SS
void Time::printMilitary ()
{
 cout << (hour < 10 ? "0" : "") << hour << ":"
 << (minute < 10 ? "0" : "") << minute << ":"
 << (second < 10 ? "0" : "") << second; }

// Отображение времени 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. 6.8.** Использование конструктора с аргументами по умолчанию (часть 2 из 4)

Для каждого класса может существовать только один конструктор с умолчанием. В этой программе конструктор вызывает функцию-элемент `setTime` со значениями, передаваемыми конструктору (или значениями по умолчанию), чтобы гарантировать, что значение, предназначенное для `hour`, находится в диапазоне от 0 до 23, а значения для `minute` и `second` — в диапазоне от 0 до 59. Если значение выходит за пределы диапазона, оно устанавливается равным нулю с помощью `setTime` (это пример гарантии того, что данные-элементы будут в непротиворечивом состоянии).

```
// FIG6_8.CPP
// Демонстрация функции конструктора с умолчанием
// для класса Time.
#include <iostream.h>
#include "time2.h"

main()
{
 Time t1, t2(2), t3(21, 34), t4(12, 25, 42),
 t5(27, 74, 99);

 cout << "Варианты:" << endl
 << "все аргументы по умолчанию:" << endl << " ";
 t1.printMilitary();
 cout << endl << " ";
 t1.printStandard();

 cout << endl << "часы заданы; минуты и секунды по умолчанию:"
 << endl << " ";
 t2.printMilitary();
 cout << endl << " ";
 t2.printStandard();

 cout << endl << "часы и минуты заданы; секунды по умолчанию:"
 << endl << " ";
 t3.printMilitary();
 cout << endl << " ";
 t3.printStandard();

 cout << endl << "часы, минуты и секунды заданы:"
 << endl << " ";
 t4.printMilitary();
 cout << endl << " ";
 t4.printStandard();

 cout << endl << "все значения заданы неверно:"
 << endl << " ";
 t5.printMilitary();
 cout << endl << " ";
 t5.printStandard();

 return 0;
}
```

Рис. 6.8. Использование конструктора с аргументами по умолчанию (часть 3 из 4)

```

Варианты:
все аргументы по умолчанию:
 00:00:00
 12:00:00 AM
часы заданы; минуты и секунды по умолчанию:
 02:00:00
 2:00:00 AM
часы и минуты заданы; секунды по умолчанию:
 21:34:00
 9:34:00 PM
часы, минуты и секунды заданы:
 12:25:42
 12:25:42 PM
все значения заданы неверно:
 00:00:00
 12:00:00 AM

```

**Рис. 6.8.** Использование конструктора с аргументами по умолчанию (часть 4 из 4)

Заметим, что конструктор `Time` мог бы быть написан с включением тех же самых операторов, что имеются в функции-элементе `setTime`. Это могло бы быть несколько более эффективным, потому что исключается лишний вызов `setTime`. Однако, одинаковые коды в конструкторе `Time` и в функции-элементе `setTime` затрудняют сопровождение этой программы. Если реализация функции-элемента `setTime` изменится, соответствующие изменения надо будет вносить и в реализацию конструктора `Time`. То, что конструктор `Time` непосредственно вызывает `setTime`, позволяет делать любые изменения только в реализации `setTime`. Это уменьшает вероятность ошибки программирования при изменениях в реализации. Кроме того, эффективность конструктора `Time` может быть увеличена путем явного объявления конструктора `inline` или путем описания конструктора в определении класса (это означает неявное определение функции как `inline`).

### Замечание по технике программирования 6.16

Если функция-элемент класса уже обеспечивает все или часть функциональных возможностей, требуемых конструктором (или другой функцией-элементом), вызывайте эту функцию-элемент из конструктора (или другой функции-элемента). Это упрощает сопровождение программы и уменьшает вероятность ошибки при изменении реализации кода.

### Хороший стиль программирования 6.9

Объявляйте аргументы функции по умолчанию только в прототипе функции внутри определения класса в заголовочном файле.

### Типичная ошибка программирования 6.6

Указание начальных значений по умолчанию для одной и той же функции-элемента как в заголовочном файле, так и в описании функции-элемента.

Программа на рис. 6.8 создает 5 экземпляров объектов `Time` и задает им начальные значения: одному — со всеми тремя аргументами по умолчанию в вызове конструктора, второму — с одним указанным аргументом, третье-

му — с двумя указанными аргументами, четвертому — с тремя указанными аргументами и пятому — с тремя неверно указанными аргументами. Отображается содержание данных каждого объекта после его создания и задания начальных значений.

Если для класса не определено никакого конструктора, компилятор создает конструктор с умолчанием. Такой конструктор не задает никаких начальных значений, так что после создания объекта нет никакой гарантии, что он находится в непротиворечивом состоянии.

## 6.12. Использование деструкторов

*Деструктор* — это специальная функция-элемент класса. Имя деструктора совпадает с именем класса, но перед ним ставится символ *тильда* (~). Это соглашение о наименовании появилось интуитивно, потому что, как мы увидим в последующих главах, операция тильда является поразрядной операцией дополнения, а по смыслу деструктор является дополнением конструктора.

Деструктор класса вызывается при уничтожении объекта — например, когда выполняемая программа покидает область действия, в которой был создан объект этого класса. На самом деле деструктор сам не уничтожает объект — он выполняет *подготовку завершения* перед тем, как система освобождает область памяти, в которой хранился объект, чтобы использовать ее для размещения новых объектов.

Деструктор не принимает никаких параметров и не возвращает никаких значений. Класс может иметь только один деструктор — перегрузка деструктора не разрешается.

### Типичная ошибка программирования 6.7

Попытки передать аргументы деструктору, вернуть значения из деструктора или перегрузить деструктор.

Заметим, что представленные до сих пор классы не были обеспечены деструкторами. На самом деле, деструкторы редко используются с простыми классами. В главе 8 мы увидим, что деструкторы имеют смысл в классах, использующих динамическое распределение памяти под объекты (например, для массивов и строк). В главе 7, мы обсудим, как динамически распределять и перераспределять память.

## 6.13. Когда вызываются конструкторы и деструкторы

Конструкторы и деструкторы вызываются автоматически. Последовательность, в которой выполняется вызов этих функций, зависит от последовательности, в которой процесс выполнения входит и выходит из областей действия, в которых создаются объекты. В общем случае вызовы деструктора выполняются в порядке, обратном вызовам конструктора. Однако, классы памяти могут изменять последовательность вызовов деструкторов.

Конструкторы объектов, объявленных в глобальной области действия, вызываются раньше, чем любая функция данного файла (включая `main`) на-

чинает выполняться. Соответствующие деструкторы вызываются, когда завершается `main` или когда вызывается функция `exit` (смотри главу 18, «Другие темы»).

Конструкторы автоматических локальных объектов вызываются, когда процесс выполнения достигает места, где объекты объявляются. Соответствующие деструкторы вызываются, когда покидается область действия объектов (т.е. покидается блок, в котором эти объекты объявлены). Конструкторы и деструкторы для автоматических объектов вызываются каждый раз при входе и выходе из области действия.

Конструкторы статических локальных объектов вызываются сразу же, как только процесс выполнения достигает места, где объекты были впервые объявлены. Соответствующие деструкторы вызываются, когда завершается `main` или когда вызывается функция `exit`.

Программа на рис. 6.9 показывает последовательность, в которой вызываются конструкторы и деструкторы объектов типа `CreateAndDestroy` в нескольких областях действия. Программа объявляет объект `first` в глобальной области действия. Его конструктор вызывается, как только программа начинает выполнение, а его деструктор вызывается по завершении программы, после того, как все другие объекты уничтожены.

Функция `main` объявляет три объекта. Объекты `second` и `fourth` являются локальными автоматическими объектами, а объект `third` — статическим локальным объектом. Конструкторы каждого из этих объектов вызываются, когда процесс выполнения достигает места, где объекты были объявлены. Деструкторы объектов `fourth` и `second` вызываются в соответствующем порядке, когда заканчивается `main`. Поскольку объект `third` — статический, он существует до завершения программы. Деструктор объекта `third` вызывается раньше деструктора для `first`, но после уничтожения всех других объектов.

Функция `creat` объявляет три объекта — локальные автоматические объекты `fifth` и `seventh` и статический локальный объект `sixth`. Деструкторы для объектов `seventh` и `fifth` вызываются в соответствующем порядке по окончании `creat`. Поскольку `sixth` — статический объект, он существует до завершения программы. Деструктор для `sixth` вызывается раньше деструктора для `third` и `first`, но после уничтожения всех других объектов.

```
// CREATE.H
// Определение класса CreateAndDestroy.
// Функции-элементы определены в CREATE.CPP.

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
 CreateAndDestroy(int); // конструктор
 ~CreateAndDestroy(); // деструктор
private:
 int data;
};

#endif
```

**Рис. 6.9.** Демонстрация последовательности, в которой вызываются конструктор и деструктор (часть 1 из 4)

```
// CREATE.CPP
// Определения функций-элементов для класса CreateAndDestroy
#include <iostream.h>
#include "create.h"

CreateAndDestroy::CreateAndDestroy(int value)
{
 data = value;
 cout << "Объект " << data << " конструктор";
}

CreateAndDestroy::~CreateAndDestroy()
{ cout << "Объект " << data << " деструктор " << endl; }
```

Рис. 6.9. Демонстрация последовательности, в которой вызываются конструктор и деструктор (часть 2 из 4)

```
// FIG6_9.CPP
// Демонстрация последовательности, в которой вызываются
// конструкторы и деструкторы.
#include <iostream.h>
#include "create.cpp"

void create(void); // прототип
CreateAndDestroy first(1); // глобальный объект
main ()
{
 cout << " (глобальный созданный до main)" << endl;

 CreateAndDestroy second(2); // локальный объект
 cout << " (локальный автоматический в main)" << endl;

 static CreateAndDestroy third(3); // локальный объект
 cout << " (локальный статический в main)" << endl;
 create(); // вызов функции создания объектов

 CreateAndDestroy fourth(4); // локальный объект
 cout << " (локальный автоматический в main)" << endl;
return 0;
}

// Функция создания объектов
void create(void)
{
 CreateAndDestroy fifth(5);
 cout << " (локальный автоматический в create)" << endl;

 static CreateAndDestroy sixth(6);
 cout << " (локальный статический в create)" << endl;

 CreateAndDestroy seventh(7);
 cout << " (локальный автоматический в create)" << endl;
}
```

Рис. 6.9. Демонстрация последовательности, в которой вызываются конструктор и деструктор (часть 3 из 4)

|                      |                                     |
|----------------------|-------------------------------------|
| Объект 1 конструктор | (глобальный созданный до main)      |
| Объект 2 конструктор | (локальный автоматический в main)   |
| Объект 3 конструктор | (локальный статический в main)      |
| Объект 5 конструктор | (локальный автоматический в create) |
| Объект 6 конструктор | (локальный статический в create)    |
| Объект 7 конструктор | (локальный автоматический в create) |
| Объект 7 деструктор  |                                     |
| Объект 5 деструктор  |                                     |
| Объект 4 конструктор | (локальный автоматический в main)   |
| Объект 4 деструктор  |                                     |
| Объект 2 деструктор  |                                     |
| Объект 6 деструктор  |                                     |
| Объект 3 деструктор  |                                     |
| Объект 1 деструктор  |                                     |

Рис. 6.9. Демонстрация последовательности, в которой вызываются конструктор и деструктор (часть 4 из 4)

## 6.14. Использование данных-элементов и функций-элементов

Закрытые данные-элементы можно изменять только с помощью функций-элементов (и дружественных функций) класса. Типичным изменением могла бы быть корректировка баланса клиентов банка (например, закрытого элемента данных класса `BankAccount`) с помощью функции-элемента `computeInterest`.

Классы часто предусматривают открытые функции-элементы, позволяющие клиентам класса устанавливать (т.е. записывать) или получать (т.е. читать) значения закрытых данных-элементов. Эти функции можно не называть конкретно «`set`» или «`get`», но все же часто применяют именно эти названия. Более точно, функция-элемент, которая устанавливает элемент данных `interestRate`, должна была бы называться `setInterestRate`, а функция-элемент, которая читает элемент данных `interestRate`, должна была бы называться `getInterestRate`. Читающие функции обычно называют также функциями «запросов».

Казалось бы, что предоставление возможностей как установки, так и чтения, по существу, то же самое, что задание открытых данных-элементов. Однако в C++ существует одна тонкость, которая делает этот язык таким привлекательным для создания программного обеспечения. Если данные-элементы открытые, то они могут быть прочитаны или записаны по желанию с помощью любой функции в программе. Если данные-элементы закрытые, то открытая функция «`get`», позволяет другим функциям читать при желании эти данные, причем функция «`get`» управляет форматированием и отображением данных. Открытая функция «`set`» может, а более точно — должна тщательно анализировать любую попытку изменить значение закрытого элемента данных. Это должно обеспечивать гарантию, что новое значение соответствует этому элементу данных. Например, должны быть отвергнуты попытки установить день месяца, равный 37, отрицательный вес человека, численную величину символьного значения, значение оценки, равное 185, при шкале оценок от 0 до 100 и т.д.

### **Замечание по технике программирования 6.17**

Задание закрытых данных-элементов и управление доступом к ним, особенно доступом к записи данных-элементов, посредством открытых функций-элементов помогает гарантировать целостность данных.

Выгоды целостности данных не вытекают просто автоматически из того, что данные-элементы объявляются закрытыми. Программист должен обеспечивать проверку их правильности. C++ предоставляет среду программирования, в которой программист может проектировать хорошие программы в удобной ему манере.

### **Хороший стиль программирования 6.10**

Функции-элементы, которые записывают значения закрытых данных, должны проверять правильность предполагаемых новых значений; если они неправильные, то эти функции установить закрытые данные-элементы в соответствующее им непротиворечивое состояние.

Клиенты класса должны быть уведомлены о попытке присвоить данным-элементам неверные значения. По этой причине функции записи данных-элементов класса часто оформляются так, чтобы они возвращали значения, указывающие на то, что была совершена попытка присвоить объекту класса неверное значение. Это дает возможность клиентам класса проверить значения, возвращаемые функцией записи, чтобы определить, является ли объект, которым они манипулируют, допустимым, и предпринять соответствующие действия, если объект недопустимый. В упражнениях вас попросят изменить программу на рис. 6.10 так, чтобы возвращать соответствующие значения ошибок из записывающих функций.

Программа на рис. 6.10 расширяет наш класс `Time` так, чтобы он включал функции чтения и записи закрытых данных-элементов `hour`, `minute` и `second`. Функции записи жестко управляют установкой данных-элементов. Попытки задать любым данным-элементам неправильные значения вызывают присваивание этим данным-элементам нулевых значений (и, таким образом, приведение данных-элементов в непротиворечивое состояние). Каждая функция чтения просто возвращает соответствующее значение данных-элементов. Программа сначала использует функции записи, чтобы задать правильные значения закрытым данным-элементам объекта `t` класса `Time`, затем использует функцию чтения, чтобы вывести эти значения на экран. Далее функции записи пытаются задать элементам `hour` и `second` неправильные значения, а элементу `minute` — правильное, и затем функции чтения направляют эти значения на экран. Результат подтверждает, что неправильные значения вызывают установку данных-элементов в нулевое состояние. В итоге программа устанавливает время **11:58:00** и прибавляет 3 минуты при вызове функции `incrementMinutes`. Функция `incrementMinutes` не является элементом класса; поэтому она использует функции-элементы записи и чтения для соответствующего увеличения элемента `minute`. Это функционирует правильно, но снижает производительность из-за многократных вызовов функций. В следующей главе мы обсудим запись дружественных функций как средства устранения этого недостатка.

### Типичная ошибка программирования 6.8

Конструктор может вызывать другие функции-элементы класса такие, как функции записи и чтения. Но поскольку конструктор инициализирует объект, данные-элементы могут в этот момент еще не быть в непротиворечивом состоянии. Использование данных-элементов до того, как они получили соответствующие начальные значения, может вызвать ошибки.

Функции записи особенно важны с точки зрения разработки программного обеспечения, поскольку они могут производить проверку правильности данных. Функции записи и чтения имеют и другие достоинства при разработке программного обеспечения.

```
// TIME3. H
// Объявление класса Time.
// Функции-элементы, определены в TIME3.CPP

// Предотвращение многократного включения заголовочного файла
#ifndef TIME3_H
#define TIME3_H

class Time {
public:
 Time(int = 0, int = 0, int = 0); // конструктор

 // функции записи
 void setTime(int, int, int); // установка часов, минут,
 // секунд
 void setHour(int); // установка часа
 void setMinute(int); // установка минут
 void setSecond(int); // установка секунд

 // функции чтения
 int getHour(); // возвращает час
 int getMinute(); // возвращает минуты
 int getSecond(); // возвращает секунды

 void printMilitary(); // выводит военное время
 void printStandard(); // выводит стандартное время

private:
 int hour; // 0 - 23
 int minute; // 0 - 59
 int second; // 0 - 59
};

#endif
```

Рис. 6.10. Объявление класса Time (часть1 из 4)

```
// TIME3.CPP
// Определения функций-элементов класса Time.
#include "time3.h"
#include <iostream.h>

// Функция конструктор для задания начальных значений
// закрытых данных вызывает функцию-элемент setTime, чтобы
// установить значения переменных по умолчанию равными нулю
// (смотри определение класса).
Time::Time(int hr, int min, int sec) { setTime(hr, min, sec); }

// Установка значений часов, минут, секунд.
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;
}

// Установка значения часа
void Time::setHour(int h) { hour = (h >= 0 && h < 24) ? h : 0; }

// Установка значения минут
void Time::setMinute(int m)
{ minute = (m >= 0 && m < 60) ? m : 0; }

// Установка значения секунд
void Time::setSecond(int s)
{ second = (s >= 0 && s < 60) ? s : 0; }

// Получение значения часа
int Time::getHour() { return hour; }

// Получение значения минут
int Time::getMinute() { return minute; }

// Получение значения секунд
int Time::getSecond() { return second; }

// Отображение времени в военном формате: HH:MM:SS
void Time::printMilitary()
{
 cout << (hour < 10 ? "0" : " ") << hour << ":"
 << (minute < 10 ? "0" : " ") << minute << ":"
 << (second < 10 ? "0" : " ") << second;
}

// Отображение времени 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. 6.10. Определения функций-элементов класса Time (часть 2 из 4)

```
// FIG6_10.CPP
// Демонстрация функций записи и чтения класса Time

#include <iostream.h>
#include "time3.h"

void incrementMinutes(Time &, int);

main()
{
 Time t;

 t.setHour(17);
 t.setMinute(34);
 t.setSecond(25);

 cout << "Результат установки всех правильных значений:" << endl
 << " час: " << t.getHour()
 << " Минуты: " << t.getMinute()
 << " Секунды: " << t.getSecond() << endl << endl;

 t.setHour(234);
 t.setMinute(43);
 t.setSecond(6373);

 cout << "Результат попытки установить неправильные часы и "
 << "секунды: " << endl << " час: " << t.getHour()
 << " Минуты: " << t.getMinute()
 << " Секунды: " << t.getSecond() << endl << endl;

 t.setTime(11, 58, 0);
 incrementMinutes(t, 3);
 return 0;
}

void incrementMinutes(Time &tt, int count)
{
 cout << "Увеличение минут на " << count
 << endl << "Начальное время: ";
 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 << endl << "минуты + 1: ";
 tt.printStandard();
 }

 cout << endl;
}
```

Рис. 6.10. Использование функций записи и чтения (часть 3 из 4)

**Результат установки всех правильных значений:**

Час: 17 Минуты: 34 Секунды: 25

**Результат попытки установить неправильные часы и секунды:**

Час: 0 Минуты: 43 Секунды: 0

**Увеличение минут на 3**

**Начальное время:** 11: 58:00AM

минуты + 1: 11: 59:00AM

минуты + 1: 12:00:00PM

минуты + 1: 12:01:00PM

Рис. 6.10. Использование функций записи и чтения (часть 4 из 4)

### **Замечание по технике программирования 6.18**

Доступ к закрытым данным посредством функций записи и чтения не только защищает данные-элементы от присваивания им неправильных значений, но и отделяет клиентов класса от внутреннего представления данных-элементов. Таким образом, если внутреннее представление этих данных по каким-либо причинам (обычно из-за требований сокращения объема памяти или повышения производительности) изменяется, достаточно изменить только функции-элементы, а клиентам не требуется вносить никаких изменений, пока остается неизменным интерфейс функций-элементов. Однако, возможно, потребуется перекомпиляция клиентов данного класса.

## **6.15. Тонкий момент: возвращение ссылки на закрытые данные-элементы**

Ссылка на объект сама по себе является псевдонимом объекта и, следовательно, может быть использована с левой стороны оператора присваивания. Таким образом, ссылка является вполне допустимой L-величиной, которая может получать значение. Единственный (к сожалению!) способ использовать эту возможность — иметь открытую функцию-элемент класса, возвращающую неконстантную ссылку на закрытый элемент данных этого класса.

Рисунок 6.11 использует упрощенную версию класса Time для демонстрации возвращения ссылки на закрытый элемент данных. Такое возвращение в действительности осуществляется вызовом псевдонима функции badSetHour для закрытого элемента данных hour. Этот вызов функции можно использовать во всех отношениях так же, как закрытый элемент данных, включая его как L-величину в оператор присваивания!

### **Хороший стиль программирования 6.11**

Никогда не возвращайте из открытой функции-элемента неконстантную ссылку (или указатель) на закрытый элемент данных. Возвращение такой ссылки нарушает инкапсуляцию класса.

Программа начинается объявлением объекта t класса Time и ссылкой hourRef, которой присвоено значение, возвращенное вызовом t.badSetHour(20). Программа отображает на экране значение псевдонима hourRef. Далее этот псевдоним используется для установки значения hour равным 30 (неправильное значение) и это значение снова отображается на экране. В конце как L-величина используется вызов самой функции, ему присваивается значение 74 (другое неправильное значение), которое тоже отображается на экране.

```

// TIME4.H
// Объявление класса Time.
// Функции-элементы, определены в TIME4.CPP

// Предотвращение многократного включения 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

```

**Рис. 6.11.** Возвращение ссылки на закрытый элементу данных (часть 1 из 3)

```

// TIME4.CPP
// Определения функций-элементов класса Time.
#include "time4.h"
#include <iostream.h>

// Функция конструктор для задания начальных значений
// закрытым данным вызывает функцию-элемент setTime, чтобы
// установить значения переменных по умолчанию равными нулю
// (смотри определение класса).
Time::Time(int hr, int min, int sec) { setTime(hr, min, sec); }

// Установка значений часов, минут, секунд.
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;
}

// Установка значения часа
int Time::getHour() { return hour; }

// ПЛОХАЯ ПРАКТИКА ПРОГРАММИРОВАНИЯ:
// Возвращение ссылки на закрытый элемент данных.
int &Time::badSetHour(int hh)
{
 hour = (hh >= 0 && hh < 24) ? hh : 0;
 return hour; // ОПАСНОЕ возвращение ссылки
}

```

**Рис. 6.11.** Возвращение ссылки на закрытый элементу данных (часть 2 из 3)

```

// FIG6_11.CPP
// Демонстрация открытой функции-элемента, которая
// возвращает ссылку на закрытый элемент данных.
// Класс Time для этого примера укорочен.

#include <iostream.h>
#include "time4.h"

main()
{
 Time t;
 int &hourRef = t.badSetHour(20);

 cout << "часы перед изменением: " << hourRef << endl;
 hourRef = 30; // неправильное изменение
 cout << "часы после изменения: " << t.getHour() << endl;

 // ОПАСНОСТЬ: вызов функции, которая возвращает ссылку,
 // может быть использован как L-величина.
 t.badSetHour(12) = 74;
 cout << endl << "*****" << endl
 << "ПЛОХАЯ ПРАКТИКА ПРОГРАММИРОВАНИЯ!!!" << endl
 << "badSetHour как L-величина, часы: "
 << t.getHour()
 << endl << "*****" << endl;

 return 0;
}

```

---

Часы перед изменением: 20

Часы после изменения: 30

---

\*\*\*\*\*  
 ПЛОХАЯ ПРАКТИКА ПРОГРАММИРОВАНИЯ!!!  
 badSetHour как L-величина, часы: 74  
 \*\*\*\*\*

Рис. 6.11. Возвращение ссылки на закрытый элементу данных (часть 3 из 3)

## 6.16. Присваивание побитовым копированием по умолчанию

Для присваивания объекта другому объекту того же типа используется операция присваивания (`=`). Такое присваивание обычно выполняется с помощью *побитового копирования*: каждый элемент объекта копируется индивидуально в такой же элемент другого объекта (смотри рис. 6.12). (Замечание: побитовое копирование может вызвать серьезные проблемы, когда применяется к классу, чьи данные-элементы используют динамическое распределение памяти; в главе 8 «Перегрузка операций» мы обсудим эти проблемы и покажем, как их решать.)

Объекты могут передаваться как аргументы функций и могут возвращаться из функций. Такие передача и возвращение выполняются по умолчанию вызовом по значению, т.е. передается или возвращается копия объекта (мы представим несколько примеров в главе 8 «Перегрузка операций»).

```

// FIG6_12.CPP
// Демонстрация присваивания одного объекта другому
// с использованием побитового копирования по умолчанию

#include <iostream.h>

// Простой класс Date
class Date {
public:
 Date(int = 1, int = 1, int = 1990); // конструктор
 // с умолчанием
 void print();
private:
 int month;
 int day;
 int year;
};

// Простой конструктор Date без проверки диапазонов
Date::Date(int m, int d, int y)
{
 month = m;
 day = d;
 year = y;
}

// Печать Date в форме mm-dd-yyyy
void Date::print ()
{
 cout << month << '-' << day << '-' << year; }
}

main ()
{
 Date date1(7, 4, 1993), date2; //д2 по умолчанию
 //равен 1/1/90

 cout << "date1 = ";
 date1.print();
 cout << endl << "date2 = ";
 date2.print();

 date2 = date1; //присваивание побитовым копированием
 // по умолчанию
 cout << endl << endl
 << "После побитового копирования по умолчанию date2 = ";
 date2.print();
 cout << endl;

 return 0;
}

date1 = 7-4-1993
date2 = 1-1-1990

```

**После побитового копирования по умолчанию date2 = 7-4-1993**

**Рис. 6.12.** Присваивание одного объекта другому с использованием побитового копирования по умолчанию

### Совет по повышению эффективности 6.4

Передача объекта вызовом по значению хороша с точки зрения безопасности, поскольку вызываемая функция не имеет доступа к исходному объекту, но вызов по значению может ухудшить производительность в случае создания копии большого объекта. Объект может быть передан вызовом по ссылке путем передачи либо указателя, либо ссылки на объект. Вызов по ссылке способствует хорошей производительности, но с точки зрения безопасности хуже предыдущего, поскольку функции предоставлен доступ к исходному объекту. Безопасной альтернативой является вызов со ссылкой `const`.

## 6.17. Повторное использование программного обеспечения

Те, кто пишет объектно-ориентированные программы, концентрируют внимание на реализации полезных классов. Существуют огромные возможности сбора и каталогизации классов для обеспечения возможности доступа к ним значительной части сообщества программистов. Многие библиотеки классов уже существуют, другие повсеместно разрабатываются. Существуют попытки сделать эти библиотеки широкодоступными с целью конструирования программного обеспечения из существующих, хорошо определенных, тщательно проверенных, хорошо документированных, компактных, широко доступных компонентов. Этот способ повторного использования программ может ускорить разработку мощного, высококачественного программного обеспечения. Становится возможной *ускоренная разработка приложений* (*rapid applications development — RAD*).

Однако, для полной реализации потенциала повторного использования программного обеспечения нужно решить ряд важных проблем. Необходимы способы каталогизации, лицензирования, механизмы защиты от искажений авторских копий классов, способы паспортизации, помогающие разработчикам новых систем определить, существуют ли необходимые им объекты, механизмы поиска для определения того, какие из классов доступны и насколько они соответствуют требованиям разработчика программного обеспечения, и тому подобное. Необходимо провести множество интересных исследований и решить множество проблем разработки. Поскольку потенциальное значение решения этих проблем огромно, они будут решены.

## 6.18. Размышления об объектах: программирование классов для моделирования лифта

В разделах «Размышления об объектах» глав с 1 по 5 мы познакомились с основами объектного ориентирования и провели объектно-ориентированное проектирование модели лифта. В рамках главы 6 мы познакомились с деталями программирования и использования классов на языке C++. К настоящему моменту вы готовы (и, возможно, страстно желаете) начать программирование вашей модели лифта.

### Лабораторное задание 5 по лифту

1. Для каждого класса, который вы выделили в разделах «Размышления об объектах» глав со 2 по 5, напишите соответствующие определения классов на C++. Для каждого класса включите как заголовочный файл, так и исходный файл определения функций-элементов.
2. Напишите программу драйвера, который проверяет каждый из этих классов и пытается запустить на выполнение полную модель лифта. ПРЕДУПРЕЖДЕНИЕ: возможно, вам придется подождать, пока вы не изучите главу 7 «Классы: часть II», прежде чем вы будете в состоянии создать приемлемо работающую версию вашей модели. Так что будьте терпеливы и реализуйте только те части модели лифта, для которых вам хватает знаний, приобретенных в главе 6. В главе 7 вы узнаете о композиции, т.е. о создании классов, которые содержат другие классы в качестве элементов; эта техника могла бы помочь вам представить, например, объект кнопку внутри лифта как элемент лифта. Кроме того, в главе 7 вы узнаете, как динамически создавать и уничтожать объекты с помощью `new` и `delete`; это поможет вам создавать новые объекты пассажиров, когда они должны появиться в модели, и уничтожать эти объекты пассажиров, когда они покидают модель (после выхода пассажира из лифта).
3. Для первой версии вашей модели спроектируйте только простой вывод, ориентированный на текст, который отобразит сообщения о каждом существенном событии, которое будет происходить. Ваши сообщения могли бы включать такие строки, как «Пассажир 1 прибывает на Этаж 1», «Пассажир 1 нажимает Кнопку «Вверх» Этажа 1», «Лифт прибывает на Этаж 1», «Пассажир 1 входит в Лифт» и т.д. Заметим, что мы рекомендуем вам печатать с заглавной буквы те слова каждого сообщения, которые представляют объекты вашей модели. Заметим также, что вы можете предпочесть отложить эту часть лабораторного занятия до прочтения главы 7.
4. Наиболее амбициозные студенты захотят использовать анимированный графический выход, который показывает, как модель лифта движется на экране вверх и вниз.

### Резюме

- Структуры — это совокупности типов данных, построенные с использованием данных других типов.
- Ключевое слово `struct` начинает определение структуры. Тело структуры заключается в фигурные скобки (`{` и `}`). Каждое определение структуры должно заканчиваться точкой с запятой.
- Тэг (имя-этикетка) структуры может использоваться для объявления переменных данного типа структуры.
- Определения структуры не резервируют место в памяти; они создают новые типы данных, которые используются далее для объявления переменных.
- Элементы структуры или класса доступны при использовании операций доступа к элементу — операции точка (`.`) и операции стрелка (`->`).

Операция точка обеспечивает доступ к элементу структуры посредством имени переменной объекта или ссылки на объект. Операция стрелка обеспечивает доступ к элементу структуры посредством указателя на объект.

- Препятствиями к созданию новых типов данных с помощью `struct` являются: возможность существования данных, не имеющих начальных значений; возможность существования данных с неправильными начальными значениями; необходимость изменения всех программ, использующих `struct`, при изменении реализации `struct`; отсутствие средств защиты, гарантирующих, что данные не содержат несогласованных значений.
- Классы предоставляют программисту возможность моделировать объекты с атрибутами и различными вариантами поведения. Типы классов можно определять в C++, используя ключевые слова `class` и `struct`, но обычно для этих целей используется ключевое слово `class`.
- Имя класса можно использовать для объявления объектов этого класса.
- Определения класса начинаются с ключевого слова `class`. Тело определения класса заключается в фигурные скобки ({ и }). Определение класса заканчивается точкой с запятой.
- Любые данные-элементы или функции-элементы, объявленные в классе после метки `public:`, являются открытыми и видимыми для любой функции, для которой доступен объект данного класса.
- Любые данные-элементы или функции-элементы, объявленные в классе после метки `private:`, являются закрытыми и видимыми только друзьям и другим элементам класса.
- Метки доступа к элементу всегда заканчиваются двоеточием (:) и могут появляться в определении класса неоднократно и в любой последовательности.
- Закрытые данные недоступны вне класса.
- Говорят, что реализация класса скрыта от его клиентов.
- Конструктор — это специальная функция-элемент с тем же именем, что и класс, которая используется для инициализации элементов объекта класса. Конструкторы вызываются при создании объектов соответствующих классов.
- Функция с тем же именем, что и класс, но предваряемая знаком тильды (~), называется деструктором.
- Набор открытых функций-элементов класса называется интерфейсом класса или открытым интерфейсом.
- Если функция-элемент определена вне определения данного класса, имя функции предваряется именем класса и бинарной операцией разрешения области действия (::).
- Функции-элементы, определенные с использованием операции разрешения области действия вне определения класса, имеют областью действия этот класс.

- Функции-элементы, определенные в определении класса, автоматически встраиваются *inline*. Компилятор сохраняет право не встраивать любую функцию.
- Вызов функций-элементов более компактен, чем вызов функций в процедурном программировании, потому, что большинство данных, используемых в функции-элементе, непосредственно доступно ей в объекте.
- Внутри области действия класс на элементы класса можно ссылаться просто с помощью их имен. Вне области действия класс на элементы класса можно ссылаться посредством либо имени объекта, либо ссылки на объект, либо указателя на объект.
- Для доступа к элементам класса используются операции . и ->.
- Фундаментальный принцип разработки хорошего программного обеспечения состоит в отделении интерфейса от реализации для улучшения модифицируемости программы.
- Определения класса обычно помещаются в заголовочных файлах, а определения функций-элементов — в файлах исходных кодов, имеющих такое же имя.
- По умолчанию способ доступа в классах — *private*, так что все элементы после заголовка класса и до первого спецификатора доступа считаются закрытыми.
- Открытые элементы класса предоставляют набор услуг, которыми класс обеспечивает своих клиентов.
- Доступом к закрытым данным класса можно эффективно управлять с помощью функций-элементов, называемых функциями доступа. Если класс хочет позволить клиентам читать закрытые данные, он может обеспечить это с помощью функций чтения «*get*». Изменять закрытые данные класс может позволить своим клиентам с помощью функций записи «*set*».
- Данные-элементы класса обычно делаются закрытыми, а функции-элементы — открытыми. Некоторые функции-элементы могут быть закрытыми и играть роль функций-утилит для других функций класса.
- Данным-элементам класса нельзя задавать начальные значения в определении класса. Они должны получать начальные значения в конструкторе или же их значения могут быть установлены после создания соответствующего объекта.
- Конструкторы можно перегружать.
- После того, как объект класса получил соответствующие начальные значения, все функции-элементы, которые манипулируют объектом, должны давать гарантию, что объект остается в непротиворечивом состоянии.
- В объявлении объекта класса могут быть предусмотрены начальные значения. Эти начальные значения передаются конструктору класса.
- Конструкторы могут определять аргументы по умолчанию.
- Конструкторы не могут ни указывать тип возвращаемых значений, ни пытаться возвращать значения.

- Если для класса не определено никаких конструкторов, компилятор создает конструктор с умолчанием. Такой конструктор не выполняет присваивания никаких начальных значений, так что после создания объекта нет гарантий, что он находится в непротиворечивом состоянии.
- Деструктор автоматического объекта вызывается при выходе из области действия объекта. Сам по себе деструктор в действительности не уничтожает объект, но он выполняет подготовку его уничтожения перед тем, как система использует память объекта, которую ранее занимал объект.
- Деструкторы не получают параметров и не возвращают значений. Класс может иметь только один деструктор.
- Операция присваивания (=) используется для присваивания объекта другому объекту того же типа. Такое присваивание обычно выполняется с помощью побитового копирования по умолчанию. Побитовое копирование не является идеальным для всех классов.

## Терминология

|                                  |                                                  |
|----------------------------------|--------------------------------------------------|
| <code>class</code>               | операция разрешения области действия (::)        |
| <code>private:</code>            | операция ссылки &                                |
| <code>protected:</code>          | определение класса                               |
| <code>public:</code>             | открытые элементы                                |
| абстрактный тип данных           | открытый интерфейс класса                        |
| атрибут                          | побитовая копия                                  |
| бинарная операция разрешения     | поведение                                        |
| области действия (::)            | повторно используемый код                        |
| встраиваемая <code>inline</code> | повторное использование программного обеспечения |
| функция-элемент                  | предикатная функция                              |
| вхождение в область действия     | принцип наименьших привилегий                    |
| выход из области действия        | процедурное программирование                     |
| глобальный объект                | расширяемость                                    |
| данные-элементы                  | реализация класса                                |
| деструктор                       | скрытие информации                               |
| заголовочный файл                | сообщение                                        |
| закрытые элементы                | спецификаторы доступа к элементам                |
| инициализация объекта класса     | список инициализации элементов                   |
| инкапсуляция                     | статический локальный объект                     |
| интерфейс класса                 | тильда (-) в имени деструктора                   |
| исходный файл                    | тип данных                                       |
| класс                            | тип, определяемый пользователем                  |
| клиент класса                    | управление доступом к элементам                  |
| конструктор                      | ускоренная разработка приложений (RAD)           |
| конструктор с умолчанием         | услуги класса                                    |
| набор функций                    | функция доступа                                  |
| непротиворечивое состояние       | функция записи (set)                             |
| данных-элементов                 | функция запроса                                  |
| нестатический локальный объект   | функция не элемент класса                        |
| область действия класс           | функция чтения (get)                             |
| область действия файл            | функция-утилита                                  |
| объект                           | функция-элемент                                  |
| объектно-ориентированное         | экземпляр объекта класса                         |
| программирование (ООП)           |                                                  |
| операция доступа к элементу      |                                                  |
| класса (.)                       |                                                  |

## Типичные ошибки программирования

- 6.1. Забывается точка с запятой в конце определения класса (или структуры).
- 6.2. Попытка явно присвоить начальное значение данным-элементам в определении класса.
- 6.3. Попытка перегрузить функцию-элемент класса с помощью функции не из области действия этого класса.
- 6.4. Попытка с помощью функции, не являющейся элементом определенного класса (или другом этого класса) получить доступ к элементам этого класса.
- 6.5. Попытка объявить тип возвращаемого значения для конструктора или возвратить значение из конструктора.
- 6.6. Указание начальных значений по умолчанию для одной и той же функции-элемента как в заголовочном файле, так и в описании функции-элемента.
- 6.7. Попытки передать аргументы деструктору, вернуть значения из деструктора или перегрузить деструктор.
- 6.8. Конструктор может вызывать другие функции-элементы класса такие, как функции записи и чтения. Но поскольку конструктор инициализирует объект, данные-элементы могут в этот момент еще не быть в непротиворечивом состоянии. Использование данных-элементов до того, как они получили соответствующие начальные значения, может вызвать ошибки.

## Хороший стиль программирования

- 6.1. Используйте при определении класса каждый спецификатор доступа к элементам только один раз, что сделает программу более ясной и простотой для чтения. Размещайте первыми элементы `public`, являющиеся общедоступными.
- 6.2. Используйте директивы препроцессора `#ifndef`, `#define` и `#endif` для того, чтобы избежать включения в программу заголовочных файлов более одного раза.
- 6.3. Используйте имя заголовочного файла с символом подчеркивания вместо точки в директивах препроцессора `#ifndef` и `#define` заголовочного файла.
- 6.4. Если вы намереваетесь сначала перечислить в определении класса закрытые элементы, используйте явно метку `private`, несмотря на то, что `private` предполагается по умолчанию. Это облегчит чтение программы. Но мы предпочитаем первым в определении класса помещать список открытой части `public`.
- 6.5. Несмотря на то, что спецификаторы `public` и `private` могут повторяться и чередоваться, перечисляйте сначала все элементы класса открытой группы, а затем все элементы закрытой группы. Это кон-

центрирует внимание клиентов класса в большей степени на его открытом интерфейсе, чем на реализации класса.

- 6.6. Использование спецификаторов доступа к элементам `public`, `protected` и `private` лишь по одному разу в любом определении класса позволяет избежать путаницы.
- 6.7. В соответствующих случаях (почти всегда) предусматривайте конструктор для уверенности в том, что каждый объект получил соответствующие, имеющие смысл начальные значения.
- 6.8. Каждая функция-элемент (или дружественная функция), которая изменяет исходные данные-элементы, должна гарантировать, что данные остаются в не противоречащем друг другу согласованном состоянии.
- 6.9. Объявляйте аргументы функции по умолчанию только в прототипе функции внутри определения класса в заголовочном файле.
- 6.10. Функции-элементы, которые записывают значения закрытых данных, должны проверять правильность предполагаемых новых значений; если они неправильные то эти функции должны установить закрытые данные-элементы в соответствующее им непротиворечивое состояние.
- 6.11. Никогда не возвращайте из открытой функции-элемента неконстантную ссылку (или указатель) на закрытый элемент данных. Возвращение такой ссылки нарушает инкапсуляцию класса.

### Советы по повышению эффективности

- 6.1. Обычно структуры передаются вызовом по значению. Чтобы избежать накладных расходов на копирование структуры, передавайте структуры вызовом по ссылке.
- 6.2. Чтобы избежать накладных расходов вызова по значению и вдобавок получить выгоды защиты исходных данных от изменения, передавайте аргументы большого размера как ссылки `const`.
- 6.3. Описание небольших функций-элементов внутри определения класса автоматически встраивает функцию-элемент `inline` (если компилятор решит делать это). Это может улучшить производительность, но не способствует улучшению качества проектирования программного обеспечения.
- 6.4. Передача объекта вызовом по значению хороша с точки зрения безопасности, поскольку вызываемая функция не имеет доступа к исходному объекту, но вызов по значению может ухудшить производительность в случае создания копии большого объекта. Объект может быть передан вызовом по ссылке путем передачи либо указателя, либо ссылки на объект. Вызов по ссылке способствует хорошей производительности, но с точки зрения безопасности хуже предыдущего, поскольку функции предоставлен доступ к исходному объекту. Безопасной альтернативой является вызов со ссылкой `const`.

## Замечания по технике программирования

- 6.1. Важно писать программы, которые легко понимать и поддерживать. Изменения являются скорее правилом, чем исключением. Программисты должны предвидеть, что их коды будут изменяться. Как мы увидим, классы способствуют модифицируемости программ.
- 6.2. Клиенты класса используют класс, не зная внутренних деталей его реализации. Если реализация класса изменяется (например, с целью улучшения производительности), интерфейс класса остается неизменным и исходный код клиента класса не требует изменений. Это значительно упрощает модификацию систем.
- 6.3. Функции-элементы обычно короче, чем обычные функции в программах без объектной ориентации, потому что достоверность данных, хранимых в данных-элементах, идеально проверена конструктором и функциями-элементами, которые сохраняют новые данные.
- 6.4. Клиенты имеют доступ к интерфейсу класса, но не имеют доступа к реализации класса.
- 6.5. Объявление функций-элементов внутри определения класса и описание функций-элементов вне этого определения отделяет интерфейс класса от его реализации. Это способствует высокому качеству разработки программного обеспечения.
- 6.6. Использование принципов объектно-ориентированного программирования часто может упростить вызовы функции за счет уменьшения числа передаваемых параметров. Это достоинство объектно-ориентированного программирования проистекает из того факта, что инкапсуляция данных-элементов и функций-элементов внутри объекта дает функциям-элементам право прямого доступа к данным-элементам.
- 6.7. Помещайте объявление класса в заголовочный файл, чтобы оно было доступно любому клиенту, который захочет использовать класс. Это формирует открытый интерфейс класса. Помещайте определения функций-элементов класса в исходный файл. Это формирует реализацию класса.
- 6.8. Клиенты класса не нуждаются в доступе к исходному коду класса для того, чтобы использовать класс. Однако, клиенты должны иметь возможность связаться с объектным кодом класса.
- 6.9. Информация, важная для интерфейса класса, должна включаться в заголовочный файл. Информация, которая будет использоваться только внутри класса и которая не является необходимой для клиентов класса, должна включаться в неоглашаемый исходный файл. Это еще один пример принципа наименьших привилегий.
- 6.10. C++ способствует созданию программ, не зависящих от реализации. Если, изменяется реализация класса, используемого программой, не зависящей от реализации, то код этой программы не требует изменения, но может потребоваться его перекомпиляция.

- 6.11. Делайте все данные-элементы класса закрытыми. Используйте открытые функции-элементы для задания и получения значений закрытых данных-элементов. Такая архитектура помогает скрыть реализацию класса от его клиентов, что снижает число ошибок и улучшает модифицируемость программ.
- 6.12. Разработчики классов используют доступ типа `public`, `protected` или `private`, чтобы обеспечить скрытие информации и принцип наименьших привилегий.
- 6.13. Разработчик класса не обязательно должен снабжать каждый элемент закрытых данных функциями `get` и `set`; такие возможности должны быть обеспечены, только тогда, когда это имеет смысл, и лишь после тщательного обдумывания разработчиком класса.
- 6.14. Задание данных-элементов класса закрытыми, а функций-элементов класса открытыми облегчает отладку, так как проблемы с манипуляциями данных локализуются в рамках либо функций-элементов класса, либо друзей класса.
- 6.15. Функции-элементы можно разбить на ряд категорий: функции, которые читают и возвращают значения закрытых данных-элементов; функции, которые устанавливают значения закрытых данных-элементов; функции, которые реализуют возможности класса; функции, которые выполняют для класса различные вспомогательные операции, такие, как задание начальных значений объектам класса, присваивания объектам класса, преобразования между классами и встроенными типами или между классами и другими классами, выделение памяти для объектов класса.
- 6.16. Если функция-элемент класса уже обеспечивает все или часть функциональных возможностей, требуемых конструктором (или другой функцией-элементом), вызывайте эту функцию-элемент из конструктора (или другой функции-элемента). Это упрощает сопровождение программы и уменьшает вероятность ошибки при изменении реализации кода.
- 6.17. Задание закрытых данных-элементов и управление доступом к ним, особенно доступом к записи данных-элементов, посредством открытых функций-элементов помогает гарантировать целостность данных.
- 6.18. Доступ к закрытым данным посредством функций записи и чтения не только защищает данные-элементы от присваивания им неправильных значений, но и отделяет клиентов класса от внутреннего представления данных-элементов. Таким образом, если внутреннее представление этих данных по каким-либо причинам (обычно из-за требований сокращения объема памяти или повышения производительности) изменяется, достаточно изменить только функции-элементы, а клиентам не требуется вносить никаких изменений, пока остается неизменным интерфейс функций-элементов. Однако, возможно, потребуется перекомпиляция клиентов данного класса.

## Упражнения для самопроверки

**6.1.** Заполнить пробелы в следующих утверждениях:

- Ключевое слово \_\_\_\_\_ начинает определение структуры.
- Элементы класса доступны посредством операции \_\_\_\_\_ в сочетании с объектом класса или посредством операции \_\_\_\_\_ в сочетании с указателем на объект класса.
- Элементы класса, указанные как \_\_\_\_\_, доступны только функциям-элементам класса и друзьям класса.
- \_\_\_\_\_ является специальной функцией-элементом, используемой для задания начальных значений элементам данных класса.
- По умолчанию доступ к элементам класса — \_\_\_\_\_.
- Функция \_\_\_\_\_ используется для присваивания значений закрытым данным-элементам класса.
- \_\_\_\_\_ можно использовать для присваивания объекта класса другому объекту того же класса.
- Функции-элементы класса обычно делаются \_\_\_\_\_ типа, а данные-элементы — \_\_\_\_\_ типа.
- Функция \_\_\_\_\_ используется для получения значений закрытых данных класса.
- Набор открытых функций-элементов класса рассматривается как \_\_\_\_\_ класса.
- Говорят, что реализация класса скрыта от его клиентов или \_\_\_\_\_.
- Для введения определения класса можно использовать ключевые слова \_\_\_\_\_ и \_\_\_\_\_.
- Элементы класса, указанные как \_\_\_\_\_, доступны везде в области действия объекта класса.

**6.2.** Найдите ошибку (или ошибки) в каждом из следующих пунктов и объясните, как их исправить.

- Допустим, что в классе Time объявлен следующий прототип.

```
void ~Time(int);
```

- Следующий фрагмент является частью определения класса Time.

```
class Time {
public:
 // прототипы функций
private:
 int hour = 0;
 int minute = 0;
 int second = 0;
};
```

- Допустим, что в классе Employee объявлен следующий прототип.

```
int Employee(const char *, const char *);
```

## Ответы на упражнения для самопроверки

- 6.1. a) struct. b) точка (.), стрелка (->). c) private. d) Конструктор. e) private. f) записи «set». g) Поэлементное копирование по умолчанию (с помощью операции присваивания). h) открытого, закрытого. i) чтения «get». j) интерфейс. k) инкапсулирована. l) class, struct. m) public.
- 6.2. a) Ошибка: Деструкторы не могут возвращать значения или принимать аргументы.  
Исправление: переместите тип **void** возвращаемого значения и параметр **int** из определения.
- b) Ошибка: элементы не могут явно получать начальные значения в определении класса.  
Исправление: уберите явное задание начальных значений из определения класса и задавайте начальные значения элементов в конструкторе.
- c) Ошибка: конструкторы не могут возвращать значения.  
Исправление: переместите тип **int** возвращаемого значения из объявления.

## Упражнения

- 6.3. Каково назначение операции разрешения области действия?
- 6.4. Сравните и сопоставьте нотацию **struct** и **class** в C++.
- 6.5. Создайте конструктор, способный использовать текущее время, даваемое функцией **time()**, объявленной в заголовочном файле **time.h** стандартной библиотеки С, чтобы задавать начальные значения объекту класса **Time**.
- 6.6. Создайте класс с именем **Complex** для выполнения арифметических действий с комплексными числами. Напишите программу драйвера для проверки вашего класса.

Комплексные числа имеют форму

`realPart + imaginaryPart *j`

где *j* — квадратный корень из -1.

Используйте переменные с плавающей запятой для представления закрытых данных этого класса. Создайте функцию конструктор, которая позволяет объекту этого класса принимать начальные значения при его объявлении. Создайте открытые функции-элементы для каждого из следующих пунктов:

- а) Сложение двух комплексных чисел: отдельно складываются действительные и мнимые части.
- б) Вычитание двух комплексных чисел: действительная часть правого операнда вычитается из действительной части левого операнда, а мнимая часть правого операнда вычитается из мнимой части левого операнда.
- в) Печать комплексных чисел в форме (a, b), где a — действительная часть, а b — мнимая часть.

- 6.7. Создайте класс по имени **Rational** для выполнения арифметических действий с дробями. Напишите программу драйвера для проверки вашего класса.

Используйте целые переменные для представления закрытых данных класса — числителя и знаменателя. Создайте функцию конструктор, которая позволяет объекту этого класса принимать начальные значения при его объявлении. Конструктор должен содержать значения по умолчанию на случай отсутствия заданных начальных значений и должен хранить дроби в сокращенном виде (т.е. дробь  $2/4$  должна храниться в объекте как 1 в числителе и 2 в знаменателе). Создайте открытые функции-элементы для каждого из следующих случаев:

- a) Сложение двух чисел **Rational**. Результат должен храниться в сокращенной форме.
  - b) Вычитание двух чисел **Rational**. Результат должен храниться в сокращенной форме.
  - c) Перемножение двух чисел **Rational**. Результат должен храниться в сокращенной форме.
  - d) Деление двух чисел **Rational**. Результат должен храниться в сокращенной форме.
  - e) Печать чисел **Rational** в форме  $a / b$ , где  $a$  — числитель,  $a b$  — знаменатель.
  - f) Печать чисел **Rational** в форме с плавающей точкой.
- 6.8. Модифицируйте класс **Time** на рис. 6.10 так, чтобы включить функцию-элемент **tick**, которая дает приращение времени, хранящегося в объекте **Time**, равное одной секунде. Объект **Time** должен всегда находиться в непротиворечивом состоянии. Напишите программу-драйвер для проверки функции-элемента **tick** в цикле, которая печатала бы время в стандартном формате на каждой итерации цикла и иллюстрировала правильную работу функции-элемента **tick**. Удовствуйтесь в правильности работы в следующих случаях:
- a) Приращение с переходом в следующую минуту.
  - b) Приращение с переходом в следующий час.
  - c) Приращение с переходом в следующий день (т.е. от 11:59:59 PM к 12:00:00 AM).
- 6.9. Модифицируйте класс **Date** на рис.6.12 так, чтобы выполнить проверку ошибки в списке начальных значений для данных-элементов **month**, **day** и **year**. Кроме того, создайте функцию-элемент **nextDay**, которая бы увеличивала день на единицу. Объект **Date** должен всегда находиться в непротиворечивом состоянии. Напишите программу-драйвер, проверяющую функцию **nextDay** в цикле и печатающую время в стандартном формате на каждой итерации цикла, чтобы проиллюстрировать правильную работу функции **nextDay**. Удовствуйтесь в правильности работы в следующих случаях:
- a) Приращение с переходом в следующий месяц.
  - b) Приращение с переходом в следующий год.

- 6.10.** Объедините модифицированный класс `Time` упражнения 6.8 и модифицированный класс `Date` упражнения 6.9 в один класс по имени `DateAndTime` (в главе 9 мы обсудим наследование, которое позволит нам быстро решить эту задачу без изменения существующих определений классов). Модифицируйте функцию `tick` так, чтобы вызывать функцию `nextDay`, если время получает приращение с переходом на следующий день. Модифицируйте функции `PrintStandard` и `PrintMilitary`, чтобы выводить в добавление к времени еще и дату. Напишите программу-драйвер, проверяющую новый класс `DateAndTime`. Особо проверьте приращение времени с переходом на следующий день.
- 6.11.** Модифицируйте набор функций в программе на рис. 6.10 так, чтобы возвращать ошибочные значения в случае попытки задать неправильные значения данным-элементам объекта класса `Time`.
- 6.12.** Создайте класс `Rectangle` (прямоугольник). Класс имеет атрибуты `length` (длина) и `width` (ширина), каждый из которых по умолчанию равен 1. Он имеет функции-элементы, которые вычисляют периметр (`perimeter`) и площадь (`area`) прямоугольника. Он имеет функции записи и чтения как для `length`, так и для `width`. Функции записи должны проверять, что `length` и `width` — числа с плавающей запятой, находящиеся в пределах от 0.0 до 20.0.
- 6.13.** Создайте более изощренный, чем в упражнении 6.12, класс `Rectangle`. Этот класс хранит только декартовы координаты четырех углов прямоугольника. Конструктор вызывает набор функций, которые принимают четыре группы координат и проверяют, чтобы каждая из координат `x` и `y` находилась в первом квадранте, в диапазоне от 0.0 до 20.0. Это множество функций должно также проверять, что переданные координаты действительно определяют прямоугольник. Должны быть предусмотрены функции-элементы, вычисляющие `length`, `width`, `perimeter`, и `area`. Длиной должно считаться большее из двух измерений. Включите предикатную функцию `square`, которая определяла бы, не является ли прямоугольник квадратом.
- 6.14.** Модифицируйте класс `Rectangle` в упражнении 6.13 так, чтобы включить в него функцию `draw`, которая изображает прямоугольник внутри окна 25 на 25, перекрывающего часть первого квадранта, в котором находится прямоугольник. Включите функцию `setFillCharacter`, чтобы задавать символ, которым будет заполняться прямоугольник внутри. Включите функцию `setPerimeterCharacter`, чтобы задавать символ, которым будут печататься границы прямоугольника. Если вы войдете во вкус, вы можете включить функции масштабирования размера прямоугольника, его вращения и перемещения в пределах первого квадранта.
- 6.15.** Создайте класс `HugeInteger`, который использует массив из 40 элементов для хранения целых чисел вплоть до больших целых, содержащих по 40 цифр. Создайте функции-элементы `inputHugeInteger`, `outputHugeInteger`, `addHugeIntegers` и `subtractHugeIntegers` для ввода, вывода, сложения и вычитания этих больших целых. Для сравнения объектов `HugeInteger` создайте функции

`isEqualTo`, `isNotEqualTo`, `isGreater Than`, `isLessThan`, `isGreater ThanOrEqual To`, `isLessThanOrEqual To` — каждая из них является предикатной функцией, которая просто возвращает 1 (истина), если соответствующее соотношение между двумя большими целыми выполняется, и 0 (ложь) — если оно не выполняется. Создайте предикатную функцию `isZero`. Если вы войдете во вкус, подготовьте также функции-элементы `multiplayHugeIntegers`, `divideHugeIntegers` и `modulusHugeIntegers`.

- 6.16. Создайте класс `TicTacToe`, который предоставит вам возможность написать полную программу для игры в тик-так-тоу. Класс содержит как закрытые данные двумерный массив целых чисел 3 на 3, описывающий доску для игры. Конструктор должен присваивать нулевые начальные значения всем пустым полям. Играют два игрока. Помещайте 1 в клетку, указанную при перемещении первым игроком, и 2 в клетку, указанную вторым игроком. Каждое перемещение допустимо только на пустую клетку. После каждого перемещения определяйте, не выиграна ли игра или не получилась ли ничья. Если вы вошли во вкус, модифицируйте вашу программу так, чтобы компьютер выполнял перемещения за одного из игроков автоматически. При этом позвольте игроку указывать, хотел бы он или она ходить первым или вторым. Если же вы почувствовали исключительное увлечение, развивайте программу так, чтобы играть в трехмерный тик-так-тоу на доске 4 на 4 на 4 (*Предупреждение:* это чрезвычайно сложный проект, который может потребовать многонедельных усилий!).

г л а в а

---

# 7

## Классы: часть II



### Ц е л и

- Научиться динамически создавать и уничтожать объекты.
- Научиться определять константные объекты и функции-элементы.
- Понять назначение дружественных функций и классов.
- Понять, как используются статические данные-элементы и функции-элементы.
- Понять концепцию классов контейнеров.
- Понять принципы записи классов итераторов через элементы классов контейнеров.
- Понять, как используется указатель `this`.

## План

- 7.1. Введение
- 7.2. Константные объекты и функции-элементы
- 7.3. Композиция: классы как элементы других классов
- 7.4. Дружественные функции и дружественные классы
- 7.5. Использование указателя *this*
- 7.6. Динамическое распределение памяти с помощью операций *new* и *delete*
- 7.7. Статические элементы класса
- 7.8. Абстракция данных и скрытие информации
- 7.9. Пример: абстрактный тип данных массив
- 7.10. Пример: абстрактный тип данных строка
- 7.11. Пример: абстрактный тип данных очередь
- 7.12. Классы контейнеры и итераторы
- 7.13. Размышления об объектах: использование композиции и динамического управления объектом в модели лифта

*Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике проиграммирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения*

### 7.1. Введение

В этой главе мы продолжим изучение классов и абстрагирования данных. Мы обсудим ряд сложных тем и заложим основы для обсуждения классов и перегрузки операций в главе 8. Затем в главах 9 и 10 мы познакомимся с наследованием и полиморфизмом — техникой истинно объектно-ориентированного программирования.

## 7.2 Константные объекты и функции-элементы

Мы еще раз особо отмечаем *принцип наименьших привилегий* как один из наиболее фундаментальных принципов создания хорошего программного обеспечения. Рассмотрим один из способов применения этого принципа к объектам.

Некоторые объекты должны допускать изменения, другие — нет. Программист может использовать ключевое слово `const` для указания на то, что объект неизменяем — является константным и что любая попытка изменить объект является ошибкой. Например,

```
const Time noon(12, 0, 0);
```

объявляет как константный объект пооп класса `Time` и присваивает ему начальное значение 12 часов пополудни.

### Замечание по технике программирования 7.1

Обявление константного объекта помогает провести в жизнь принцип наименьших привилегий. Случайные попытки изменить объект отлавливаются во время компиляции и не вызывают ошибок во время выполнения.

Компиляторы C++ воспринимают обьявления `const` настолько неукоснительно, что в итоге не допускают никаких вызовов функций-элементов константных объектов (некоторые компиляторы дают в этих случаях только предупреждения). Это жестоко, поскольку клиенты объектов возможно захотят использовать различные функции-элементы чтения «`get`», а они, конечно, не изменяют объект. Чтобы обойти это, программист может обьявить константные функции-элементы; только они могут оперировать константными объектами. Конечно, константные функции-элементы не могут изменять объект — это не позволит компилятору.

Константная функция указывается как `const` и в обьявлении, и в описании с помощью ключевого слова `const` после списка параметров функции, но перед левой фигурной скобкой, которая начинает тело функции. Например, в приведенном ниже предложении обьявлена как константная функция-элемент некоторого класса `A`

```
int A::getValue() const {return privateDateMember};
```

которая просто возвращает значение одного из данных-элементов объекта. Если константная функция-элемент описывается вне определения класса, то *как обьявление функции-элемента, так и ее описание должны включать `const`.*

Здесь возникает интересная проблема для конструкторов и деструкторов, которые обычно должны изменять объект. Для конструкторов и деструкторов константных объектов обьявление `const` не требуется. Конструктор должен иметь возможность изменять объект с целью присваивания ему соответствующих начальных значений. Деструктор должен иметь возможность выполнять подготовку завершения работ перед уничтожением объекта.

Программа на рис. 7.1 создает константный объект класса `Time` и пытается изменить объект неконстантными функциями-элементами `setHour`, `setMinute` и `setSecond`. Как результат показаны генерированные компилятором Borland C++ предупреждения. Опция компилятора была установлена такой, чтобы в случае появления любого предупреждения компилятор не создавал исполняемого файла.

```

// TIME5.H
// Объявление класса Time.
// Функции-элементы описаны в TIMES.CPP

#ifndef TIME5_H
#define TIME5_H

class Time {
public:
 Time(int = 0, int = 0, int = 0); // конструктор по умолчанию

 // функции записи set
 void setTime(int, int, int); // установка времени
 void setHour(int); // установка часа
 void setMinute(int); // установка минут
 void setSecond(int); // установка секунд

 // функции чтения get (обычно объявляются const)
 int getHour() const; // возвращает значение часа
 int getMinute() const; // возвращает значение минут
 int getSecond() const; // возвращает значение секунд
 // функции печати (обычно объявляются const)
 void printMilitary() const; // печать военного
 // времени
 void printStandard() const; // печать стандартного
 // времени

private:
 int hour; // 0 - 23
 int minute; // 0-59
 int second; // 0 - 59
};

#endif

```

**Рис. 7.1.** Использование класса **Time** с константными объектами и константными функциями-элементами (часть 1 из 3)

### Хороший стиль программирования 7.1

Объявляйте как **const** все функции-элементы, которые предполагается использовать с константными объектами.

### Типичная ошибка программирования 7.1

Описание константной функции-элемента, которая изменяет данные-элементы объекта.

### Типичная ошибка программирования 7.2

Описание константной функции-элемента, которая вызывает неконстантную функцию-элемент.

```
// TIME5.CPP
// Описания функций-элементов класса Time.
#include <iostream.h>
#include "time5.h"

// Функция конструктор для инициализации закрытых данных.
// По умолчанию значения равны 0 (смотри описание класса).
Time::Time(int hr, int min, int sec) { setTime(hr, min, sec);}

// Установка значений часа, минут и секунд.
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;
}

// Установка значения часа
void Time::setHour(int h) { hour = (h >= 0 && h < 24) ? h : 0; }

// Установка значения минут
void Time::setMinute(int m)
{ minute = (m >= 0 && m < 60) ? m : 0; }

// Установка значения секунд
void Time::setSecond(int s)
{ second = (s >= 0 && s < 60) ? s : 0; }

// Чтение значения часа
int Time::getHour() const { return hour; }

// Чтение значения минут
int Time::getMinute() const { return minute; }

// Чтение значения секунд
int Time::getSecond() const { return second; }

// Отображение времени в военном формате: HH:MM:SS
void Time::printMilitary() const
{
 cout << (hour < 10 ? "0": "") << hour << ":"
 << (minute < 10 ? "0":"") << minute << ":"
 << (second < 10 ? "0"":") << second;
}

// Отображение времени в стандартном формате: HH:MM:SS AM
// (или PM)
void Time::printStandard() const
{
 cout << ((hour == 12) ? 12: hour % 12) << ":"
 << (minute < 10 ? "0"":") << minute << ":"
 << (second < 10 ? "0"":") << second
 << (hour< 12 ? "AM" : "PM");
}
```

Рис. 7.1. Использование класса **Time** с константными объектами и константными функциями-элементами (часть 2 из 3)

```

// FIG7_1.CPP
// Попытка получить доступ к константному объекту
// с не-константными функциями-элементами.

#include <iostream.h>
#include "time5.h"

main ()
{
 const Time t(19, 33, 52); // константный объект

 t.setHour(12); // ОШИБКА: не-константная функция элемент
 t.setMinute(20); // ОШИБКА: не-константная функция элемент
 t.setSecond(39); // ОШИБКА: не-константная функция элемент

 return 0;
}

Compiling FIG7_1.CPP:
Warning FIG7_1.CPP: Non-const function
 Time::setHour(int) called for const object
Warning FIG7_1.CPP: Non-const function
 Time::setMinute(int) callers for const object
Warning FIG7_1.CPP: Non-const function
 Time::setSecond(int) called for const object

```

**Рис. 7.1.** Использование класса **Time** с константными объектами и константными функциями-элементами (часть 3 из 3)

### Типичная ошибка программирования 7.3

Вызов неконстантной функции-элемента для константного объекта.

### Типичная ошибка программирования 7.4

Попытка изменить константный объект.

### Замечание по технике программирования 7.2

Константная функция-элемент может быть перегружена неконстантным вариантом. Выбор того, какая из перегруженных функций-элементов будет использоваться, осуществляется компилятором автоматически в зависимости от того, был объявлен объект как **const** или нет.

Константный объект не может быть изменен с помощью присваивания, так что он должен получить начальное значение. Если данные-элементы класса объявлены как **const**, то надо использовать *инициализатор элементов*, чтобы обеспечить конструктор объекта этого класса начальными значением данных-элементов. Рис. 7.2 демонстрирует использование инициализатора элементов для задания начального значения константному элементу **increment** класса **Increment**. Конструктор для **Increment** изменяется следующим образом:

```
Increment :: Increment (int c, int i)
: increment (i)
{ count = c; }
```

Запись : **increment (i)** вызывает задание начального значения элемента **increment**, равного **i**. Если необходимо задать начальные значения сразу нескольким элементам, просто включите их в список после двоеточия, разделяя запятыми. Используя инициаторы элементов, можно присвоить начальные значения всем данным-элементам.

```
// FIG7_2.CPP
// Использование инициализаторов элементов для
// инициализации данных константного встроенного типа.
#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 константного элемента
{ count = c; }

// Печать данных
void Increment::print() const
{
 cout << "count = " << count
 << ", increment = " << increment << endl;
}

main()
{
 Increment value(10, 5);

 cout << "Перед приращением: ";
 value.print();

 for (int j = 1; j <= 3; j++) {
 value.addIncrement();
 cout << "После приращения " << j << ": ";
 value.print();
 }
 return 0;
}
```

---

Перед приращением: count = 10, increment = 5

После приращения 1: count = 15, increment = 5

После приращения 2: count = 20, increment = 5

После приращения 3: count = 25, increment = 5

**Рис. 7.2.** Использование инициализаторов элементов для инициализации данных константного встроенного типа

Рисунок 7.3 иллюстрирует ошибки компиляции, выдаваемые компилятором Borland C++ в программе, которая пытается задать начальное значение элементу `increment`, используя оператор присваивания, а не инициализатор элементов.

```
// FIG7_3.CPP
// Попытка инициализировать данные константного
// встроенного типа 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 'increment'
 // не получает начального значения
 count = c;
 increment = i; // ОШИБКА: константный объект не может
 // быть изменен
}

// Печать данных
void Increment::print() const
{
 cout << "count = " << count
 << ", increment = " << increment << endl;
}

main()
{
 Increment value(10, 5);

 cout << "Перед приращением: ";
 value.print();

 for (int j = 1; j <= 3; j++) {
 value.addIncrement();
 cout << "После приращения " << j << ": ";
 value.print();
 }
 return 0;
}
```

---

Compiling FIG7\_3.CPP:

Warning FIG7\_3.CPP 18: Constant member 'increment' is  
not initialized

Error FIG7\_3.CPP 20: Cannot modify a const object

Warning FIG7\_3.CPP 21: Parameter 'i' is never used

**Рис. 7.3.** Ошибочная попытка задать начальное значение данным константного встроенного типа с помощью присваивания

### Типичная ошибка программирования 7.5

Нет инициализаторов константных данных-элементов.

### Замечание по технике программирования 7.3

Константные элементы класса (объекты и «переменные») должны получать начальные значения с помощью инициализаторов элементов. Присваивания недопустимы.

## **7.3. Композиция: классы как элементы других классов**

Поскольку объекту класса `AlarmClock` необходимо знать, когда предполагается подача сигнала тревоги, то почему бы не включить объект `Time` как элемент объекта `AlarmClock`? Такая возможность называется *композицией*. Класс может включать в себя объекты других классов в качестве элементов.

### Замечание по технике программирования 7.4

Одним из способов повторного использования программного обеспечения является композиция, когда класс включает в себя объекты других классов в качестве элементов.

Когда объект входит в область действия, автоматически вызывается его конструктор и нам надо указать, как аргументы передаются конструкторам объектов-элементов. Объекты-элементы создаются в том порядке, в котором они объявлены (а не в том порядке, в котором они перечислены в списке инициализаторов элементов конструктора), и до того, как будут созданы объекты включающего их класса.

Программа на рис. 7.4 использует классы `Employee` и `Date` для демонстрации объектов как элементов других объектов. Класс `Employee` содержит закрытые данные-элементы `lastName`, `firstName`, `birthDate` и `hireDate`. Элементы `birthDate` и `hireDate` являются объектами класса `Date`, который содержит закрытые данные-элементы `month`, `day` и `year`. Программа создает объект `Employee`, задает начальные значения его данным-элементам и отображает их на экране. Приведем синтаксис заголовка функции в описании 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)
```

Этот конструктор принимает восемь аргументов (`fname`, `lname`, `bmonth`, `bday`, `byear`, `hmonth`, `hday` и `hyear`). Двоеточие в заголовке отделяет инициализаторы элементов от списка параметров. Инициализаторы элементов ука-

зывают, что аргументы `Employee` передаются конструкторам объектов-элементов. В частности, `bmonth`, `bday` и `byear` передаются конструктору `birthDate`, а `hmonth`, `hday` и `hyear` — конструктору `hirehDate`. Инициализаторы элементов в списке разделяются запятыми.

Объекты-элементы не нуждаются в задании начальных значений посредством инициализаторов элементов. Если инициализаторы элементов не заданы, конструктор с умолчанием объекта-элемента будет вызван автоматически. Значения, если они были установленные конструктором с умолчанием, могут быть затем изменены с помощью функции записи `«set»`.

### Типичная ошибка программирования 7.6

Не предусмотрен конструктор с умолчанием для объекта-элемента, когда для этого объекта элемента не задан инициализатор элементов. Это может привести к тому, что объект-элемент не будет инициализирован.

### Совет по повышению эффективности 7.1

Инициализируйте объекты-элементы явно с помощью инициализаторов элементов. Это исключает накладные расходы связанные с повторной инициализацией объектов-элементов: первой при вызове конструктора с умолчанием объекта-элемента и второй при задании начальных значений объекта-элемента с помощью функции записи `«set»`.

```
// DATE1.H
// Объявление класса Date.
// Функции-элементы определены в DATE1.CPP
#ifndef DATE1_H
#define DATE1_H

class Date {
public:
 Date(int = 1, int = 1, int = 1900); //конструктор по умолчанию
 void print () const; // печать данных в формате
 // месяц/день/год

private:
 int month; // 1-12
 int day; // 1-31 в зависимости от месяца
 int year; // любой год

 // функция утилита для проверки соответствия дня месяцу и году
 int checkDay(int);
};

#endif
```

Рис. 7.4. Использование инициализаторов объектов-элементов (часть 1 из 5)

```
// DATE1.CPP
// Определения функций-элементов класса Date.

#include <iostream.h>
#include "date1.h"

// Конструктор: поддержка соответствующего значения месяца;
// вызов функции-утилиты checkDay для поддержки соответствующего
// значения дня.
Date::Date(int mn, int dy, int yr)

{
 if (mn > 0 && mn <= 12) // проверка месяца
 month = mn;
 else {
 month = 1;
 cout << "Месяц " << mn
 << " неправильный. Установлен месяц 1."
 << endl;
 }

 year = yr; // можно было бы тоже проверить
 day = checkDay(dy); //проверка дня

 cout << "Конструктор объекта Date ";
 print ();
 cout << endl;
}

// Функция-утилита для поддержки соответствующего
// значения дня в зависимости от месяца и года.
int Date::checkDay(int testDay)
{
 static int daysPerMonth[13] = {0, 31, 28, 31, 30,
 31, 30, 31, 31, 30,
 31, 30, 31};

 if (testDay > 0 && testDay <= daysPerMonth[month])
 return testDay;

 if (month == 2 && // Февраль: проверка високосного года
 testDay == 29 &&
 (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0)))
 return testDay;

 cout << "День " << testDay
 << " неправильный. Установлен день 1." << endl;

 return 1; // если неверное значение, объект остается
 // в непротиворечивом состоянии
}

// Печать объекта Date в форме месяц/день/год
void Date::print() const
{ cout << month << '/' << day << '/' << year;}
```

Рис. 7.4. Использование инициализаторов объектов-элементов (часть 2 из 5)

```

// EMPLOYEE.H
// Объявление класса Employee.
// Функции-элементы определены в EMPLOYEE.CPP
#ifndef EMPLOYEE_H
#define EMPLOYEE_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. 7.4.** Использование инициализаторов объектов-элементов (часть 3 из 5)

```

// EMPLOYEE.CPP
// Описания функций-элементов класса Employee.
#include <iostream.h>
#include <string.h>
#include "employee.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)
{
 // копирование fname в firstName и проверка их совпадения
 int length = strlen(fname);
 length = length < 25 ? length : 24;
 strncpy(firstName, fname, length);
 firstName[length] = '\0';

 // копирование lname в lastName и проверка их совпадения
 length = strlen(lname);
 length = length < 25 ? length : 24;
 strncpy(lastName, lname, 24);
 lastName[length] = '\0';

 cout << "Конструктор объекта Employee: "
 << firstName << ' ' << lastName << endl;
}

void Employee::print() const
{
 cout << lastName << ", " << firstName << endl << "Нанят: ";
 hireDate.print();
 cout << " День рождения: ";
 birthDate.print();
 cout << endl;
}

```

**Рис. 7.4.** Использование инициализаторов объектов-элементов (часть 4 из 5)

```

// FIG7_4.CPP
// Демонстрация объекта с объектом-элементом.
#include <iostream.h>
#include "employ1.h"
main()
{
 Employee e("Боб", "Джон", 7, 24, 49, 3, 12, 88);
 cout << endl;
 e.print();

 cout << endl << "Проверка конструктора Date "
 << "с неправильными значениями:" << endl;
 Date d(14, 35, 94); //Неправильные значения Date

 return 0;
}

```

---

```

Конструктор объекта Date 7/24/49
Конструктор объекта Date 3/12/88
Конструктор объекта Employee: Боб Джон

```

```

Джон, Боб
Нанят: 3/12/88 День рождения: 7/24/49

```

```

Проверка конструктора Date с неправильными значениями:
Месяц 14 неправильный. Установлен месяц 1.
День 35 неправильный. Установлен день 1.
Конструктор объекта Date 1/1/94

```

**Рис. 7.4.** Использование инициализаторов объектов-элементов (часть 5 из 5)

## 7.4. Дружественные функции и дружественные классы

*Дружественные функции* класса определяются вне области действия этого класса, но имеют право доступа к закрытым элементам *private* (и, как мы увидим в главе 9 «Наследование», к элементам *protected*) данного класса. Функция или класс в целом могут быть объявлены *другом* (*friend*) другого класса.

Дружественные функции используются для повышения производительности. Приведем формальный пример, показывающий, как работает дружественная функция. Далее в этой книге дружественные функции применяются для перегрузки операций, используемых классами, и для создания классов итераторов. Объекты класса итератора используются, чтобы последовательно выделять элементы или выполнять операции над элементами в объекте класса контейнера (смотри раздел 7.9). Объекты классов контейнеров способны хранить множество элементов в форме, подобной массиву.

Чтобы объявить функцию как друга (**friend**) класса, перед ее прототипом в описании класса ставится ключевое слово **friend**. Чтобы объявить класс **ClassTwo** как друга класса **ClassOne**, запишите объявление в форме

```
friend ClassTwo
```

в определение класса **ClassOne**.

### Замечание по технике программирования 7.5

Спецификаторы доступа к элементам **private**, **protected** и **public** не имеют отношения к объявлению дружественности, так что эти объявления дружественности могут помещаться в любом месте в описании класса.

### Хороший стиль программирования 7.2

Помещайте объявления дружественности первыми в классе непосредственно после его заголовка и не предваряйте их каким-либо спецификатором доступа к элементам.

Дружественность требует разрешения, т.е. чтобы класс В стал другом класса А, класс А должен объявить, что класс В — его друг. Таким образом дружественность не обладает ни свойством симметричности, ни свойством транзитивности, т.е. если класс А — друг класса В, а класс В — друг класса С, то отсюда не следует, что класс В — друг класса А, что класс С — друг класса В, или что класс А — друг класса С.

### Замечание по технике программирования 7.6

Некоторые члены сообщества объектно-ориентированного программирования считают, что «дружественность» портит скрытие информации и ослабляет значения объектно-ориентированного подхода к проектированию.

Программа на рис. 7.5 демонстрирует объявление и использование дружественной функции **setX** для установки закрытого элемента данных **x** класса **Count**. Заметим, что объявление **friend** появляется первым (по соглашению) в объявлении класса, даже раньше объявления закрытых функций-элементов.

```
// FIG7_5.CPP
// Друзья могут иметь доступ к закрытым элементам класса.
#include <iostream.h>

// Измененный класс Count
class Count{
 friend void setX(Count &, int); // объявление друга
public:
 Count() { x = 0; } // конструктор
 void print() const {cout << x << endl;} // вывод
private:
 int x; // элемент данных
};
```

Рис. 7.5. Друзья могут иметь доступ к закрытым элементам класса (часть 1 из 2)

```

// Можно изменять закрытую переменную класса Count,
// так как setX объявлена как дружественная функция класса Count
void setX(Count &c, int val)
{
 c.x = val; // разрешено: setX - друг Count
}

main()
{
 Count object;

 cout << "object.x после своего создания: ";
 object.print();
 cout << "object.x после вызова дружественной функции setX: ";
 setX(object, 8); // задание x другом
 object.print();

 return 0;
}

```

---

object.x после своего создания: 0

object.x после вызова дружественной функции setX: 8

Рис. 7.5. Друзья могут иметь доступ к закрытым элементам класса (часть 2 из 2)

Программа на рис. 7.6 демонстрирует сообщения, вырабатываемые компилятором, когда для изменения закрытого элемента данных `x` вызывается функция `cannotSetX`, не являющаяся дружественной. Рисунки 7.5 и 7.6 предназначены для ознакомления с формальным механизмом использования дружественных функций; практические примеры использования дружественных функций появятся в последующих главах.

```

// FIG7_6.CPP
// Функции не друзья или не элементы не могут иметь доступ
// к закрытым элементам класса.
#include <iostream.h>
// Измененный класс Count
class Count {
public:
 Count() { x = 0; } // конструктор
 void print() const { cout << x << endl; } // выход
private:
 int x; // элемент данных
};

// Функция пытается изменить закрытые данные класса Count,
// но не может, так как она - не друг Count.
void cannotSetX(Count &c, int val)
{
 c.x = val; // ОШИБКА: 'Count::x' недоступна
}

main()
{
 Count object;
 cannotSetX(object, 3); // cannotSetX не друг
 return 0;
}

```

Рис. 7.6. Функции, не являющиеся друзьями или элементами, не могут иметь доступ к закрытым элементам класса (часть 1 из 2)

```
Compiling FIG7_6.CPP:
Error FIG7 6.CPP 17: 'Count::x' is not accessible
Warning FIG7 6.CPP 18: Parameter 'c' is never used
Warning FIG7 6.CPP 18: Parameter 'val' is never used
```

**Рис. 7.6.** Функции, не являющиеся друзьями или элементами, не могут иметь доступ к закрытым элементам класса (часть 2 из 2)

Друзьями класса можно определить перегруженные функции. Каждая перегруженная функция, предназначенная в друзья, должна быть явно объявлена в описании класса как друг этого класса.

## 7.5. Использование указателя *this*

Когда функция-элемент ссылается на другой элемент какого-то объекта данного класса, откуда у C++ берется уверенность, что имеется ввиду соответствующий объект? Ответ заключается в том, что каждый объект сопровождается указателем на самого себя — называемым *указателем this* — это неявный аргумент во всех ссылках на элементы внутри этого объекта. Указатель *this* можно использовать также и явно. Каждый объект может определить свой собственный адрес с помощью ключевого слова *this*.

Указатель *this* неявно используется для ссылки как на данные-элементы, так и на функции-элементы объекта. Тип указателя *this* зависит от типа объекта и от того, объявлена ли функция-элемент, в которой используется *this*, как *const*. В неконстантной 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 *const Employee \* const* (константный указатель на объект *Employee*, который тоже константный).

А теперь мы покажем простой пример явного использование указателя *this*; позже мы представим несколько реальных сложных примеров использования *this*. Каждая функция-элемент имеет доступ к указателю *this* на объект, для которого вызван этот элемент.

### Совет по повышению эффективности 7.2

С целью экономии памяти для каждой функции-элемента существует только одна копия на класс и эта функция-элемент вызывается каждым объектом данного класса. С другой стороны, каждый объект имеет свою собственную копию данных-элементов класса.

Программа на рис. 7.7 демонстрирует явное использование указателя *this*, чтобы дать возможность функции-элементу класса *Test* печатать закрытую переменную *x* объекта *Test*.

С иллюстративными целями функция-элемент *print* на рис. 7.7 сначала печатает *x* непосредственно. Затем программа использует две различных записи для доступа к *x* посредством указателя *this* — операцию стрелки (*->*), примененную к указателю *this*, и операцию точка (*.*) для разыменования указателя *this*.

```

// FIG7_7.CPP
// Использование указателя this для ссылки на объекты-элементы.
#include <iostream.h>

class Test {
public:
 Test (int = 0); // конструктор по умолчанию
 void print() const;
private:
 int x;
};

Test::Test(int a) { x = a; } // конструктор

void Test::print() const
{
 cout << " x = " << x << endl
 << " this->x = " << this->x << endl
 << " (*this).x = " <<(*this).x << endl;
}

main()
{
 Test a(12);

 a.print();

 return 0;
}

 x = 12
 this->x = 12
(*this).x = 12

```

Рис. 7.7. Использование указателя **this**

Отметим круглые скобки, в которые заключено **\*this**, когда используется операция доступа к элементу точка **(.)**. Круглые скобки необходимы, так как операция точка имеет более высокий приоритет по сравнению с операцией **\***. Без круглых скобок выражение

**\*this.x**

оценивалось бы так же, как если бы были круглые скобки следующего вида  
**\*(this.x)**

Компилятор C++ воспринял бы это выражение как синтаксическую ошибку, так как операция доступа к элементу не может быть использована с указателем.

### Типичная ошибка программирования 7.7

Попытка использовать операцию доступа к элементу **(.)** с указателем на объект (операцию доступа к элементу точка можно использовать только с объектом или со ссылкой на объект).

Одним интересным применением указателя `this` является предотвращение присваивания объекта самому себе. Как мы увидим в главе 8 «Перегрузка операций», самоприсваивание может стать причиной серьезных ошибок в случаях, когда объекты содержат указатели на динамически распределяемую память.

Другим применением указателя `this` является возможность сцепленных вызовов функций элементов. Программа на рис. 7.8 иллюстрирует возвращение ссылки на объект `Time`, которое дает возможность сцепления вызовов функций-элементов класса `Time`. Каждая из функций-элементов `setTime`, `setHour`, `setMinute` и `setSecond` возвращает `*this` с типом возврата `Time &`.

Почему возвращение `*this` работает как ссылка? Операция точка `(.)` имеет ассоциативность слева направо, так что выражение

```
t.setHour(18).setMinute(30).setSecond(22);
```

сначала вычисляет `t.setHour(18)`, а затем возвращает ссылку на объект `t` как значение вызова этой функции.

```
// TIME6.H
// Объявление класса Time.
// Функции-элементы определены в TIME6.CPP

#ifndef TIME6_H
#define TIME6_H

class Time {
public:
 Time(int = 0, int = 0, int = 0); // конструктор по умолчанию

 // функции записи "set"
 Time &setTime(int, int, int); // установка часа, минут
 // и секунд
 Time &setHour(int); // установка часа
 Time &setMinute(int); // установка минут
 Time &setSecond(int); // установка секунд

 // функции чтения get (обычно объявляются const)
 int getHour() const; // возвращает значение часа
 int getMinute() const; // возвращает значение минут
 int getSecond() const; // возвращает значение секунд

 // функции печати (обычно объявляются const)
 void printMilitary() const; // печать военного
 // времени
 void printStandard() const; // печать стандартного
 // времени

private:
 int hour; // 0-23
 int minute; // 0-59
 int second; // 0-59
};

#endif
```

Рис. 7.8. Сцепление вызовов функций-элементов (часть 1 из 5)

Оставшееся выражение затем интерпретируется как

```
t.setMinute(30).setSecond(22);
```

Вызов `t.setMinute(30)` выполняется и возвращает эквивалент `t`. Оставшееся выражение интерпретируется как

```
t.setSecond(22);
```

Отметим, что вызовы

```
t.setTime(20, 20, 20).printStandard()
```

также используют особенности сцепления. Эти вызовы должны появляться именно в указанной последовательности, потому что `printStandard`, как описано в классе, не возвращает ссылку на `t`. Расположение вызова `printStandard` в предыдущем операторе перед вызовом `setTime` приводит к синтаксической ошибке.

```
// TIME6.CPP
// Определения функций-элементов класса Time.
#include "time6.h"
#include <iostream.h>

// Функция конструктор для задания начальных значений
// закрытым данным.
// Вызов функций-элементов setTime для установки переменных.
// По умолчанию значения равны 0 (смотри описание класса).
Time::Time(int hr, int min, int sec) { setTime(hr, min, sec); }

// установка часа, минут и секунд.
Time &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;
 return *this; // возможность сцепления
}

// Установка значения часа
Time &Time::setHour(int h)
{
 hour = (h >= 0 && h < 24) ? h : 0;
 return *this; // возможность сцепления
}

// Установка значения минут
Time &Time::setMinute(int m)
{
 minute = (m >= 0 && m < 60) ? m : 0;
 return *this; // возможность сцепления
}
```

Рис. 7.8. Сцепление вызовов функций-элементов (часть 2 из 5)

```

// TIME6.CPP: Продолжение
// Установка значения секунд
Time &Time::setSecond(int s)
{
 second = (s >= 0 && s < 60) ? s : 0;

 return *this; // возможность сцепления
}

// Получение значения часа
int Time::getHour() const { return hour; }

// Получение значения минут
int Time::getMinute() const { return minute; }

// Получение значения секунд
int Time::getSecond() const { return second; }

// Отображение времени в военном формате: HH:MM:SS
void Time::printMilitary() const

{
 cout << (hour < 10 ? "0:") << hour << ":"
 << (minute < 10 ? "0:") << minute << ":"
 << (second < 10 ? "0:") << second;
}
// Отображение времени 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");
}

```

**Рис. 7.8.** Сцепление вызовов функций-элементов (часть 3 из 5)

```

// FIG7_8.CPP
// Сцепление вызовов функций-элементов указателем this
#include <iostream.h>
#include "time6.h"

main()
{
 Time t;

 t.setHour(18).setMinute(30).setSecond(22);
 cout << "Военное время: ";
 t.printMilitary();
 cout << endl << "Стандартное время: ";
 t.printStandard();

 cout << endl << endl << "Новое стандартное время: ";
 t.setTime(20, 20, 20).printStandard();
 cout << endl;
 return 0;
}

```

**Рис. 7.8.** Сцепление вызовов функций-элементов (часть 4 из 5)

```
Военное время: 18:30:22
Стандартное время: 6:30:22PM

Новое стандартное время: 8:20:20PM
```

Рис. 7.8. Сцепление вызовов функций-элементов (часть 5 из 5)

## 7.6. Динамическое распределение памяти с помощью операций new и delete

Операции **new** и **delete** обеспечивают более удобные средства для реализации динамического распределения памяти (для любых встроенных или определенных пользователем типов) по сравнению с вызовами функций **malloc** и **free** в С. Рассмотрим следующее предложение:

```
TypeName *typeNamePtr;
```

В Си ANSI, чтобы динамически создать объект типа **TypeName**, вы должны написать:

```
typeNamePtr = malloc(sizeof());
```

Это требует вызова функции **malloc** и явной ссылки на операцию **sizeof**. В версии С, предшествующей С ANSI, вы должны были бы также привести тип указателя, возвращенного **malloc**, операцией приведения типа (**TypeName \***). Функция **malloc** не обеспечивает никакого метода инициализации выделяемого блока памяти. В С++ вы просто пишете

```
typeNamePtr = new TypeName;
```

Операция **new** автоматически создает объект соответствующего размера, вызывает конструктор объекта и возвращает указатель правильного типа. Если **new** не в состоянии найти необходимое пространство в памяти, она возвращает указатель 0.

Чтобы освободить пространство, выделенное ранее для этого объекта, в С++ вы должны использовать операцию **delete** в следующем виде:

```
delete typeNamePtr;
```

C++ позволяет вам использовать инициализатор для только что созданного объекта

```
float *thingPtr = new float (3.14159);
```

который задает начальное значение вновь созданному объекту типа **float**, равное 3.14159.

Массив можно создать и присвоить его **chessBoardPtr** в следующем виде:

```
int *chessBoardPtr = new int{8}{8};
```

Этот массив может быть уничтожен с помощью оператора

```
delete {} chessBoardPtr;
```

Как мы увидим, использование `new` и `delete` вместо `malloc` и `free` дает и другие преимущества. В частности, `new` автоматически активизирует конструктор, а `delete` автоматически активизирует деструктор класса.

### Типичная ошибка программирования 7.8

Смешивание способов динамического распределения памяти в стиле `new-delete` со стилем `malloc-free`: пространство, созданное с помощью `malloc`, не может быть освобождено с помощью `delete`; объекты, созданные с помощью `new`, не могут быть уничтожены с помощью `free`.

### Хороший стиль программирования 7.3

Хотя программы на C++ могут поддерживать память, выделяемую с помощью `malloc` и уничтожаемую с помощью `free`, и объекты, создаваемые с помощью `new` и уничтожаемые с помощью `delete`, лучше использовать только `new` и `delete`.

## 7.7. Статические элементы класса

Обычно каждый объект класса имеет свою собственную копию всех данных-элементов класса. Но в определенных случаях во всех объектах класса должна фигурировать только одна копия некоторых данных-элементов для всех объектов класса. Для этих и других целей используются статические данные-элементы, которые содержат информацию «для всего класса». Обявление статических элементов начинается с ключевого слова `static`.

### Совет по повышению эффективности 7.3

Если вам достаточно единственной копии данных, используйте для сокращения затрат памяти статические данные-элементы.

Хотя может показаться, что статические элементы данные похожи на глобальные переменные, тем не менее они имеют областью действия класс. Статические элементы могут быть открытыми, закрытыми или защищенными (`protected`). Статическим данным-элементам можно задать начальные значения *один* (*и только один*) раз в области действия файла. Доступ к открытым статическим элементам класса возможен посредством любого объекта класса или посредством имени класса, с помощью бинарной операции разрешения области действия. Закрытые и защищенные статические элементы класса должны быть доступны открытым функциям-элементам этого класса или друзьям класса. Статические элементы класса существуют даже тогда, когда не существует никаких объектов этого класса. Чтобы в этом случае обеспечить доступ к открытому статическому элементу, просто поставьте перед элементом данных имя класса и бинарную операцию разрешения области действия. Для обеспечения доступа в указанном случае к закрытому или защищенному элементу класса должна быть предусмотрена открытая статическая функция-элемент, которая должна вызываться с добавлением перед ее именем имени класса и бинарной операции разрешения области действия.

Программа на рис. 7.9 демонстрирует использование закрытого статического элемента данных и открытой статической функции-элемента. Элементу данных `count` задается нулевое начальное значение в области действия файла с помощью оператора

```
int Employee::count = 0;
```

Элемент данных `count` обслуживает подсчет количества объектов класса `Employee`, которые были созданы. Если объекты класса `Employee` существуют, элемент `count` может быть вызван посредством любой функции-элемента объекта `Employee` (в данном примере посредством как конструктора, так и деструктора). Если никаких объектов класса `Employee` не существует, элемент `count` также может быть вызван, но только посредством вызова статической функции-элемента `getCount` следующим образом:

```
Employee::getCount()
```

В этом примере функция `getCount` использована для определения текущего числа созданных объектов класса `Employee`. Отметим, что когда в программе еще не создано ни одного объекта, используется вызов функции `Employee::getCount()`. Но когда объекты уже созданы, функция `getCount` может быть вызвана из одного из объектов оператором

```
e1Ptr->getCount()
```

```
// ENPLOY1.H
// Класс employee
#ifndef EMPLOY1_H
#define EMPLOY1_H

class Employee {
public:
 Employee(const char*, const char*); // конструктор
 ~Employee(); // деструктор
 const char *getFirstName() const; // возвращает имя
 const char *getLastName() const; // возвращает фамилию
 // статическая функция-элемент
 static int getCount(); // возвращает число
 // созданных объектов

private:
 char *firstName;
 char *lastName;

 // статический элемент данных
 static int count; // число созданных объектов
};

#endif
```

Рис. 7.9. Использование статического элемента данных для подсчета количества объектов класса (часть 1 из 5)

```

// EMPLOY1.CPP
// Определения функций-элементов класса Employee
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ1.h"

// Задание начального значения элемента данных
int Employee::count = 0;

// Определение статической функции-элемента, которая
// возвращает количество созданных объектов.
int Employee::getCount() { return count; }

// Конструктор динамически выделяет память для
// имени и фамилии и использует strcpy, чтобы
// скопировать имя и фамилию в объект.
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);

 ++count; // увеличение статического счетчика
 // служащих
 cout << "Конструктор Employee для " << firstName
 << ' ' << lastName << " вызван." << endl;
}

// Деструктор освобождает динамически выделенную память
Employee::~Employee()
{
 cout << "~Employee() вызван для " << firstName
 << ' ' << lastName << endl;
 delete [] firstName; // освобождение памяти
 delete [] lastName; // освобождение памяти
 --count; // уменьшение статического
 // счетчика служащих
}

// Возвращение имени служащего
const char *Employee::getFirstName() const
{
 // Const перед возвращаемым типом предотвращает изменение
 // клиентом закрытых данных. Клиент должен скопировать
 // возвращенную строку перед тем, как деструктор освободит
 // динамическую память; это позволит избежать неопределенного
 // указателя.
 return firstName;
}

```

**Рис. 7.9.** Использование статического элемента данных для подсчета количества объектов класса (часть 2 из 5)

```

// EMPLOY1.CPP: Продолжение
// Возвращение фамилии служащего
const char *Employee::getLastName() const
{
 // Const перед возвращаемым типом предотвращает изменение
 // клиентом закрытых данных. Клиент должен скопировать
 // возвращенную строку перед тем, как деструктор освободит
 // динамическую память; это позволит избежать
 // неопределенного указателя.
 return lastName;
}

```

**Рис. 7.9.** Использование статического элемента данных для подсчета количества объектов класса (часть 3 из 5)

```

// FIG7_9.CPP
// Драйвер для проверки класса employee
#include <iostream.h>
#include "employ1.h"

main()
{
 cout << "Количество служащих перед созданием объектов равно "
 << Employee:: getCount() << endl; // используется
 // имя класса
 Employee *e1Ptr = new Employee("Susan", "Baker");
 Employee *e2Ptr = new Employee("Robert", "Jones");

 cout << "Количество служащих после создания объектов равно "
 << e1Ptr->getCount() << endl;

 cout << endl << "Служащий 1: "
 << e1Ptr->getFirstName()
 << " " << e1Ptr->getLastName()
 << endl << "Служащий 2: "
 << e2Ptr->getFirstName()
 << " " << e2Ptr->getLastName() << endl << endl;

 delete e1Ptr; // освобождение памяти
 delete e2Ptr; // освобождение памяти

 cout << "Количество служащих после удаления равно "
 << Employee:: getCount() << endl;
return 0;
}

```

**Рис. 7.9.** Использование статического элемента данных для подсчета количества объектов класса (часть 4 из 5)

Функция-элемент тоже может быть объявлена как **static**, если она не должна иметь доступ к нестатическим элементам класса. В отличие от нестатических функций-элементов статическая функция-элемент не имеет указателя **this**, потому что статические данные-элементы и статические функции-элементы существуют независимо от каких-либо объектов класса.

```
Количество служащих перед созданием объектов равно 0
Конструктор Employee для Susan Baker вызван.
Конструктор Employee для Robert Jones вызван.
Количество служащих после создания объектов равно 2
```

```
Служащий 1: Susan Baker
Служащий 2: Robert Jones
~Employee() вызван для Susan Baker
~Employee() вызван для Robert Jones
Количество служащих после удаления равно 0
```

**Рис. 7.9.** Использование статического элемента данных для подсчета количества объектов класса (часть 5 из 5)

Отметим использование `assert` в функции конструкторе `Employee`, функции `getFirstName` и функции `getLastName`. Утилита `assert`, определенная в заголовочном файле `assert.h`, проверяет значение выражения. Если значение выражения равно 0 (ложь), то `assert` печатает сообщение об ошибке и вызывает функцию `abort` (из общей библиотеки утилит — `stdlib.h`), которая завершает выполнение программы. Это полезное отладочное средство для проверки, имеет ли переменная правильное значение. В этой программе `assert` определяет, способна ли операция `new` осуществить динамическое выделение необходимого объема памяти. Например, в функции конструкторе `Employee` следующая строка (называемая также *оператором контроля*)

```
assert(firstName != 0);
```

проверяет указатель `firstName`, чтобы определить, не равен ли он 0. Если условие в операторе контроля истинно, то программа продолжается без прерывания. Если же это условие ложно, то выдается сообщение об ошибке, содержащее номер строки, проверяемое условие, печатается имя файла, в котором проявил себя оператор контроля, и программа завершается. Программист может сосредоточить свое внимание на этой части кода, чтобы найти причину ошибки. В главе 13 «Обработка исключений» мы узнаем более совершенные методы работы с ошибками во время выполнения.

Операторы контроля не надо удалять из программы после окончания отладки. Когда операторы контроля больше не нужны для отладки программы, в начале файла программы вставляется строка

```
#define NDEBUG
```

Это приводит к тому, что препроцессор игнорирует все операторы контроля, что удобнее, чем удалять каждый такой оператор контроля вручную.

### Типичная ошибка программирования 7.9

Ссылка на указатель `this` внутри статической функции-элемента.

### Типичная ошибка программирования 7.10

Обявление статической функции-элемента как `const`.

### Замечание по технике программирования 7.7

Статические данные-элементы и статические функции-элементы существуют и могут быть использованы, даже если не создано никаких объектов соответствующего класса.

Заметим, что реализации функций `getFirstName` и `getLastName` возвращают клиенту класса постоянные указатели на символьные строки. Если клиент желает сохранить копию имени или фамилии, он должен скопировать динамически распределенную область памяти объекта класса `Employee` после того, как получит от объекта постоянный указатель на символьную строку. Заметим, что можно также реализовать `getFirstName` и `getLastName` так, что от клиента будет требоваться передавать каждой функции массив символов и его размер. Тогда функции могли бы копировать имя и фамилию в массив символов, который передал им клиент.

## 7.8. Абстракция данных и скрытие информации

Обычно классы скрывают детали своей реализации от клиентов класса. В качестве примера скрытия информации рассмотрим структуру данных, называемую *стек*.

Представьте себе стек в виде стопки тарелок. Когда тарелка ставится на стопку, она всегда помещается на ее вершину (это называется *поместить в стек* — *pushing onto the stack*), а когда тарелка убирается из стопки, то всегда убирается тарелка с ее вершины (это называется *вытолкнуть из стека* — *popping off the stack*). Стеки известны как структуры данных типа *последним вошел — первым вышел* (*last-in, first-out* — *LIFO*) — последний элемент, помещенный (вставленный) в стек, является первым элементом, выталкиваемым (удаляемым) из стека.

Программист может создать класс стек и скрыть от его клиентов реализацию стека. Стеки можно легко реализовать с помощью массивов и других способов (таких как связные списки — смотри главу 15 «Структуры данных»). Клиенту класса стек не нужно знать, как реализован стек. Клиенту только надо знать, что когда он разместил элементы данных в стеке, то к ним можно обращаться только в последовательности *последним вошел — первым вышел*. Такой подход называется *абстракция данных*, а классы C++ определяют *абстрактные типы данных* (АТД). Хотя может оказаться, что пользователь знает детали реализации класса, но он может писать программу, не обращая внимания на эти детали. Это означает, что какой-то класс, например, тот, который реализует стек и его операции `push` (поместить) и `pop` (вытолкнуть), можно заменить другой его версией, не затрагивая остальной части системы, пока не изменен открытый интерфейс этого класса.

Задача языка высокого уровня — создать представление, удобное для использования программистом. Не существует единственного приемлемого стандартного представления и это одна из причин того, что существует так много языков программирования. Объектно-ориентированное программирование на C++ дает еще одно представление.

Большинство языков программирования делает акцент на действия. В этих языках данные существуют для поддержки действий, необходимых программе. Так или иначе, данные «менее интересны», чем действия. Данные в этих языках «негибки». Существует всего несколько встроенных типов данных и создание программистом своих собственных новых типов данных представляет определенные трудности.

Этот взгляд изменился с появлением C++ и объектно-ориентированного стиля программирования. C++ повышает значение данных. Основная дея-

тельность при работе с C++ заключается в создании новых типов данных (т.е. классов) и представлении взаимодействия между объектами этих типов данных.

Для продвижения в этом направлении среда языков программирования нуждается в формализации некоторых записей, относящихся к данным. Мы рассматриваем формализацию как запись абстрактных типов данных (АТД). АТД уделяют сейчас так же много внимания, как структурному программированию два десятилетия назад. АТД не заменяют структурное программирование. Скорее, АТД обеспечивают дополнительную формализацию, которая может улучшить процесс разработки программ.

Что такое абстрактный тип данных? Рассмотрим встроенный тип `int`. В голову приходит целое число в математике, однако `int` в компьютере — это не то же самое, что целое в математике. В частности, компьютерные значения `int` обычно жестко ограничены по размеру. Например, на 32-разрядной машине значение `int` может быть ограничено диапазоном от +2 миллиардов до -2 миллиардов. Если результат вычислений выходит из этого диапазона, происходит ошибка и машина реагирует каким-либо машинно-ориентированным способом, включая возможность получения «втихомолку» неправильного результата. Для математических целых чисел этой проблемы не существует. Таким образом, компьютерная запись `int` на самом деле лишь приблизительно соответствует реально существующим целым числам. То же самое справедливо и по отношению к числам с плавающей десятичной запятой `float`.

Приближением является даже тип `char`; значения `char` обычно представляют собой 8-битовые образы, ничего общего не имеющие с символами, которые они отображают, такими как заглавная буква Z, строчная z, знак доллара (\$), цифра (5) и т.д. Значения типа `char` в большинстве компьютеров жестко ограничены по сравнению с диапазоном реально существующих символов. 7-битовый набор символов ANSI обеспечивает представление лишь 127 различных значений символов. Это совершенно неадекватно представлению таких языков как японский и китайский, которые требуют тысяч символов.

Существует точка зрения, что встроенные типы данных, обеспечиваемые такими языками программирования, как C++, на самом деле являются только аппроксимациями или моделями понятий и поведения реального мира. Мы взяли для иллюстрации этой точки зрения `int`, но вы можете рассмотреть и другие примеры. Типы, подобные `int`, `float`, `char` и т.д. — это примеры абстрактных типов данных. По существу, они представляют собой способы представления реально существующих понятий с некоторым допустимым уровнем точности внутри компьютерной системы.

Абстрактные типы данных на самом деле охватывают два понятия, а именно, представление данных и операции, которые разрешены над этими данными. Например, запись `int` определяет в C++ операции сложения, вычитания, умножения, деления и модуля, но деление на нуль не определено; эти разрешенные операции выполняются способом, чувствительным к параметрам машины, таким, как размер фиксированного слова используемой компьютерной системы. Другим примером является запись отрицательных целых чисел, для которых операции и представление данных ясны, но операция вычисления квадратного корня из отрицательного числа не определена. В C++ программист для реализации абстрактных типов данных использует классы.

### 7.8.1. Пример: абстрактный тип данных массив

Мы обсуждали массивы в главе 4. Массив — это не более чем указатель и некоторая область памяти. Если программист осторожен и нетребователен, для выполнения операций с массивами достаточно этих примитивных знаний. Существует много операций, которые хотелось бы выполнять с массивами, однако они не встроены в C++. С помощью классов C++ программист может создать АТД массив, который предпочтительнее «сырых» массивов. Класс массив (такой класс будет создан в главе 8) может обеспечивать ряд новых полезных возможностей, таких как

- проверка диапазона индексов;
- произвольный выбор диапазона индексов;
- присваивание массива;
- ввод и вывод массива;
- массивы, которые знают свой размер.

Недостаток здесь заключается в том, что мы создаем заказной, нестандартный тип данных, который не доступен точно в таком виде в большинстве реализаций C++. Впрочем, использование C++ и объектно-ориентированного программирования быстро возрастает. Решающим моментом является то, что работы по профессиональному программированию развиваются в направлении крупномасштабной стандартизации и распространения библиотек классов для более полной реализации потенциала объектной ориентации.

C++ имеет небольшой набор встроенных типов. Абстрактные типы данных расширяют базу языка программирования.

#### Замечание по технике программирования 7.8

Программист имеет возможность создавать новые типы, используя формализм классов. Эти новые типы можно применять так же, как и встроенные типы данных. Поэтому C++ является расширяемым языком. Несмотря на возможность легко расширять его с помощью новых типов, базовый язык сам по себе остается неизменным.

Новые АТД, созданные в средах C++, могут быть собственностью отдельных людей, небольших групп или компаний. АТД можно помещать в стандартные библиотеки классов с целью их широкого распространения. Это не способствует стандартизации, хотя фактически вопрос о стандартизации возникает. В полном объеме значение C++ будет реализовано лишь тогда, когда станут широко доступны важные и стандартизованные библиотеки классов. Необходимы формальные процедуры поддержки разработки стандартизованных библиотек. В Соединенных Штатах такая стандартизация часто осуществляется при участии ANSI — Американского Национального Института Стандартов. ANSI непрерывно развивает стандартную версию C++. Независимо от того, каким образом в конце концов появляются эти библиотеки, читатель, который знает C++ и объектно-ориентированное программирование, будет готов овладеть достоинствами быстрой компонентно-ориентированной разработки программного обеспечения благодаря возможностям библиотек АТД.

## 7.8.2. Пример: абстрактный тип данных строка

C++ — намеренно ограниченный язык, обеспечивающий программиста только заготовками возможностей построения систем широкого профиля. Язык спроектирован так, чтобы минимизировать накладные расходы, связанные с производительностью. C++ подходит для проектирования как приложений, так и систем, а последнее накладывает чрезвычайные требования на эффективность программ. Конечно, можно было бы включить абстрактный тип данных строку в число встроенных типов данных C++. Вместо этого язык был спроектирован так, чтобы предоставить механизмы построения и реализации строковых абстрактных типов данных с помощью классов. Мы создадим свой собственный АТД строки в главе 8.

## 7.8.3. Пример: абстрактный тип данных очередь

Каждый из нас время от времени стоит в *очереди*: в магазине, на бензоколонке, на автобусной остановке, к железнодорожной кассе, а студенты очень хорошо знают очередь на регистрацию курсов, которые они хотят изучить. Множество очередей используется внутри компьютерных систем, так что нам нужны программы, которые моделируют очереди и их функционирование.

Очередь представляет собой хороший пример абстрактного типа данных. Очередь предлагает своим клиентам четко определенное поведение. Клиенты ставят некие элементы в очередь по одному за раз, используя операцию *поставить в очередь*, и получают их по одному за раз по запросу, используя операцию *исключить из очереди*. В принципе очередь может быть бесконечно длинной. Реальная очередь, конечно, ограничена. Элементы возвращаются из очереди в соответствии с дисциплиной — *первый вошел — первый вышел* (*first-in, first-out — FIFO*), т.е. первый элемент, вставленный в очередь, первым же ее покидает.

Очередь скрывает внутреннее представление данных, которое как-то отражает процесс ожидания элементов в очереди. Она предлагает своим клиентам набор операций, а именно — *поставить в очередь* и *исключить из очереди*. Клиентам не надо обращать внимание на способ реализации очереди. Клиенты просто хотят работать с очередью «как объявлено». Когда клиент ставит в очередь новый элемент, очередь должна принять этот элемент и разместить его внутри себя в какой-то структуре данных типа *первый вошел — первый вышел*. Когда клиент хочет получить следующий элемент из начала очереди, очередь должна взять этот элемент из внутреннего представления и выпустить его во внешний мир в соответствии с дисциплиной FIFO, т.е. элемент, который находился в очереди дольше всех, должен быть следующим, возвращенным очередной операцией *исключить из очереди*.

АТД очередь гарантирует целостность своей внутренней структуры данных. Клиенты могут не манипулировать непосредственно этой структурой данных. Только АТД очередь имеет доступ к своим внутренним данным. Клиенты могут вызывать лишь разрешенные операции для их выполнения над представлением данных; операции, которыми открытый интерфейс АТД не обеспечен, соответствующим образом скрыты в АТД. К ним могли бы относиться выдача сообщений об ошибках, прекращение выполнения или просто игнорирование требований операций.

## 7.9. Классы контейнеры и итераторы

К наиболее популярным типам классов относятся *классы контейнеры* (называемые также *классы совокупностей*), т.е. классы, спроектированные для хранения в них совокупностей объектов. Классы контейнеры обычно снабжены такими возможностями, как вставка, удаление, поиск, сортировка, проверка наличия элемента в классе и тому подобное. Массивы, стеки, очереди, связные списки — все это примеры классов контейнеров.

Принято ассоциировать *объекты итераторы*, или, короче — *итераторы*, с классами контейнерами. Итератор — это объект, который возвращает следующий элемент совокупности (или определяет некоторое действие над следующим элементом совокупности). Когда написан итератор класса, легко получить следующий элемент этого класса. Итераторы обычно пишутся как друзья классов, с которыми они работают. Это предоставляет итераторам возможность прямого доступа к закрытым данным этих классов. Подобно тому, как книга, читаемая несколькими людьми, могла бы иметь в себе сразу несколько закладок, класс контейнер может иметь несколько одновременно работающих итераторов. Каждый итератор поддерживает свою собственную «позицию» информации.

## 7.10. Размышления об объектах: использование композиции и динамического управления объектом в модели лифта

В главах с 2 по 5 вы проектировали модель лифта, а в главе 6 начали программирование этой модели. На протяжении главы 7 мы обсуждали методы, необходимые для завершения полной рабочей модели лифта. В частности, мы обсудили методику динамического управления объектами, которая позволяет использовать `new` и `delete` для создания и уничтожения объектов, необходимых для функционирования модели лифта. Мы обсудили также композицию, которая позволяет создавать классы, содержащие в качестве элементов другие классы. Композиция позволяет вам создать класс здание, который содержит лифт и этажи, и, в свою очередь, создать класс лифт, который содержит кнопки.

### Лабораторное задание 6 по лифту

1. Каждый раз, когда в модели появляется очередной пассажир, вы должны использовать `new`, чтобы создать представляющий его объект Пассажир. Заметим, что `new` активизирует конструктор создаваемого объекта и, конечно, конструктор должен задать объекту соответствующие начальные значения. Каждый раз, когда пассажир покидает модель (после выхода из лифта), вы должны использовать `delete`, чтобы уничтожить объект пассажира и восстановить память, занятую этим объектом.
2. Перечислите отношения композиции между классами, которые вы реализовали для вашей модели лифта. Модифицируйте описания классов, которые вы создали в разделе «Размышления об объектах» в главе 6, чтобы отразить эти отношения композиции.

3. Завершите реализацию работающей моделирующей программы. В последующих главах мы подскажем, как расширить модель лифта.

## Резюме

- Ключевое слово `const` указывает, что объект константный — его нельзя изменять.
  - Компилятор C++ не позволяет вызывать неконстантную функцию-элемент константного объекта.
  - Функция указывается как `const` и в ее объявлении, и в ее описании.
  - Константная функция-элемент может быть перегружена неконстантным вариантом. Выбор того, какая из перегруженных функций-элементов будет использоваться, осуществляется компилятором автоматически в зависимости от того, был объявлен объект как `const` или нет.
  - Константный объект должен получить начальные значения в своем объявлении.
  - Если класс содержит константные данные-элементы, конструктор этого класса должен быть обеспечен инициализаторами элементов.
  - Классы могут быть композицией объектов других классов.
  - Объекты-элементы создаются в том порядке, в котором они объявлены, и до того, как будут созданы объекты включающего их класса.
  - Если объект-элемент не имеет инициализатора элементов, вызывается конструктор объекта-элемента с умолчанием.
  - Дружественная функция класса — это функция, определенная вне класса, но имеющая право доступа к элементам класса, объявленным как `private` и `protected`.
  - Объявление дружественности может помещаться в любом месте определения класса.
  - Каждая перегруженная функция, предназначенная быть другом, должна быть явно объявлена как друг класса.
  - Каждый объект поддерживает указатель на самого себя, называемый указателем `this`, который является неявным аргументом во всех ссылках на элементы внутри этого объекта. Указатель `this` неявно используется при ссылках и на функции-элементы, и на данные-элементы объекта.
  - Каждый объект может определить свой собственный адрес путем использования ключевого слова `this`.
  - Указатель `this` можно использовать явно, но чаще он используется неявно.
  - Операция `new` автоматически создает объект соответствующего размера и возвращает указатель правильного типа. Чтобы освободить область памяти, занимаемую этим объектом, используется операция `delete`.
  - Массив объектов можно размещать автоматически с помощью операции `new`, например, оператор
- ```
int *ptr = new int[100];
```

выделяет место для массива из 100 целых чисел и присваивает адрес начала массива указателю `ptr`. Этот массив может быть уничтожен оператором

```
delete [ ] ptr;
```

- Статические данные-элементы имеют только одну копию для всех объектов класса и содержат информацию «для всего класса». Объявление статического элемента начинается с ключевого слова `static`.
- Статические данные-элементы имеют областью действия класс.
- Статические элементы класса доступны через объект этого класса или по имени класса с использованием операции разрешения области действия (если этот элемент открытый).
- Функция-элемент может быть объявлена как `static`, если она не должна иметь доступ к нестатическим элементам класса. В отличие от нестатических функций-элементов статическая функция-элемент не имеет указателя `this`, потому что статические данные-элементы и статические функции-элементы существуют независимо от каких-либо объектов класса.

Терминология

бинарная операция разрешения	
области действия (::)	
вложенный класс	
деструктор	
деструктор по умолчанию	
динамические объекты	
друг класса	
дружественная функция	
инициализатор элемента	
итератор	
классы контейнеры	
композиция	
константная функция-элемент	
константный объект	
конструктор	
конструктор объекта-элемента	
конструктор по умолчанию	

область действия класс	
объект-элемент	
операция <code>delete</code>	
операция <code>delete []</code>	
операция <code>new</code>	
операция доступа к элементу (.)	
операция доступа к элементу по указателю (->)	
принцип наименьших привилегий	
расширяемость языка	
спецификаторы (метки) доступа к элементам	
статическая функция-элемент	
статические данные-элементы	
сцепленные вызовы	
функций-элементов	
указатель <code>this</code>	

Типичные ошибки программирования

- 7.1. Описание константной функции-элемента, которая изменяет данные-элементы объекта.
- 7.2. Описание константной функции-элемента, которая вызывает неконстантную функцию-элемент.
- 7.3. Вызов неконстантной функции-элемента для константного объекта.
- 7.4. Попытка изменить константный объект.
- 7.5. Нет инициализаторов константных данных-элементов.

- 7.6. Не предусмотрен конструктор с умолчанием для объекта-элемента, когда для этого объекта элемента не задан инициализатор элементов. Это может привести к тому, что объект-элемент не будет инициализирован.
 - 7.7. Попытка использовать операцию доступа к элементу (.) с указателем на объект (операцию доступа к элементу точка можно использовать только с объектом или со ссылкой на объект).
 - 7.8. Смешивание способов динамического распределения памяти в стиле `new-delete` со стилем `malloc-free`: пространство, созданное с помощью `malloc`, не может быть освобождено с помощью `delete`; объекты, созданные с помощью `new`, не могут быть уничтожены с помощью `free`.
 - 7.9. Ссылка на указатель `this` внутри статической функции-элемента.
- 7.10. Объявление статической функции-элемента как `const`.**

Хороший стиль программирования

- 7.1. Объявляйте как `const` все функции-элементы, которые предполагается использовать с константными объектами.
- 7.2. Помещайте объявления дружественности первыми в классе непосредственно после его заголовка и не предваряйте их каким-либо спецификатором доступа к элементам.
- 7.3. Хотя программы на C++ могут поддерживать память, выделяемую с помощью `malloc` и уничтожаемую с помощью `free`, и объекты, создаваемые с помощью `new` и уничтожаемые с помощью `delete`, лучше использовать только `new` и `delete`.

Советы по повышению эффективности

- 7.1. Инициализируйте объекты-элементы явно с помощью инициализаторов элементов. Это исключает накладные расходы связанные с повторной инициализацией объектов-элементов: первой при вызове конструктора с умолчанием объекта-элемента и второй при задании начальных значений объекта-элемента с помощью функции записи «`set`».
- 7.2. С целью экономии памяти для каждой функции-элемента существует только одна копия на класс и эта функция-элемент вызывается каждым объектом данного класса. С другой стороны, каждый объект имеет свою собственную копию данных-элементов класса.
- 7.3. Если вам достаточно единственной копии данных, используйте для сокращения затрат памяти статические данные-элементы.

Замечания по технике программирования

- 7.1. Объявление константного объекта помогает провести в жизнь принцип наименьших привилегий. Случайные попытки изменить объект отлавливаются во время компиляции и не вызывают ошибок во время выполнения.

- 7.2. Константная функция-элемент может быть перегружена неконстантным вариантом. Выбор того, какая из перегруженных функций-элементов будет использоваться, осуществляется компилятором автоматически в зависимости от того, был объявлен объект как `const` или нет.
- 7.3. Константные элементы класса (объекты и «переменные») должны получать начальные значения с помощью инициализаторов элементов. Присваивания недопустимы.
- 7.4. Одним из способов повторного использования программного обеспечения является композиция, когда класс включает в себя объекты других классов в качестве элементов.
- 7.5. Спецификаторы доступа к элементам `private`, `protected` и `public` не имеют отношения к объявлению дружественности, так что эти объявления дружественности могут помещаться в любом месте в описании класса.
- 7.6. Некоторые члены сообщества объектно-ориентированного программирования считают, что «дружественность» портит скрытие информации и ослабляет значения объектно-ориентированного подхода к проектированию.
- 7.7. Статические данные-элементы и статические функции-элементы существуют и могут быть использованы, даже если не создано никаких объектов соответствующего класса.
- 7.8. Программист имеет возможность создавать новые типы, используя формализм классов. Эти новые типы можно применять так же, как и встроенные типы данных. Поэтому C++ является расширяемым языком. Несмотря на возможность легко расширять его с помощью новых типов, базовый язык сам по себе остается неизменным.

Упражнения для самопроверки

- 7.1. Заполнить пробелы в следующих утверждениях:
 - а) Для задания начальных значений постоянных элементов класса используется ____ .
 - б) Функция, не являющаяся элементом, которая должна иметь доступ к закрытым данным-элементам класса, должна быть объявлена как ____ этого класса.
 - в) Операция ____ динамически выделяет память для объекта указанного типа и возвращает ____ на этот тип.
 - г) Константный объект должен быть ____ ; он не может быть изменен после своего создания.
 - д) ____ элемент данных имеет одну копию для всех объектов класса.
 - е) Функции-элементы объекта поддерживают указатель на объект, называемый указатель ____ .
 - ж) Ключевое слово ____ указывает, что объект или переменную нельзя изменить после задания им начальных значений.

- h) Если объект-элемент класса не снабжен инициализатором, вызывается ____ этого класса.
- i) Функция-элемент может быть объявлена как **static**, если она не должна иметь доступ к _____ элементам класса.
- j) Дружественные функции могут иметь доступ к элементам класса с доступом ____ и ____ .
- k) Объекты-элементы создаются ____, чем объект включающего их класса.
- l) Операция ____ освобождает память, выделенную перед этим с помощью **new**.

7.2. Найдите ошибку или ошибки в каждом из следующих фрагментов программ и объясните, как их исправить.

```
a) 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;
};

b) char *string;
    string = new char[20];
    free(string);
```

Ответы на упражнения для самопроверки

7.1. a) инициализатор элементов. b) друг (**friend**). c) **new**, указатель. d) инициализирован. e) Статический. f) **this**. g) **const**. h) конструктор с умолчанием. i) нестатическим. j) **private**, **protected**. k) раньше. l) **delete**.

7.2. a) Ошибка: определение класса **Example** имеет две ошибки. Первая заключена в функции **getIncrementedData**. Функция объявлена как **const**, но она изменяет объект.

Исправление: чтобы исправить первую ошибку, удалите ключевое слово **const** из описания функции **getIncrementedData**.

Ошибка: вторая ошибка заключена в функции **getCount**. Эта функция объявлена как **static**, так что ей не разрешен доступ к любым нестатическим элементам класса.

Исправление: чтобы исправить вторую ошибку, удалите строку выходных данных из определения функции **getCount**.

b) Ошибка: память, динамически выделенная с помощью **new**, освобождается функцией **free** стандартной библиотеки С.

Исправление: используйте операцию C++ `delete` для освобождения памяти. Динамическое распределение памяти в стиле С нельзя смешивать с операциями `new` и `delete` C++.

Упражнения

- 7.3. Что общее и в чем различие между динамическим выделением памяти в C++ с помощью операций `new` и `delete` и динамическим выделением памяти с помощью функций `malloc` и `free` стандартной библиотеки С.

- 7.4. Объясните понятие дружественности в C++. Объясните отрицательные стороны дружественности, о которых написано в книге.

- 7.5. Может ли правильное описание класса `Time` включать оба следующих конструктора?

```
Time (int h = 0, int m = 0, int s = 0);  
Time();
```

- 7.6. Что случится, если для конструктора или деструктора указать тип возвращаемого значения, даже `void`?

- 7.7. Создайте класс `Date` со следующими возможностями:

- а) Вывод дат в таких нескольких форматах:

DD YYYY
MM/DD/YY
июнь 14, 1992

- б) Используйте перегруженные конструкторы для создания объектов `Date` с начальными значениями дат в форматах пункта а).

- с) Создайте конструктор `Date`, который читает системную дату, используя стандартные библиотечные функции заголовочного файла `time.h`, и передает ее элементам `Date`.

В главе 8 мы сможем создавать операции для проверки равенства двух дат и для сравнения дат, чтобы определить, какая из дат предшествует другой.

- 7.8. Создайте класс `SavingsAccount` (хранение вкладов). Используйте статический элемент данных `annualInterestRate` (процентная ставка) для хранения информации о каждом вкладчике. Каждый элемент этого класса содержит закрытый элемент данных `savingsBalance`, указывающий сумму, которую вкладчик имеет на депозите. Напишите функцию-элемент `calculateMonthlyInterest` (расчет ежемесячного дохода), которая ежемесячно вычисляет доход путем деления на 12 произведения `balance` и `annualInterestRate`; этот доход должен прибавляться к `savingsBalance`. Напишите статическую функцию-элемент `modifyInterestRate` (изменение процентной ставки), которая задает `annualInterestRate` новое значение. Напишите программу драйвер для проверки класса `SavingsAccount`. Создайте два различных объекта `SavingsAccount` `saver1` и `saver2`, с балансами \$2000.00 и \$3000.00 соответственно. Установите `annualInterestRate` равным 3%, затем вычислите ежемесячный доход по вкладу и напечатайте новые балансы для каждого из вкладчиков. Затем установ-

вите `annualInterestRate` равным 4%, вычислите месячный доход по вкладу и напечатайте новые балансы для каждого из вкладчиков.

- 7.9. Создайте класс `IntegerSet` (множество целых). Каждый объект класса может вмещать целые в диапазоне от 0 до 100. Множество представлено внутренне как массив из нулей и единиц. Элемент массива `a[i]` равен 1, если целое `i` находится в множестве. Элемент массива `a[j]` равен 0, если целое `j` не находится в множестве. Конструктор по умолчанию инициализирует множество как пустое, т.е. множество, чье представление в виде массива содержит только нули.

Напишите функции-элементы для типичных операций над множествами. Например, функцию-элемент `unionOfIntegerSet`, которая создает третье множество, являющееся теоретико-множественным объединением двух существующих (т.е. элемент массива третьего множества устанавливается равным 1, если этот элемент равен 1 хотя бы в одном или обоих существующих множествах, и элемент массива третьего множества устанавливается равным 0, если этот элемент равен 0 в обоих существующих множествах).

Напишите функцию-элемент `intersectionOfIntegerSets`, которая создает третье множество, являющееся теоретико-множественным пересечением двух существующих наборов (т.е. элемент массива третьего множества устанавливается равным 0, если этот элемент равен 0 в одном или обоих существующих множествах, и элемент массива третьего множества устанавливается равным 1, если этот элемент равен 1 в обоих существующих множествах).

Напишите функцию-элемент `insertElement`, которая вставляет новое целое `k` в множество (устанавливает элемент `a[k]` равным 1). Напишите функцию-элемент `deleteElement`, которая удаляет из множества целое `m` (устанавливает элемент `a[m]` равным 0).

Напишите функцию-элемент `setPrint`, которая печатает множество в виде списка чисел, разделенных пробелами.

Напишите функцию-элемент `isEqualTo`, которая определяет, равны ли друг другу два множества.

Напишите дополнительный конструктор, который получал бы пять целых аргументов для инициализации множества. Если вы хотите передать множеству менее пяти значений, используйте значения по умолчанию остальных аргументов, равное -1.

Теперь напишите программу драйвер для проверки вашего класса `IntegerSet`. Создайте несколько объектов `IntegerSet`. Убедитесь, что все ваши функции-элементы работают соответствующим образом.

г л а в а

8

Перегрузка операций



Ц е л и

- Понять, как переопределять операции для работы с новыми типами.
- Понять, как преобразовывать объект из одного класса в другой.
- Усвоить, когда нужно, и когда не следует перегружать операции.
- Изучить несколько интересных классов, которые используют перегруженные операции.
- Создать абстрактные типы данных массив, строка и дата.

План

- 8.1 Введение
- 8.2 Основы перегрузки операций
- 8.3 Ограничения на перегрузку операций
- 8.4 Функции-операции как элементы класса и как дружественные функции
- 8.5 Перегрузка операций поместить в поток и взять из потока
- 8.6 Перегрузка унарных операций
- 8.7 Перегрузка бинарных операций
- 8.8 Учебный пример: класс массив
- 8.9 Преобразования типов
- 8.10 Учебный пример: класс строка
- 8.11 Перегрузка ++ и --
- 8.12 Учебный пример: класс дата

Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

8.1. Введение

В главах 6 и 7 мы познакомились с основами классов в C++ и понятием абстрактных типов данных (АТД). Манипуляции над объектами классов (то есть, экземплярами АТД) реализовывались путем посылки сообщений объектам (в виде вызовов функций-элементов). Запись этих вызовов функций громоздка для определенного рода классов, особенно математических классов. Для классов такого рода было бы удобно использовать богатый набор имеющихся в C++ встроенных операций для измененных объектов. В этой главе мы покажем, как предоставить операциям C++ возможность работать с объектами классов. Такой механизм называется *перегрузкой операций*. Это простой и естественный путь обогащения C++ новыми возможностями.

Операция << используется в C++ для многих целей: и как операция поместить в поток, и как операция сдвига влево. Это пример перегрузки операции. Подобным же образом перегружается операция >>; она используется и как операция взять из потока, и как операция сдвига вправо. Каждая из этих операций перегружена в библиотеке классов C++. Язык C++ сам по себе перегружает + и -. Эти операции выполняются по-разному, в зависимости от того, входят ли они в выражения целочисленной арифметики, арифметики с плавающей запятой или арифметики указателей.

C++ предоставляет программисту возможность перегружать большинство операций и делать их чувствительными к контексту, в котором они используются. Компилятор генерирует соответствующий код, основываясь на способе использования операции. Некоторые операции перегружаются часто, особенно операция присваивания и различные арифметические операции, такие как + и -. Работа, выполняемая перегруженными операциями, может быть также выполнена и с помощью явных вызовов функций, но запись операции обычно читать легче.

Мы обсудим, когда следует использовать перегрузку операции, а когда нет. Мы покажем, как перегружать операции, и дадим много законченных программ, использующих перегруженные операции.

8.2. Основы перегрузки операций

Программирование на C++ — процесс, чувствительный к типам и основанный на типах. Программист может использовать встроенные типы, а может определить и новые типы. Встроенные типы можно использовать с богатым набором операций C++. Операции обеспечивают программиста краткими средствами записи для выражения манипуляций с объектами встроенного типа.

Программист может также использовать операции с типами, определенными пользователем. Хотя C++ и не позволяет создавать новые операции, он все же позволяет перегружать существующие операции так, что при использовании этих операций с объектами классов они приобретают смысл, соответствующий новым типам. Это одно из наиболее мощных средств C++.

Замечание по технике программирования 8.1

Перегрузка операций способствует расширяемости C++, являясь, несомненно, одним из наиболее привлекательных свойств этого языка.

Хороший стиль программирования 8.1

Используйте перегрузку операций, если она делает программу более ясной по сравнению с применением явных вызовов функций для выполнения тех же операций.

Хороший стиль программирования 8.2

Избегайте чрезмерного или непоследовательного использования перегрузки операций, так как это может сделать программу непонятной и затруднит ее чтение.

Хотя перегрузка операций звучит экзотически, большинство программистов неявно регулярно ее используют. Например, операция сложения (+) выполняется для целых чисел, чисел с плавающей запятой и с удвоенной точностью совершенно по-разному. Но тем не менее сложение прекрасно работает с любыми типами — `int`, `float`, `double` и многими другими встроенными типами, потому что операция сложения (+) перегружена в самом C++.

Операции перегружаются путем составления описания функции (с заголовком и телом), как это вы обычно делаете, за исключением того, что в этом случае имя функции состоит из ключевого слова `operator`, после которого записывается перегружаемая операция. Например, имя функции `operator+` можно использовать для перегрузки операции сложения.

Чтобы использовать операцию над объектами классов, эта операция должна быть перегружена, но есть два исключения. Операция присваивания (=) может быть использована с каждым классом без явной перегрузки. По умолчанию операция присваивания сводится к *побитовому копированию* данных-элементов класса. Мы увидим вскоре, что такое побитовое копирование опасно для классов с элементами, которые указывают на динамически выделенные области памяти; для таких классов мы будем явно перегружать операцию присваивания. Операция адресации (&) также может быть использована с объектами любых классов без перегрузки; она просто возвращает адрес объекта в памяти. Но операцию адресации можно также и перегружать.

Перегрузка больше всего подходит для математических классов. Они часто требуют перегрузки значительного набора операций, чтобы обеспечить согласованность со способами обработки этих математических классов в реальной жизни. Например, было бы странно перегружать только сложение класса комплексных чисел, потому что обычно с комплексными числами используются и другие арифметические операции.

C++ — язык, богатый операциями. Те из программирующих на C++, кто понимает смысл и контекст каждой операции, сделают, вероятно, обоснованный выбор, когда придется перегружать операции для новых классов.

Цель перегрузки операций состоит в том, чтобы обеспечить такие же краткие выражения для типов, определенных пользователем, какие C++ обеспечивает с помощью богатого набора операций для встроенных типов. Однако, перегрузка операций не выполняется автоматически; чтобы выполнить требуемые операции, программист должен написать функции, осуществляющие перегрузки операций. Иногда эти функции лучше сделать функциями-элементами, иногда — функциями-друзьями, а случается, что лучше не делать их ни элементами, ни друзьями.

При чрезмерных злоупотреблениях перегрузкой операция + могла бы выполнять операцию квазивычитания, а операция / — операцию квазумножения, но такое использование перегрузки могло бы только дискредитировать программу.

Хороший стиль программирования 8.3

Перегружайте операции, чтобы они выполняли над объектами класса ту же функцию или близкие к ней функции, что и операции, выполняемые над объектами встроенных типов.

Хороший стиль программирования 8.4

Перед написанием программы на С++ с перегруженными операциями обратитесь к руководству по вашему компилятору С++, чтобы осознать разнообразные специфические ограничения и требования отдельных операций.

8.3. Ограничения на перегрузку операции

Большинство операций С++ перегружать можно. Они показаны на рис. 8.1. На рис. 8.2 показаны операции, которые перегружать нельзя.

Операции, которые могут быть перегружены									
+	-	*	/	%	^	&			
~	!	=	<	>	+=	-=	*=		
/=	%=	^=	&=	I=	<<	>>	>>=		
<<=	==	!=	<=	>=	&&		+		
--	->*	, ->	[]	()	new	delete	.		

Рис. 8.1. Операции, которые могут быть перегружены

Операции, которые не могут быть перегружены									
.	*	::	:	sizeof					

Рис. 8.2. Операции, которые не могут быть перегружены

Типичная ошибка программирования 8.1

Попытка перегрузить операцию, запрещенную для перегрузки.

Старшинство операций не может быть изменено перегрузкой. Это могло бы привести к щекотливым ситуациям, в которых операция перегружается таким образом, для которого установленное в языке старшинство не подходит. Однако с помощью скобок можно принудительно изменить последовательность оценки перегруженных операций в выражениях.

Ассоциативность операций не может быть изменена перегрузкой. С перегруженными операциями нельзя использовать аргументы по умолчанию.

Изменить количество operandов, которое берет операция, невозможно: перегруженные унарные операции остаются унарными, перегруженные бинарные операции остаются бинарными. В С++ не может быть перегружена единственная тернарная операция ?: . Каждая из операций &, *, + и — может иметь унарный и бинарный варианты; эти унарные и бинарные варианты могут перегружаться раздельно.

Создавать новые операции невозможно; перегружать можно только уже существующие операции. Это запрещает программисту использовать популярные нотации, подобные операции **, используемой в BASIC для экспоненты.

Типичная ошибка программирования 8.2

Попытка создавать новые операции.

Нельзя изменить с помощью перегрузки операции смысл работы операции с объектом встроенного типа. Программист, например, не может изменить смысл того, как с помощью + складываются два целых числа. Перегрузка операций применима только для работы с объектами типов, определенных пользователем, или со смесью объектов типов, определенных пользователем, и встроенных типов.

Типичная ошибка программирования 8.3

Попытка изменить работу операции с объектами встроенного типа.

Замечание по технике программирования 8.2

По меньшей мере один аргумент функции-операции должен быть объектом класса или ссылкой на объект класса. Это предохраняет программиста от изменения работы операции с объектом встроенного типа.

Перегрузка операций присваивания и сложения, разрешающая такие операторы, как

```
object2 = object2 + object1;
```

не означает, что операция += также перегружается с целью разрешить такие операторы, как

```
object2 += object1;
```

Это может быть достигнуто только путем явной перегрузки операции += для данного класса.

Типичная ошибка программирования 8.4

Предположение, что перегрузка операции (такой как +) автоматически перегружает связанные с ней операции (такие как +=). Операции можно перегружать только явно; неявной перегрузки не существует.

Хороший стиль программирования 8.5

Чтобы обеспечить согласованность связанных операций используйте одни из них для реализации других (т.е. используйте перегруженную операцию + для реализации перегруженной операции +=).

8.4. Функции-операции как элементы класса и как дружественные функции

Функции-операции могут быть, или не быть функциями-элементами; если функции не являются элементами, они обычно являются друзьями. Функции-элементы неявно используют указатель this, чтобы получить один

из своих аргументов в виде объекта класса. Этот аргумент должен быть указан явно в списке при вызове функции, не являющейся элементом.

При перегрузке операций (), [], -> или = функция перегрузки операции должна быть объявлена как элемент класса. Для других операций функции перегрузки операций могут не быть функциями-элементами (тогда они обычно являются друзьями).

Реализована ли функция-операция как функция-элемент или нет, операция в выражении реализуется одинаково. Так какая же реализация лучше?

Когда функция-операция реализована как функция-элемент, крайний левый (или единственный) операнд должен быть объектом того класса (или ссылкой на объект того класса), элементом которого является функция. Если левый операнд должен быть объектом другого класса илистроенного типа, такая функция-операция не может быть реализована как функция-элемент (мы будем это делать в разделе 8.5 при перегрузке << и >> как операций поместить в поток и взять из потока соответственно). Функция-операция, реализованная не как функция-элемент, должна быть другом, если эта функция должна иметь прямой доступ к закрытым или защищенным элементам этого класса.

Перегруженная операция << должна иметь левый операнд типа `ostream &` (такой, как `cout` в выражении `cout << classObject`), так что она не может быть функцией-элементом. Аналогично, перегруженная операция >> должна иметь левый операнд типа `istream &` (такой, как `cin` в выражении `cin << classObject`), так что она тоже не может быть функцией-элементом. К тому же каждая из этих перегруженных функций-операций может потребовать доступа к закрытым элементам данным объекта класса, являющегося входным или выходным потоком, так что эти перегруженные функции-операции делают иногда функциями-друзьями класса из соображений эффективности.

Совет по повышению эффективности 8.1

Можно было бы перегружать операцию не как элемент и не как дружественную функцию, но такая функция, нуждающаяся в доступе к закрытым или защищенным данным класса, потребовала бы использования функций `set` или `get`, предусмотренных открытым интерфейсом этого класса. Накладные расходы на вызовы этих функций могли бы вызвать ухудшение производительности.

Функции-элементы операций вызываются только в случае, если левый операнд бинарной операции или единственный операнд унарной операции являются объектом того класса, элементом которого является функция.

Другая причина того, что для перегрузки операции можно выбирать функцию, не являющуюся элементом, состоит в возможности сделать операцию коммутативной. Например, мы имеем объект `number` типа `long int` и объект `bigInteger1` класса `HugeInteger` (класса, в котором целые могут быть произвольно большими и не ограниченными размерами машинного слова используемого оборудования; класс `HugeInteger` разработан в упражнениях к данной главе). Операция сложения (+) создает временный объект `HugeInteger` как сумму `long int` и `HugeInteger` (как в выражении `number + bigInteger1`). Таким образом, мы требуем, чтобы операция сложения была коммутативной (как это обычно и есть). Проблема состоит в том, что объект класса должен находиться слева от знака сложения, если операция перегружена как функция-элемент. Поэтому мы перегружаем операцию как друга,

чтобы позволить `HugeInteger` находиться справа от знака сложения. Функция `operator+`, которая имеет дело с `HugeInteger`, расположенным только слева, может быть и функцией-элементом.

8.5. Перегрузка операций поместить в поток и взять из потока

C++ способен вводить и выводить стандартные типы данных, используя операцию поместить в поток `>>` и операцию взять из потока `<<`. Эти операции уже перегружены в библиотеках классов, которыми снабжены компиляторы C++, чтобы обрабатывать каждый стандартный тип данных, включая строки и адреса памяти. Операции поместить в поток и взять из потока можно также перегрузить для того, чтобы выполнять ввод и вывод типов, определенных пользователем. Программа на рисунке 8.3 демонстрирует перегрузку операций поместить в поток и взять из потока для обработки данных определенного пользователем класса телефонных номеров `PhoneNumber`. В этой программе предполагается, что телефонные номера вводятся правильно. Проверку ошибок мы оставляем для упражнений.

На рис. 8.3 функция-операция взять из потока (`operator>>`) получает как аргументы ссылку `input` типа `istream`, и ссылку, названную `num`, на определенный пользователем тип `PhoneNumber`; функция возвращает ссылку типа `istream`. Функция-операция (`operator>>>`) используется для ввода номеров телефонов в виде

(800) 555-1212

в объекты класса `PhoneNumber`. Когда компилятор видит выражение

```
cin >> phone
```

в `main`, он генерирует вызов функции

```
operator>>(cin, phone);
```

После выполнения этого вызова параметр `input` становится псевдонимом для `cin`, а параметр `num` становится псевдонимом для `phone`. Функция-операция использует функцию-элемент `getline` класса `istream`, чтобы прочитать как строки три части телефонного номера вызванного объекта класса `PhoneNumber` (`num` в функции-операции и `phone` в `main`) в `areaCode` (код местности), `exchange` (коммутатор) и `line` (линия). Функция `getline` детально объяснена в главе 11. Символы круглых скобок, пробела и дефиса пропускаются при вызове функции-элемента `ignore` класса `istream`, которая отбрасывает указанное количество символов во входном потоке (один символ по умолчанию). Функция `operator>>` возвращает ссылку `input` типа `istream` (т.е. `cin`). Это позволяет операциям ввода объектов `PhoneNumber` быть сцепленными с операциями ввода других объектов `PhoneNumber` или объектов других типов данных. Например, два объекта `PhoneNumber` могли бы быть введены следующим образом:

```
cin >> phone1 >> phone2;
```

Сначала было бы выполнено выражение `cin >> phone1` путем вызова `operator>>(cin, phone1);`

```
// FIG8_3.CPP
// Перегрузка операций поместить в поток и взять из потока.
#include <iostream.h>

class PhoneNumber{
    friend ostream &operator << (ostream &, const PhoneNumber &);
    friend istream &operator >> (istream &, PhoneNumber &);

private:
    char areaCode[4]; //трехцифровой код местности и нулевой символ
    char exchange[4]; //трехцифровой коммутатор и нулевой символ
    char line[5];     //четырехцифровая линия и нулевой символ
};

// Перегруженная операция поместить в поток
// (она не может быть функцией элементом).
ostream &operator<<(ostream &output, const PhoneNumber &num)
{
    output << "(" << num.areaCode << " ) "
        << num.exchange << "-" << num.line;

    return output;           // разрешает cout << a << b << c;
}

// Перегруженная операция взять из потока
istream &operator>> (istream &input, PhoneNumber &num)
{
    input.ignore();          //пропуск (
    input.getline(num.areaCode, 4); //ввод кода местности
    input.ignore(2);          //пропуск ) и пробела
    input.getline(num.exchange, 4); //ввод коммутатора
    input.ignore();          //пропуск дефиса (-)
    input.getline(num.line, 5); //ввод линии

    return input;            //разрешает cin >> a >>b >>c;
}

main()
{
    PhoneNumber phone;      // создание объекта phone

    cout << "Введите номер телефона в "
        << "виде (123) 456-7890:" << endl;

    // cin >> phone активизирует функцию operator>>
    // путем вызова operator>> (cin, phone).
    cin >> phone;

    // cout << phone активизирует функцию operator<<
    // путем вызова operator<< (cout, phone).
    cout << "Был введен номер телефона:" << endl
        << phone << endl;
    return 0;
}
```

Рис. 8.3. Определенные пользователем операции «поместить в поток» и «взять из потока» (часть 1 из 2)

```

Ведите номер телефона в виде (123) 456-7890:
(800) 555-1212
Был введен номер телефона:
(800) 555-1212

```

Рис. 8.3. Определенные пользователем операции «поместить в поток» и «взять из потока» (часть 2 из 2)

Этот вызов мог бы затем вернуть ссылку на `cin` как значение `cin >> phone1`, так что оставшаяся часть выражения была бы интерпретирована просто как `cin >> phone2`. Это было бы выполнено путем вызова

```
operator>>(cin, phone2);
```

Операция поместить в поток получает как аргументы ссылку `output` типа `ostream` и ссылку `pnum` на определенный пользователем тип `PhoneNumber` и возвращает ссылку типа `ostream`. Функция `operator<<` выводит на экран объекты типа `PhoneNumber`. Когда компилятор видит выражение

```
cout << phone
```

в `main`, он генерирует вызов функции

```
operator<<(cout, phone);
```

Функция `operator<<` выводит на экран части телефонного номера как строки, потому что они хранятся в формате строки (функция-элемент `getline` класса `istream` сохраняет нулевой символ после завершения ввода).

Заметим, что функции `operator>>` и `operator<<` объявлены в `class PhoneNumber` не как функции-элементы, а как дружественные функции. Эти операции не могут быть элементами, так как объект класса `PhoneNumber` появляется в каждом случае как правый operand операции; а для перегруженной операции, записанной как функция-элемент, operand класса должен появляться слева. Перегруженные операции поместить в поток и взять из потока должны объявляться как дружественные, если они должны иметь прямой доступ к закрытым элементам класса по соображениям производительности.

Замечание по технике программирования 8.3

Новые возможности ввода-вывода для типов, определенных пользователем, могут быть добавлены в C++ без изменения объявлений или закрытых данных-элементов для классов `ostream` и `istream`. Это еще один пример расширяемости языка программирования C++.

8.6. Перегрузка унарных операций

Унарную операцию класса можно перегружать как нестатическую функцию-элемент без аргументов, либо как функцию, не являющуюся элементом, с одним аргументом; этот аргумент должен быть либо объектом класса, либо ссылкой на объект класса. Функции-элементы, которые реализуют перегруженные операции, должны быть нестатическими, чтобы они могли иметь доступ к данным класса. Напомним, что статические функции-элементы могут иметь доступ только к статическим данным-элементам класса.

Далее в этой главе мы перегрузим унарную операцию `!`, чтобы проверять, пуст ли объект класса `String`. Если унарная операция, такая, как `!`, перегружена как нестатическая функция-элемент без аргументов и если `s` — объект класса `String` или ссылка на объект класса `String`, то, когда компилятор видит выражение `!s`, он генерирует вызов `s.operator!()`. Операнд `s` — это объект класса, для которого вызывается функция-элемент `operator!` класса `String`. Функция объявляется в описании класса следующим образом:

```
class String {
public:
    int operator!() const;
    ...
};
```

Унарная операция, такая, как `!`, может быть перегружена как функция с одним аргументом, не являющаяся элементом, двумя различными способами: либо с аргументом, который является объектом (это требует копирования объекта, чтобы побочные эффекты функции не оказывали влияния на исходный объект), либо с аргументом, который является ссылкой на объект (никакой копии исходного объекта при этом не делается, но все побочные эффекты этой функции оказывают влияние на исходный объект). Если `s` — объект класса `String` (или ссылка на объект класса `String`), то `!s` трактуется как вызов `operator!(s)`, активизирующий дружественную функцию, не являющуюся элементом класса `String`, но объявленную в нем следующим образом:

```
class String {
    friend int operator!(const String &);
    ...
};
```

Хороший стиль программирования 8.6

При перегрузке унарных операций предпочтительнее создавать функции-операции, являющиеся элементами класса, вместо дружественных функций, не являющихся элементами. Дружественных функций и дружественных классов лучше избегать до тех пор, пока они не станут абсолютно необходимыми. Использование друзей нарушает инкапсуляцию класса.

8.7. Перегрузка бинарных операций

Бинарную операцию можно перегружать как нестатическую функцию-элемент с одним аргументом, либо как функцию, не являющуюся элементом, с двумя аргументами (один из этих аргументов должен быть либо объектом класса, либо ссылкой на объект класса).

Далее в этой главе мы перегрузим бинарную операцию `+=`, указывающую на сцепление двух объектов-строк. Если бинарная операция `+=` перегружена как нестатическая функция-элемент класса `String` с одним аргументом и если `y` и `z` — объекты класса `String`, то `y += z` рассматривается компилятором как выражение `y.operator+=(z)`, активизирующее функцию-элемент `operator+=`, объявленную ниже:

```
class String {
public:
    String &operator+=(const String &);
    ...
};
```

Бинарная операция `+=` может быть перегружена как функция, не являющаяся элементом, с двумя аргументами, один из которых должен быть объектом класса или ссылкой на объект класса. Если `y` и `z` — объекты класса `String` или ссылки на объект класса `String`, то `y += z` рассматривается как вызов `operator+=(y,z)`, активизирующий объявленную ниже дружественную функцию `operator+=`, не являющуюся элементом:

```
class String {
    friend String &operator+=(String &, const String &0);
    ...
};
```

8.8. Учебный пример: класс массив

Запись массива в C++ является альтернативой указателям, так что массивы могут служить источником множества ошибок. Например, программа может легко «выйти за пределы» массива, поскольку C++ не проверяет, не вышли ли индексы из допустимых пределов. Массивы размера `n` должны иметь номера `0, ..., n-1`; иных индексов не может быть. Массив целиком не может быть введен или выведен сразу: каждый элемент массива должен быть считан или записан индивидуально. Два массива не могут быть сравнены друг с другом с помощью операций проверки на равенство или операций отношения. Когда массив передается функции общего назначения, обрабатывающей массивы произвольного размера, размер массива должен передаваться как дополнительный аргумент. Один массив не может быть присвоен другому с помощью операции присваивания. Эти и другие возможности, несомненно, кажутся «естественными» для работы с массивами, но C++ не обеспечивает таких возможностей. Однако, C++ обеспечивает средства для реализации этих возможностей посредством механизмов перегрузки операций.

В этом примере мы разработаем класс массив, который выполняет проверку диапазона, чтобы гарантировать, что индексы остаются в пределах границ массива. Класс допускает присваивание одного объекта массива другому с помощью операции присваивания. Объекты этого класса автоматически узнают свой размер, так что при передаче массива функции передавать отдельно размер массива в качестве аргумента не требуется. Массив можно целиком выводить или вводить с помощью операций поместить в поток и взять из потока соответственно. Сравнение массивов можно осуществить с помощью операций `==` и `!=`. Наш класс массив использует статический элемент, чтобы отследить количество объектов массивов, которые были созданы в программе. Этот пример отточит ваше понимание абстракции данных. Возможно, вам захочется существенно расширить этот класс массива. Создание класса представляет собой интересный, творческий процесс, стимулирующий интеллект.

Программа на рис. 8.4 демонстрирует класс `Array` и его перегруженные операции. Сначала мы проследим программу драйвер в `main`. Затем рассмотрим определение класса, каждую функцию-элемент класса и определения дружественных функций.

Статический элемент данных `arrayCount` класса `Array` содержит количество объектов, образованных во время выполнения программы. Программа начинается с использования статической функции-элемента `getArrayCount`,

которая дает возможность получить количество созданных ранее массивов. Далее программа создает два объекта класса **Array**: **integers1** с семью элементами и **integers2** с десятью элементами по умолчанию (значение по умолчанию указано конструктором **Array**). Чтобы получить новое количество объектов снова используется статическая функция-элемент **getArrayCount**. Функция-элемент **getSize** возвращает размер массива **integers1**. Программа выводит на экран размер массива **integers1**, затем выводит сам массив, используя перегруженную операцию поместить в поток, чтобы удостовериться, что конструктор задал элементам массива правильные начальные значения. Далее выводится размер массива **integers2**, а затем с помощью операции поместить в поток выводится сам массив.

```
// 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 с умолчанием
    Array(const Array &);                          //конструктор копии
    ~Array();                                       //деструктор
    int getSize() const;                           //возвращение размера
    const Array &operator=(const Array &);        //присваивание
                                                    //массивов
    int operator==(const Array &) const;          //проверка равенства
    int operator!=(const Array &) const;          //сравнение
                                                    //на неравенство
    int &operator[](int);                          //операция индексации
    static int getArrayCount();                    //возвращение числа
                                                    //экземпляров

private:
    int *ptr;                                      //указатель на первый элемент
                                                    //массива
    int size;                                       //размер массива
    static int arrayCount;                         //число экземпляров массивов
};

#endif
```

Рис. 8.4. 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 статического элемента данных с областью
// действия файл
int Array::arrayCount = 0;           // пока нет никаких объектов
// Конструктор с умолчанием класса Array
Array::Array(int arraySize)
{
    ++arrayCount;                  //прибавление одного объекта
    size = arraySize;              //по умолчанию размер равен 10
    ptr = new int[size];           //выделение пространства для массива
    assert(ptr != 0);              //завершение, если память не выделена

    for (int i = 0; i < size; i++)
        ptr[i] = 0;                //задание массиву начальных значений
}

// Конструктор копии класса Array
Array::Array(const Array &init)
{
    ++arrayCount;                  //прибавление одного объекта
    size = init.size;              //по умолчанию размер равен 10
    ptr = new int[size];           //выделение памяти для массива
    assert(ptr != 0);              //завершение, если память не выделена

    for (int i = 0; i < size; i++)
        ptr[i] = init.ptr[i];       //копирование init в объект
}

// Деструктор класса Array
Array::~Array()
{
    --arrayCount;                  //уменьшение на один объект
    delete [ ] ptr;                //возвращение области памяти массива
}

// Получение размера массива
int Array::getSize() const { return size; }

// Перегруженная операция присваивания
const Array &Array::operator=(const Array &right)
{
    if (&right != this) {         // проверка самоприсваивания
        delete [ ] ptr;           // возвращение области памяти
        size = right.size;        // изменение размера объекта
        ptr = new int[size];      // выделение памяти для копии массива
        assert(ptr != 0);          // завершение, если память не выделена

        for (int i = 0; i < size; i++)
            ptr[i] = right.ptr[i]; // массив копии в объект
    }

    return *this;                 // позволяет = y = z;
}

```

Рис. 8.4. Определения функций-элементов класса Array (часть 2 из 7)

```
// Проверка равенства двух массивов
// и возвращение значения 1, если равны, и значения 0,
// если не равны.
int Array::operator==(const Array &right) const
{
    if (size != right.size)
        return 0;                                // массивы разных размеров
    for (int i = 0; i < size; i++)
        if (ptr [i] != right.ptr [i] )
            return 0;                            // массивы не равны

    return 1;                                // массивы равны
}

// Проверка неравенства двух массивов и возвращение значения 1,
// если не равны, и значения 0, если равны.
int Array::operator!=(const Array &right) const
{
    if (size != right.size)
        return 1;                                // массивы разных размеров
    for (int i = 0; i < size; i++)
        if (ptr[i] != right.ptr[i])
            return 1;                            // массивы не равны

    return 0;                                // массивы равны
}

// Перегруженная операция индексации
int &Array::operator[](int subscript)
{
    // проверка ошибочного выхода индекса из диапазона
    assert(0 <= subscript && subscript < size);

    return ptr[subscript]; // возвращение ссылки создает L-величину
}

// Возвращение числа возникших объектов Array
int Array::getArrayCount( ) { return arrayCount; }

// Перегруженная операция ввода для класса Array;
// вводит значения всего массива
istream &operator>>(istream &input, Array &a)
{
    for (int i = 0; i < a.size; i++)
        input >> a.ptr[i];

    return input;                           //позволяет cin >> x >> y;
}
```

Рис. 8.4. Определения функций-элементов класса Array (часть 3 из 7)

```
// 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; //позволяет cout << x << y;
}
```

Рис. 8.4. Определения функций-элементов класса Array (часть 4 из 7)

```
// FIG8_4.CPP
// Драйвер для простого класса Array
#include <iostream.h>
#include "array1.h"

main()
{
    // еще нет никаких объектов
    cout << "Количество созданных массивов = "
        << Array::getArrayCount() << endl;

    // создание двух массивов и печать количества Array
    Array integers1(7), integers2;
    cout << "Количество созданных массивов = "
        << Array::getArrayCount() << endl << endl;

    // печать размера и содержимого integers1
    cout << "Размер массива integers1 равен "
        << integers1.getSize() << endl
        << "Массив после задания начальных значений: " << endl
        << integers1 << endl;

    // печать размера и содержимого integers2
    cout << "Размер массива integers2 равен "
        << integers2.getSize() << endl
        << "Массив после задания начальных значений: " << endl
        << integers2 << endl;

    // ввод и печать integers1 и integers2
    cout << "Введите 17 целых чисел:" << endl;
    cin >> integers1 >> integers2;
    cout << "После ввода массивы содержат:" << endl
        << "integers1: " << integers1
        << "integers2: " << integers2 << endl;

    // использование перегруженной операции проверки неравенства (!=)
    cout << "Оценка: integers1 != integers2" << endl;
    if (integers1 != integers2)
        cout << "Они не равны" << endl;
```

Рис. 8.4. Драйвер для класса Array (часть 5 из 7)

```

//создание массива integers3, использующего integers1
//для задания начальных значений; печать размера и содержимого
Array integers3(integers1);

cout << endl << "Размер массива integers3 равен "
    << integers3.getSize() << endl
    << "Массив после задания начальных значений:" << endl
    << integers3 << endl;

// использование перегруженной операции присваивания (=)
cout << "Присваивание массива integers2 массиву integers1:"
    << endl;
integers1 = integers2;
cout << "integers1: " << integers1
    << "integers2: " << integers2 << endl;

// использование перегруженной операции проверки равенства (==)
cout << "Оценка: integers1 == integers2" << endl;
if (integers1 == integers2)
    cout << "Они равны" << endl << endl;

// использование перегруженной операции индексации
// для создания R-величины
cout << "integers1[5] равен " << integers1[5] << endl;

// использование перегруженной операции индексации
// для создания L-величины
cout << "Присваивание 1000 элементу integers1[5]" << endl;
integers1[5] = 1000;
cout << "integers1: " << integers1 << endl;

// попытка использовать индекс вне диапазона
cout << "Попытка присвоить 1000 элементу integers1[15]" << endl;

integers1[15] = 1000; // ОШИБКА: вне диапазона

return 0;
}

```

Рис. 8.4. Драйвер для класса Array (часть 6 из 7)

```

Количество созданных массивов = 0
Количество созданных массивов = 2

Размер массива integers1 равен 7
Массив после задания начальных значений:
0 0 0 0 0 0 0

Размер массива integers2 равен 10
Массив после задания начальных значений:
0 0 0 0 0 0 0 0 0 0

Введите 17 целых чисел:
После ввода массивы содержат:
integers1: 1 2 3 4 5 6 7
integers2: 8 9 10 11 12 13 14 15 16 17

```

```

Оценка: integers1 != integers2
Они не равны

Размер массива integers3 равен 7
Массив после задания начальных значений:
1 2 3 4 5 6 7

Присваивание массива integers2 массиву integers1:
integers1: 8 9 10 11 12 13 14 15 16 17
integers2: 8 9 10 11 12 13 14 15 16 17

Оценка: integers1 == integers2
Они равны

integers1[5] равен 13
Присваивание 1000 элементу integers1[5]
integers1: 8 9 10 11 12 1000 14 15 16 17

Попытка присвоить 1000 элементу integers1[15]
Assertion failed: 0 <= subscript && subscript < size,
    file ARRAY1.CPP, line 93
Abnormal program termination

```

Рис. 8.4. Выходные данные драйвера для класса Array (часть 7 из 7)

Пользователю предлагается ввести 17 целых чисел. Перегруженная операция взять из потока используется для чтения этих значений в оба массива с помощью оператора

```
cin >> integers1 >> integers2;
```

Первые семь значений запоминаются в **integers1**, а оставшиеся значения запоминаются в **integers2**. Содержимое обоих массивов выводится на экран операцией вставить в поток, чтобы удостовериться, что ввод был выполнен правильно.

Далее программа проверяет перегруженную операцию проверки на неравенство путем оценки условия

```
integers1 != integers2
```

и сообщает, что массивы действительно не равны.

Программа создает третий массив, названный **integers3**, и инициализирует его с помощью массива **integers1**. Программа выводит размер массива **integers3**, а затем и сам массив, используя перегруженную операцию поместить в поток, чтобы удостовериться, что конструктор задал элементам массива правильные начальные значения.

Далее программа проверяет перегруженную операцию присваивания с помощью выражения **integers1 = integers2**. Печатается содержимое обоих массивов, чтобы удостовериться, что присваивание было правильным. Интересно отметить, что **integers1** первоначально содержал 7 целых чисел и потребовалось изменить его размер, чтобы вместить копию 10 элементов массива **integers2**. Как мы увидим, перегруженная операция присваивания легко выполняет это изменение размера.

Далее программа проверяет перегруженную операцию проверки на равенство (**==**), чтобы можно было убедиться, что объекты **integers1** и **integers2** действительно идентичны после выполнения операции присваивания.

Затем программа использует перегруженную операцию индексации, чтобы найти по ссылке элемент `integers[5]`, принадлежащий `integers1`. Это индексированное имя затем используется как R-величина для печати значения `integers1[5]` и как L-величина с левой стороны оператора присваивания, чтобы присвоить новое значение 1000 элементу `integers1[5]`. Заметим, что `operator[]` возвращает ссылку и присваивает значение.

Затем программа пытается присвоить значение 1000 элементу `integers1[15]`, индекс которого находится вне допустимого диапазона. Программа отлавливает эту ошибку и аварийно завершается.

Интересно, что операция индексации массива не ограничивается применением только к массивам; ее можно использовать для выделения элементов из других видов классов-контейнеров, таких, как связные списки, строки, словари и так далее. Кроме того, индексы — это не обязательно целые числа; можно использовать, например, символы и строки.

Теперь, после того, как мы разобрались с работой этой программы, перейдем к рассмотрению заголовка класса и определений функций-элементов. Строки

```
int *ptr;           // указатель на первый элемент массива
int size;          // размер массива
static int arrayCount; // число экземпляров массивов
```

представляют закрытые элементы класса. Массив состоит из указателя `ptr` на соответствующий тип (в данном случае `int`), элемента `size`, обозначающего количество элементов в массиве, и статического элемента `arrayCount`, обозначающего количество экземпляров объектов массива, которые были образованы.

Строки

```
friend ostream &operator<< (ostream &, const Array &);
friend istream &operator>> (istream &, Array &);
```

объявляют перегруженные операции поместить в поток и взять из потока друзьями класса `Array`. Когда компилятор видит выражение, подобное

```
cout << arrayObject
```

он активизирует функцию `operator<<` путем генерации вызова

```
operator<<(cout, arrayObject)
```

Когда компилятор видит выражение, подобное

```
cin >> arrayObject
```

он активизирует функцию `operator>>` путем генерации вызова

```
operator>> (cin, arrayObject)
```

Еще раз отметим, что эти функции-операции не могут быть элементами класса `Array`, так как объект `Array` всегда упоминается с правой стороны операций поместить в поток и взять из потока. Функция `operator<<` печатает число элементов, указанное с помощью `size`, из массива, начало которого хранится в `ptr`. Функция `operator>>` непосредственно помещает вводимые значения в массив, указанный в `ptr`. Каждая из этих функций-операций возвращает соответствующую ссылку, чтобы обеспечить возможность сцеплений вызовов.

Строка

```
Array(int = 10);           //конструктор с умолчанием
```

объявляет конструктор класса с умолчанием и указывает, что размер массива по умолчанию равен 10 элементам. Когда компилятор видит объявление, подобное

```
Array integers1(7);
```

или эквивалентную форму

```
Array integers1 = 7;
```

он активизирует конструктор с умолчанием. Функция-элемент конструктор с умолчанием класса `Array` увеличивает на 1 `arrayCount`, копирует аргумент в элемент данных `size`, с помощью `new` выделяет память для хранения внутреннего представления этого массива и указатель, возвращенный операцией `new`, присваивает элементу данных `ptr`, применяет `assert` для проверки нормального завершения `new`, а затем использует цикл `for` для задания нулевых начальных значений элементам массива. Можно, конечно, сделать класс `Array`, который не инициализирует свои элементы, если, например, эти элементы все равно должны будут позднее вводиться извне. Но это рассматривается, как плохой стиль программирования. Массив должен быть всегда должным образом инициализирован согласованными данными.

Строка

```
Array(const Array &);           //конструктор копии
```

является конструктором копии. Она задает начальные значения объекту класса `Array` путем копирования существующего объекта класса `Array`. Такое копирование должно выполняться весьма тщательно, чтобы избежать ловушки, состоящей в том, что оба объекта типа `Array` указывают на одну и ту же динамически распределенную область памяти; точно такая же проблема возникла бы с побитовым копированием по умолчанию. Конструкторы копии вызываются всякий раз, когда возникает необходимость в копировании объекта, например, при вызове по значению, когда объект возвращается из вызванной функции, или при инициализации объекта, который должен быть копией другого объекта того же самого класса. Конструктор копии вызывается в объявлении, когда объект класса `Array` создается и инициализируется другим объектом класса `Array`, как, например, в следующем объявлении:

```
Array integers3(integers1);
```

или в эквивалентном объявлении

```
Array integers3 = integers1;
```

Заметим, что конструктор копии *должен* использовать вызов по ссылке, а не вызов по значению. В противном случае вызов конструктора копии приведет к бесконечной рекурсии, потому что при вызове по значению должна быть создана копия объекта, передаваемого конструктору копии, что приводит снова к вызову конструктора копии!

Функция-элемент конструктор копии класса `Array` увеличивает на 1 `arrayCount`, копирует элемент `size` массива, использованного для инициализации, в элемент данных `size` нового объекта, с помощью `new` выделяет память для размещения внутреннего представления массива и присваивает указатель, возвращенный операцией `new`, элементу данных `ptr`, применяет `assert` для проверки успешного завершения `new`, затем использует цикл `for`, чтобы скопировать все элементы инициализирующего массива в новый массив. Важно отметить, что если бы конструктор копии просто скопировал `ptr` из исходного

объекта в новый объект, то оба объекта указывали бы на одну и ту же динамически распределенную область памяти. Выполнение первого же деструктора в дальнейшем уничтожило бы динамически выделенную область памяти и указатели `ptr` остальных объектов оказались бы неопределенными, что привело бы к ситуации, способной вызвать серьезную ошибку выполнения.

Хороший стиль программирования 8.7

Конструктор, деструктор, перегруженную операцию присваивания и конструктор копии обычно применяют совместно для класса, который использует динамически распределенную память.

Строка

```
~Array();           //деструктор
```

объявляет деструктор класса. Деструктор активизируется автоматически по окончании жизни объекта класса `Array`. Деструктор уменьшает на 1 `arrayCount`, затем использует `delete` для освобождения динамической области памяти, созданной в конструкторе с помощью `new`.

Строка

```
const Array &operator=(const Array &);    //присваивание массивов
```

объявляет для класса перегруженную функцию-операцию присваивания. Когда компилятор встречает выражение, подобное

```
integers1 = integers2;
```

он активизирует функцию `operator=`, генерируя вызов

```
integers1.operator=(integers2)
```

Функция-элемент `operator=` сначала осуществляет проверку самоприсваивания. Если имеет место попытка самоприсваивания, присваивание пропускается (т.е. объект уже есть сам по себе; вскоре мы увидим, почему само-присваивание опасно). Если самоприсваивания нет, то функция-элемент использует `delete`, чтобы освободить память, ранее выделенную в массиве-адресате, копирует `size` исходного массива в `size` массива-адресата, использует `new`, чтобы выделить требуемую память массиву-адресату, и помещает указатель, возвращенный `new`, в элемент `ptr` массива, используя при этом `assert` для проверки успешного завершения `new`. Затем используется цикл `for` для копирования элементов исходного массива в массив-адресат. Независимо от того, есть самоприсваивание или нет, функция элемент затем возвращает текущий объект (т.е. `*this`) как константную ссылку; это делает возможным склеенное присваивание, такое как `x=y=z`. Если бы проверки самоприсваивания не было, функция-элемент должна была бы начать с уничтожения пространства массива-адресата. Поскольку при самоприсваивании это также и исходный массив, то массив оказался бы разрушенным.

Типичная ошибка программирования 8.5

Отсутствие проверки самоприсваивания при перегрузке оператора присваивания для класса, имеющего указатель на динамическую область памяти.

Типичная ошибка программирования 8.6

Не предусматривают перегруженную операцию присваивания и конструктор копии для класса, когда объекты этого класса содержат указатели на динамически распределенную память.

Замечание по технике программирования 8.4

Можно запретить присваивание одного объекта класса другому. Это делается путем определения операции присваивания как закрытого элемента класса.

Замечание по технике программирования 8.5

Можно предохранить объекты класса от копирования; для этого просто делают закрытыми и перегруженную операцию присваивания, и конструктор копии.

Строка

```
int operator==(const Array &) const; // проверка равенства
```

объявляет перегруженную операцию проверки на равенство (==) для класса **Array**. Когда компилятор встречает в **main** выражение вида

```
integers1 == integers2
```

он активизирует функцию-элемент **operator==**, генерируя вызов

```
integers1.operator==(integers2)
```

Если элементы **size** массивов различны, функция-элемент **operator==** сразу возвращает 0 (ложь). В противном случае функция-элемент сравнивает каждую пару элементов. Если все они одинаковы, возвращается 1 (истина). Если обнаружилась первая пара различных элементов, это сразу же вызывает возврат 0 (ложь).

Строка

```
int operator!=(const Array &) const; // сравнение на неравенство
```

объявляет для класса перегруженную операцию проверки неравенства массивов. Функция-элемент **operator!=** активизируется и работает аналогично перегруженной функции операции проверки равенства. Отметим, что перегруженная функция **operator!=** могла бы быть написана в терминах перегруженной функции **operator==** следующим образом:

```
int Array::operator!=(const Array &right) const
{ return !(*this == right); }
```

Это определение функции использует перегруженную функцию **operator==**, чтобы определить, равен ли один массив **Array** другому, а затем возвращает отрицание этого результата. Создание функции **operator!=** этим способом дает программисту возможность повторно использовать функцию **operator==** и уменьшить объем описания класса.

Строка

```
int &operator[](int); // операция индексации
```

объявляет для класса перегруженную операцию индексации. Когда компилятор встречает в **main** выражение

```
integers1[] (5)
```

он активизирует перегруженную функцию-элемент `operator[]`, генерируя вызов

```
integers1.operator[](5)
```

Функция-элемент `operator[]` проверяет, находится ли индекс в допустимом диапазоне, и если нет, то программа аварийно завершается. Если индекс в заданном диапазоне, то соответствующий элемент массива возвращается как ссылка, так что он может быть использован как L-величина (например, с левой стороны оператора присваивания).

Строка

```
static int getArrayCount(); //возвращение числа экземпляров
```

объявляет статическую функцию-элемент `getArrayCount`, которая возвращает значение статического элемента данных `arrayCount`, даже если не существует ни одного объекта класса `Array`.

8.9. Преобразования типов

Большинство программ работает с информацией разных типов. Иногда все операции остаются «в пределах одного типа». Например, сложение целого и целого дает снова целое (до тех пор, пока результат не станет слишком большим для представления целым). Но часто возникает необходимость преобразовать данные одного типа в данные другого типа. Это может случиться при присваиваниях, при вычислениях, при передаче значений функциям и возвращении значений от функций. Компилятор знает, как выполнить определенные преобразования между встроенными типами. Программист может регулировать преобразования встроенных типов путем приведения типов.

Но что делать с типами, определенными пользователем? Компилятор не может сам по себе знать, как выполнять преобразования между встроенными типами и типами, определенными пользователем. Программист должен явно указать, как выполнять такие преобразования. Эти преобразования могут быть выполнены с помощью *конструкторов преобразований* — конструкто-ров с единственным аргументом, которые преобразуют объекты разных типов (включая встроенные типы) в объекты данного класса. Позже в этой главе мы используем конструктор преобразований, чтобы преобразовать обычные строки `char*` в объекты класса `String`.

Операция преобразования (называемая также *операцией приведения*) может быть использована для преобразования объекта одного класса в объект другого класса или в объект встроенного типа. Такая операция преобразования должна быть нестатической функцией-элементом; операция преобразования этого вида не может быть дружественной функцией.

Прототип функции

```
A::operator char*() const;
```

объявляет перегруженную функцию-операцию приведения для создания временного объекта `char*` из объекта определенного пользователем типа A. Перегруженная функция-операция приведения не указывает тип возвращаемых величин, потому что тип возвращаемых величин — это тип, к которому преобразован объект. Если s — объект класса, то когда компилятор встречает выражение (`char*`)s, он порождает вызов `s.operator char*()`. Операнд s — это объект класса s, для которого была активизирована функция-элемент `operator char*`.

Перегруженная функция-операция приведения может быть определена для преобразования объектов определенных пользователем типов в объекты встроенных типов или в объекты других определенных пользователем типов. Прототипы

```
A::operator int() const;
A::operator otherClass() const;
```

объявляют перегруженные функции-операции приведения для преобразования объекта определенного пользователем типа A в целое и для преобразования объекта определенного пользователем типа A в объект другого определенного пользователем типа otherClass.

Одной из приятных особенностей операций приведения и конструкторов преобразований является то, что при необходимости компилятор может вызывать эти функции автоматически для создания временных объектов. Например, если в программе в том месте, где нормально ожидается `char*`, появился объект `s` определенного пользователем типа `String`, например, в операторе

```
cout << s;
```

то в этом случае компилятор вызывает перегруженную функцию-операцию приведения `operator char*` для преобразования объекта в `char*` и использует в выражении результирующий `char*`. Эта операция приведения для нашего класса `String` позволяет не перегружать операцию поместить в поток, предназначенную для вывода `String` с использованием `cout`.

8.10. Учебный пример: класс строка

Краеугольным камнем изучения перегрузок операций явится построение нами класса, который управляет созданием и обработкой строк. C++ не имеет встроенного строкового типа данных. C++ предоставляет нам возможность добавить в качестве класса наш собственный строковый тип и посредством механизма перегрузки обеспечить набор удобных операций для работы со строками. Различные части программы на рис. 8.5 включают заголовок класса, определения функций-элементов и программу драйвер для проверки нашего нового класса `String`.

Сначала мы рассмотрим заголовок класса `String` и обсудим закрытые данные, использованные для представления объектов `String`. Затем мы пройдемся по открытому интерфейсу класса, обсуждая каждую из услуг, которые предоставляет наш класс.

Далее мы рассмотрим программу драйвер в `main`. Мы обсудим желательный нам стиль кодирования, т.е. вид выражений, которые нам хотелось бы иметь возможность записывать для объектов нашего нового класса `String`, и соответствующий набор перегруженных операций класса.

Затем мы обсудим определения функций-элементов класса `String`. Для каждой перегруженной операции мы рассмотрим код в программе драйвере, который активизирует перегруженную функцию-операцию, и объясним, как перегруженная функция-операция работает.

```

// 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 &); // склеивание (конкатенация)
    int operator!() const;             // String пуст?
    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);             // возвращение ссылки на символ
    String operator()(int, int);       // возвращение подстроки
    int getLength() const;             // возвращение длины строки

private:
    char *sPtr;                      // указатель на начало строки
    int length;                       // длина строки
};

#endif

```

Рис. 8.5. Определение базового класса **String** (часть 1 из 8)

```

// STRING2.CPP
// Определения функций-элементов класса String
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <assert.h>
#include "string2.h"

// Конструктор преобразования: преобразовывает char * в String
String::String(const char *s)
{
    cout << "Конструктор преобразования: " << s << endl;
    length = strlen(s);           // вычисление длины
    sPtr = new char[length + 1];   // выделение памяти
    assert(sPtr != 0);            // завершение, если память не выделена
    strcpy(sPtr, s);              // копирование строки аргумента в объект
}

```

Рис. 8.5. Определения функций-элементов класса **String** (часть 2 из 8)

```

// Конструктор копии
String::String(const String &copy)
{
    cout << "Конструктор копии: " << copy.sPtr << endl;
    length = copy.length;           // копирование длины
    sPtr = new char[length + 1];    // выделение памяти
    assert(sPtr != 0);             // завершение, если память не выделена
    strcpy(sPtr, copy.sPtr);       // копирование строки
}

// Деструктор
String::~String()
{
    cout << "Деструктор: " << sPtr << endl;
    delete [ ] sPtr;              // освобождение памяти, отведенной строке
}

// Перегруженная операция = ; избегает самоприсваивания
const String &String::operator=(const String &right)
{
    cout << "вызов operator=" << endl;

    if (&right != this) {          // проверка самоприсваивания
        delete [ ] sPtr;          // предотвращение утечки памяти
        length = right.length;    // новая длина String
        sPtr = new char[length + 1]; // выделение памяти
        assert(sPtr != 0);         // подтверждение выделения памяти
        strcpy(sPtr, right.sPtr);  // копирование строки
    }
    else
        cout << "Попытка самоприсваивания String" << endl;

    return *this;                 // обеспечивает возможность сцепленных
                                  // присваиваний
}

// Сцепление (конкатенация) правого операнда с данным объектом и
// сохранение сцепленной строки в этом объекте.
String &String::operator+=(const String &right)
{
    char *tempPtr = sPtr;          // сохранение до возможности удаления
    length += right.length;       // новая длина String
    sPtr = new char[length + 1];   // выделение памяти
    assert(sPtr != 0);             // завершение, если память не выделена
    strcpy(sPtr, tempPtr);        // левая часть новой String
    strcat(sPtr, right.sPtr);     // правая часть новой String
    delete [ ] tempPtr;           // освобождение прежней области памяти

    return *this;                 // обеспечивает возможность
                                  // сцепленных вызовов
}

// Пуст ли String?
int String::operator!() const { return length == 0; }

```

Рис. 8.5. Определения функций-элементов класса **String** (часть 3 из 8)

```
// Равен ли этот String правому String?  
int String::operator==(const String &right) const  
{ return strcmp(sPtr, right.sPtr) == 0; }  
  
// Этот String неравен правому String?  
int String::operator!=(const String &right) const  
{ return strcmp(sPtr, right.sPtr) != 0; }  
  
// Меньше ли этот String правого String?  
int String::operator<(const String &right) const  
{ return strcmp(sPtr, right.sPtr) < 0; }  
  
// Больше ли этот String правого String?  
int String::operator>(const String &right) const  
{ return strcmp(sPtr, right.sPtr) > 0; }  
  
// Больше или равен этот String по сравнению с правым String?  
int String::operator>=(const String &right) const  
{ return strcmp(sPtr, right.sPtr) >= 0; }  
  
// Меньше или равен этот String по сравнению с правым String?  
int String::operator<=(const String &right) const  
{ return strcmp(sPtr, right.sPtr) <= 0; }  
  
// Возвращение ссылки на символ в String.  
char &String::operator[](int subscript)  
{  
    // первая проверка, не находится ли подстрока вне диапазона  
    assert(subscript >= 0 && subscript < length);  
  
    return sPtr[subscript];    // создание L-величины  
}  
  
// Возвращение подстроки, начинающейся с заданного индекса  
// и имеющей длину subLength, как ссылки на объект String  
String String::operator()(int index, int subLength)  
{  
    // подтверждение того, что индекс в диапазоне  
    // и длина подстроки >= 0  
    assert(index >= 0 && index < length && subLength >= 0);  
  
. String sub;                      // пустой String  
  
    // определение длины подстроки  
    if ((subLength == 0) || (index + subLength > length))  
        sub.length = length - index + 1;  
    else  
        sub.length = subLength + 1;  
  
    // выделение памяти для подстроки  
    delete sub.sPtr;                  // удаление символа из объекта  
    sub.sPtr = new char[sub.length];  
    assert(sub.sPtr != 0);            //подтверждение выделения памяти
```

Рис. 8.5. Определения функций-элементов класса **String** (часть 4 из 8)

```

// копирование подстроки в новый String
strncpy(sub.sPtr, &sPtr[index], sub.length);
sub.sPtr[sub.length] = '\0';           // завершение нового String

return sub;                         // возвращение копии sub объекта String
}

// Возвращение длины строки
int String::getLength() const { return length; }

// Перегруженная операция вывода данных
ostream &operator<< (ostream &output, const String &s)
{
    output << s.sPtr;
    return output;                  // возможность склеивания
}

// Перегруженная операция ввода данных
istream &operator>>(istream &input, String &s)
{
    char temp[100];                // буфер для хранения входных данных

    input >> setw(100) >>temp;
    s = temp;                      // использование операции присваивания
                                    // класса String
    return input;                  // возможность склеивания
}

```

Рис. 8.5. Определения функций-элементов класса **String** (часть 5 из 8)

```

// FIG8_5.CPP
// Драйвер для класса String
#include <iostream.h>
#include "string2.h"

main()
{
    String s1("поздравляем" ), s2( " с днем рождения" ), s3;

    // проверка перегруженных операций проверки равенства и отношений
    cout << "s1 равна \"" << s1 << "\"; s2 равна \"" << s2
        << "\"; s3 равна \"" << s3 << '\"' << endl
        << "Результат сравнения s2 и s1:" << endl
        << "s2 == s1 дает результат " << (s2 == s1) << endl
        << "s2 != s1 дает результат " << (s2 != s1) << endl
        << "s2 > s1 дает результат " << (s2 > s1) << endl
        << "s2 < s1 дает результат " << (s2 < s1) << endl
        << "s2 >= s1 дает результат " << (s2 >= s1) << endl
        << "s2 <= s1 дает результат " << (s2 <= s1) << endl << endl;

    // проверка перегруженной операции проверки пустого String (!)
    cout << "Проверка !s3:" << endl;
    if (!s3)
        cout << "s3 пустая; присваивание s1 строке s3;" << endl;
        s3 =s1;          // проверка перегруженного присваивания
        cout << "s3 равна \"" << s3 << "\"" << endl << endl;
}

```

Рис. 8.5. Драйвер для проверки класса **String** (часть 6 из 8)

```
// проверка перегруженной операции сцепления String
cout << "s1 += s2 дает результат s1 = ";
s1 += s2; // проверка перегруженного сцепления
cout << s1 << endl << endl;

// проверка конструктора преобразования
cout << "s1 += \" вас\" дает результат" << endl;
s1 += " вас "; // проверка конструктора преобразования
cout << "s1 = " << s1 << endl << endl;

// проверка перегруженной функции вызова операции ()
// для подстроки
cout << "Подстрока s1 начиная с " << endl
    << "0 до 27, s1(0, 27), равна: "
    << s1(0, 27) << endl << endl;

// проверка опции подстроки "до конца String"
cout << "Подстрока s1 начиная с" << endl
    << "28, s1(28, 0), равна: "
    << s1(28, 0) << endl << endl; // 0 означает "до конца
                                         // строки"

// проверка конструктора копии
String *s4Ptr = new String(s1);
cout << "*s4Ptr = " << *s4Ptr << endl << endl;

// проверка операции присваивания (=) с самоприсваиванием
cout << "самоприсваивание *s4Ptr" << endl;
*s4Ptr = *s4Ptr; // проверка перегруженного
                  // самоприсваивания
cout << "*s4Ptr = " << *s4Ptr << endl;

// проверка деструктора
delete s4Ptr;

// проверка использования операции подстроки для создания
// L-величины
s1[0] = 'П';
s1 [14] = 'Д';
cout << endl
    << "s1 после s1[0] = 'П' и s1[14] = 'Д' дает результат: "
    << s1 << endl << endl;

// проверка индекса на выход из диапазона
cout << "Попытка присвоить 'Д' элементу s1[50] дает результат: "
    << endl;
s1 [50] = 'Д'; // ОШИБКА: индекс вне диапазона

return 0;
}
```

Рис. 8.5. Драйвер для проверки класса **String** (часть 7 из 8)

```

Конструктор преобразования: поздравляем
Конструктор преобразования: с днем рождения
Конструктор преобразования:
s1 равна "поздравляем"; s2 равна " с днем рождения"; s3 равна ""
Результат сравнения s2 и s1:
s2 == s1 дает результат 0
s2 != s1 дает результат 1
s2 > s1 дает результат 0
s2 < s1 дает результат 1
s2 >= s1 дает результат 0
s2 <= s1 дает результат 1

Проверка !s3:
s3 пустая; присваивание s1 строке s3;
вызов operator=
s3 равна "поздравляем"

s1 += s2 дает результат s1 = поздравляем с днем рождения

s1 += " вас" дает результат
Конструктор преобразования: вас
Деструктор: вас
s1 = поздравляем с днем рождения вас

Конструктор преобразования:
Подстрока s1 начиная с
0 до 27, s1(0, 27), равна: поздравляем с днем рождения

Конструктор преобразования:
Подстрока s1 начиная с
28, s1(28, 0), равна: вас

Конструктор копии: поздравляем с днем рождения вас
*s4Ptr = поздравляем с днем рождения вас

самоприсваивание *s4Ptr
вызов operator=
Попытка самоприсваивания String
*s4Ptr = поздравляем с днем рождения вас
Деструктор: поздравляем с днем рождения вас

s1 после s1[0] = 'П' и s1[14] = 'д' дает результат: Поздравляем с
днем рождения вас

Попытка присвоить 'д' элементу s1[50] дает результат:
Assertion failed: subscript >= 0 && subscript < iength,
file STRING2.CPP, line 99
Abnormal program termination

```

Рис. 8.5. Выходные данные драйвера для проверки класса String (часть 8 из 8)

Мы начнем с внутреннего представления String. Строки

```

private:
char *sPtr; // указатель на начало строки
int length; // длина строки

```

объявляют закрытые элементы класса. Объект **String** имеет указатель на динамически выделенную память для хранения строки символов, и имеет поле **длины**, которое хранит количество символов в строке без учета нулевого символа в конце.

Теперь пройдемся по заголовочному файлу класса **String** на рис. 8.5. Строки

```
friend ostream &operator<<(ostream &, const String &);  
friend istream &operator>>(istream &, const String &);
```

объявляют перегруженные функции-операции поместить в поток **operator<<** и взять из потока **operator>>** друзьями класса. Реализация этого очевидна.

Строка

```
String(const char * = ""); // конструктор преобразования
```

объявляет конструктор преобразования. Этот конструктор берет аргумент **char*** (это по умолчанию пустая строка) и создает объект **String**, который включает эту строку символов. Любой конструктор с единственным аргументом можно рассматривать как конструктор преобразования. Как мы увидим, такие конструкторы полезны, когда мы выполняем любую операцию со **String**, используя аргументы **char***. Конструктор преобразования преобразует соответствующую строку в объект **String**, который затем присваивается объекту-адресату **String**. Наличие этого конструктора преобразования означает, что нет необходимости применять перегруженную операцию специально для присваивания строк символов объектам **String**. Компилятор автоматически активизирует конструктор преобразования для создания временного объекта **String**, содержащего строку символов. Затем активизируется перегруженная операция присваивания, чтобы присвоить временный объект **String** другому объекту **String**.

Замечание по технике программирования 8.6

При применении конструктора преобразования для выполнения неявного преобразования типов C++ может использовать неявный вызов только одного конструктора, чтобы попытаться удовлетворить требование другой перегруженной операции. Не возможно удовлетворять требования перегруженных операций путем выполнения последовательности неявных определенных пользователем преобразований.

Конструктор преобразования **String** может быть активизирован, например, таким объявлением: **String s1("поздравляем")**. Функция-элемент конструктора преобразования вычисляет длину строки символов и присваивает эту длину закрытому элементу данных **length**, использует **new** для присваивания значения указателя на выделенный необходимый объем памяти закрытому элементу данных **sPtr**, применяет **assert** для проверки успешного выполнения **new** и, если все нормально, использует **strcpy** для копирования строки символов в объект.

Строка

```
String(const String &); // конструктор копии
```

определяет конструктор копии. Он инициализирует новый объект **String**, являющийся копией существующего объекта **String**. Такое копирование должно быть выполнено весьма аккуратно, чтобы избежать ловушки, связанной с тем, что оба объекта **String** указывают на одну и ту же динамически распре-

деленную область памяти; такая проблема возникает, в частности, при побитовом копировании, осуществляемом по умолчанию. Конструктор копии работает аналогично конструктору преобразования, за исключением того, что он просто копирует элемент `length` из исходного объекта `String` в объект-адресат `String`. Заметим, что конструктор копии выделяет новую область памяти для внутреннего представления строки символов объекта-адресата. Если бы он просто копировал `sPtr` в исходном объекте в `sPtr` объекта-адресата, то оба объекта указывали бы на одну и ту же динамически распределенную область памяти. Выполнение первого же деструктора в дальнейшем уничтожило бы динамически выделенную область памяти и указатели `ptr` остальных объектов оказались бы неопределенными, что привело бы к ситуации, способной вызвать серьезную ошибку выполнения.

Строка

```
~String();           // деструктор
```

объявляет деструктор класса `String`. Деструктор использует `delete` для освобождения области динамически распределенной памяти, выделенной `new` под хранение строки символов.

Строка

```
const String &operator=(const String &);    // присваивание
```

объявляет перегруженную операцию присваивания. Когда компилятор встречает выражение вида `string1 = string2`, он генерирует вызов функции

```
string1.operator=(string2);
```

Перегруженная функция-операция присваивания `operator=` сначала проверяет, не осуществляется ли самоприсваивание. Если должно осуществляться самоприсваивание, функция просто возвращается, потому что объект уже существует сам по себе. Если бы эта проверка пропускалась, функция сразу же освобождала бы область памяти в объекте-адресате и поэтому теряла бы строку символов. Если производится не самоприсваивание, функция освобождает область памяти, копирует поле длины исходного объекта в объект-адресат, создает новую область памяти в объекте-адресате, использует `assert` для проверки успешного выполнения `new` и затем использует `strcpy` для копирования строки символов из исходного объекта в объект-адресат. Независимо от того, было или не было самоприсваивание, возвращается `*this` для того, чтобы обеспечивать сплленные присваивания.

Строка

```
String &operator+=(const String &);        // склеение (конкатенация)
```

объявляет перегруженную операцию склейки строк (конкатенации). Когда компилятор встречает в `main` выражение `s1 += s2`, генерируется вызов функции `s1.operator+=(s2)`. Функция `operator+=` создает временный указатель, чтобы сохранять строку символов текущего объекта до тех пор, пока не появится возможность освободить занимаемую ею память, вычисляет общую длину склеенной строки, использует `new`, чтобы зарезервировать место для суммарной строки, использует `assert` для проверки успешного выполнения `new`, использует `strcpy` для копирования исходной строки в заново выделенную область, использует `strcat` для склейки строки символов исходного объекта с новой строкой, использует `delete`, чтобы освободить область памяти, занятую исходной строкой символов этого объекта, и возвращает `*this` как `String &`, чтобы обеспечить скленные операции `+=`.

Нужна ли нам вторая перегруженная операция сцепления объекта типа **String** не с другим объектом того же типа, а со строкой типа **char***? Нет. Конструктор преобразования **const char*** преобразует строку во временный объект класса **String**, с которым затем и выполняется описанная перегруженная операция сцепления. С++ может выполнять такие преобразования, но только на глубину одного уровня. С++ может также выполнять неявное определенное компилятором преобразование между встроенными типами перед выполнением преобразования между встроенными типами и классами. Отметим, что при создании временного объекта класса **String** вызываются конструктор преобразования и деструктор (смотри на рис.8.5 результатирующие выходные данные, начиная с **s1+="вас"** дает результат). Это пример скрытых от клиента класса накладных расходов вызовов функций при создании временных объектов класса и уничтожении их во время неявных преобразований. Аналогичные накладные расходы создаются конструкторами копий при передаче им параметров вызовом по значению и при возвращении объектов класса по значению.

Совет по повышению эффективности 8.2

Наличие перегруженной операции сцепления **+=**, которая получала бы один аргумент типа **const char***, более эффективно, чем выполнение сначала неявного преобразования, а затем сцепления. С другой стороны, неявные преобразования требуют меньшего объема кодирования и вызывают меньше ошибок.

Строка

```
int operator!() const;           // String пуст?
```

объявляет перегруженную операцию отрицания. Эта операция с классами строк обычно используется для проверки, пуста ли строка. Например, когда компилятор встречает выражение **!string1**, он генерирует вызов функции

```
string1.operator!()
```

Эта функция просто возвращает результат проверки, не равняется ли **length** нулю.

Строки

```
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 проверки на равенство и отношения для класса **String**. Аналогичность всех этих операций позволяет нам ограничиться рассмотрением только одного примера, а именно, перегрузкой операции **>=**. Когда компилятор встречает выражение вида **string1 >= string2**, он генерирует вызов функции

```
string1.operator>=(string2)
```

которая возвращает 1 (истина), если **string1** больше или равна **string2**. Каждая из рассматриваемых операций использует **strcmp** для сравнения строк символов в объектах типа **String**. Подчеркнем, что каждая из операций отношения

и проверки на равенство реализуется с использованием функции `strcmp` из стандартной библиотеки С. Многие программисты на C++ отстаивают использование одних перегруженных функций-операций для реализации других функций. Например, перегруженная функция операция `operator>=` могла бы быть реализована следующим образом:

```
int String::operator>=(const String &right) const
{ return (*this > right || *this == right) ? 1 : 0; }
```

Такое определение функции-элемента `operator>=` использует перегруженные операции `>` и `==`, чтобы определить, больше ли один объект типа `String` другого объекта того же типа или не равны ли они. Реализация функций-элементов, использующих определенные ранее функции-элементы, позволяет программисту уменьшить размер кода, который должен быть написан в начале класса, и увеличить объем повторно используемого кода в классе.

Строка

```
char &operator[](int);           // возвращение ссылки на символ
```

объявляет перегруженную операцию индексации. Когда компилятор встречает выражение вида `string1[0]`, он генерирует вызов функции `string1.operator[](0)`. Функция `operator[]` сначала использует `assert`, чтобы выполнить проверку, находится ли индекс внутри допустимого диапазона; если индекс находится вне диапазона, программа печатает сообщение об ошибке и аварийно завершается. Если индекс внутри диапазона, `operator[]` возвращает соответствующий символ строки объекта класса `String` как `char &`; эту ссылку `char &` можно использовать как L-величину для модификации указанного символа строки объекта класса `String`.

Строка

```
String operator()(int, int);      // возвращение подстроки
```

объявляет *перегруженную операцию вызова функции*. В классах строк принято перегружать эту операцию для выделения подстроки из объекта класса `String`. Два целых параметра указывают индекс начала и длину подстроки, выделяемой в строке объекта класса `String`. Если индекс начала находится вне допустимого диапазона или длина подстроки отрицательная, выдается сообщение об ошибке. По соглашению, если длина подстроки равна 0, то подстрока выделяется во всех случаях до конца строки в объекте класса `String`. Например, предположим, что `string1` является объектом класса `String`, содержащим строку символов "AEIOU". Когда компилятор встречает выражение `string1(2,2)`, он генерирует вызов функции `string1.operator()(2,2)`. При выполнении этого вызова создается временный объект `String`, содержащий строку "IO", и возвращается копия этого объекта.

Перегрузка операций вызова функций () является мощным средством, потому что функции могут получать списки параметров произвольной длины и сложности. Мы можем использовать эти возможности для многих интересных задач. Одним из таких применений операции вызова функции является альтернативная запись индексированного массива: вместо неудобной принятой в С записи двойных квадратных скобок для двумерного массива, имеющей вид `a[b][c]`, некоторые программисты предпочитают перегрузить операцию вызова функции, чтобы иметь возможность записывать `a(b, c)`. Перегруженная операция вызова функции может быть только нестатической функцией-элементом. Эта операция используется только когда «имя функции» является объектом класса `String`.

Строка

```
int getLength() const;           // возвращение длины строки
```

объявляет функцию, которая возвращает длину объекта `String`. Заметим, что эта функция получает длину путем возвращения значения закрытых данных класса `String`.

Теперь читатель может сам просмотреть программу `main`, изучить выходные данные программы и исследовать результаты каждого использования перегруженной операции.

8.11. Перегрузка ++ и --

Все операции инкремента и декремента в префиксной и постфиксной формах могут быть перегружены. Мы увидим, как компилятор различает префиксные или постфиксные варианты операций инкремента или декремента.

Чтобы перегрузить операцию инкремента для получения возможности использования и префиксной, и постфиксной форм, каждая из этих двух перегруженных функций-операций должна иметь разную сигнатуру, чтобы компилятор имел возможность определить, какая версия `++` имеется в виду в каждом конкретном случае. Префиксный вариант перегружается точно так же, как любая другая префиксная унарная операция.

Предположим, например, что мы хотим прибавить 1 к дню в объекте `d1` класса `Date`. Когда компилятор встречает выражение с префиксным инкрементом

```
++d1
```

компилятор генерирует вызов функции-элемента

```
d1.operator++()
```

прототип которой должен иметь вид

```
Date operator++();
```

Если префиксная форма инкремента реализуется как функция, не являющаяся элементом, то, когда компилятор встречает выражение

```
++d1
```

он генерирует вызов функции

```
operator++(d1)
```

прототип которой должен быть объявлен в классе `Date` как дружественный:

```
friend Date operator++(Date &);
```

Перегрузка постфиксной формы операции инкремента представляет некоторые трудности, так как компилятор должен быть способен различать сигнатуры перегруженных функций-операций инкремента в префиксной и постфиксной формах. По соглашению, принятому в C++, когда компилятор встречает выражение постфиксной формы инкремента

он генерирует вызов функции

```
d1.operator++(0)
```

прототипом которой является

```
Date operator++(int)
```

Нуль (0) в генерируемом вызове функции является чисто формальным значением, введенным для того, чтобы сделать список аргументов функции **operator++**, используемой для постфиксной формы инкремента, отличным от списка аргументов функции **operator++**, используемых для префиксной формы инкремента.

Если постфиксная форма операции инкремента реализуется как функция, не являющаяся элементом, то, когда компилятор встречает выражение

```
d1++
```

он генерирует вызов функции

```
operator++(d1, 0)
```

прототип которой должен иметь вид

```
friend Date operator++(Date &, int);
```

Опять формальный аргумент 0 используется компилятором только для того, чтобы список аргументов функции **operator++**, которая используется для постфиксной формы инкремента, отличался от списка аргументов функции **operator++**, используемой для префиксной формы инкремента.

Все рассмотренное в этом разделе по отношению к перегрузке операций инкремента в префиксной и постфиксной формах, приложимо и к перегрузке операций декремента.

8.12. Учебный пример: класс **дата**

Рисунок 8.6 иллюстрирует класс **Date**. Класс использует перегруженные операции инкремента в префиксной и постфиксной формах для прибавления 1 к дню в объекте **Date**, что может вызвать в ряде случаев увеличение на 1 и месяца и года.

Класс имеет следующие функции-элементы: перегруженную операцию поместить в поток, конструктор с умолчанием, функцию **setDate**, перегруженные функции-операции инкремента в префиксной и постфиксной формах, перегруженную операцию сложения с присваиваем (**+=**), функцию проверки высокосного года и функцию, определяющую, является ли указанный день последним днем месяца.

Программа драйвера в **main** создает объекты дат: **d1**, которая по умолчанию получает начальное значение Январь 1, 1900, **d2**, которая получает начальное значение Декабрь 27, 1992 и **d3**, которой программа пытается присвоить неправильную дату. Конструктор класса **Date** вызывает **setDate** для установки заданных значений месяца, дня и года. Если месяц неправильный, он устанавливается равным 1. Неправильный год устанавливается равным 1900. Неправильный день устанавливается равным 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
                                                // с умолчанием
    void setDate(int, int, int); // установка даты
    Date operator++(); // префиксная форма инкремента
    Date operator++(int); // постфиксная форма инкремента
    const Date &operator+=(int); //добавление дня, изменение
                                // объекта
    int leapYear(int); // это високосный год?
    int endOfMonth(int); // это конец месяца?
private:
    int month;
    int day;
    int year;
    static int days[ ]; // массив дней в месяце
    void helpIncrement(); // функция-утилиты
};

#endif

```

Рис. 8.6. Определение класса **Date** (часть 1 из 6)

```

// DATE1.CPP
// Определения функций-элементов класса Date
#include <iostream.h>
#include "date1.h"

// Инициализация статического элемента области действия файла;
// одна побитовая копия.
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); }

// Установка даты
void Date::setDate(int mm, int dd, int yy)
{
    month = (mm >= 1 && mm <= 12) ? mm : 1;
    year = (yy >= 1900 && yy <= 2100) ? yy : 1900;

    if (month == 2 && leapYear(year)) // проверка високосного года
        day = (dd >= 1 && dd <= 29) ? dd : 1;
    else
        day = (dd >= 1 && dd <= days[month]) ? dd : 1;
}

```

Рис. 8.6. Определения функций-элементов класса **Date** (часть 2 из 6)

```

// Операция префиксной формы инкремента,
// перегруженная как функция-элемент.
Date Date::operator++()
{
    helpIncrement();
    return *this;           // возвращение значения, а не ссылки
}
// Операция постфиксной формы инкремента,
// перегруженная как функция-элемент.
// Заметьте, что фиктивный целый параметр не имеет имени.
Date Date::operator++(int)
{
    Date temp = *this;
    helpIncrement();

    // возвращение не увеличенного, сохраненного, временного объекта
    return temp;           // возвращение значения, а не ссылки
}

// Добавление указанного количества дней к дате
const Date &Date::operator+=(int additionalDays)
{
    for (int i = 1; i <= additionalDays; i++)
        helpIncrement();

    return *this;           // возможность сцепления
}

// Определение, високосный ли год
int Date::leapYear(int y)
{
    if (y % 400 == 0 || (y % 100 != 0 && y % 4 == 0) )
        return 1;           // високосный год
    else
        return 0;           // не високосный год
}

// Определение, последний ли день месяца
int Date::endOfMonth(int d)
{
    if (month == 2 && leapYear(year))
        return d == 29;     // последний день февраля в високосном году
    else
        return d == days[month];
}

```

Рис. 8.6. Определения функций-элементов класса **Date** (часть 3 из 6)

Программа драйвер выводит каждый из сконструированных объектов типа **Date**, используя перегруженную операцию поместить в поток. Перегруженная операция **+=** используется для прибавления 7 дней к **d2**. Затем используется функция **setDate** для установки в **d3** даты Февраль 28, 1992. Далее, в новом объекте **d4** устанавливается дата Март 18, 1969. Затем **d4** увеличивается на 1 с помощью перегруженной операции префиксного инкремента. Дата печатается до и после инкрементирования, чтобы удостовериться, что она изменилась правильно. В заключение, **d4** инкрементируется перегруженной операцией постфиксного инкремента. Дата печатается до и после инкрементирования, чтобы удостовериться, что она изменилась правильно.

```

// Функция помоши инкрементирования даты
void Date::helpIncrement()
{
    if (endOfMonth(day) && month == 12) {           // конец года
        day = 1;
        month = 1;
        ++year;
    }
    else if (endOfMonth(day)) (                      // конец месяца
        day = 1;
        ++month;
    }
    else                                         // не конец месяца или года; инкремент дня
        ++day;
}

// Перегруженная операция вывода данных
ostream &operator<< (ostream &output, const Date &d)
{
    static char *monthName[13] = {"", "Январь",
        "Февраль", "Март", "Апрель", "Май", "Июнь",
        "Июль", "Август", "Сентябрь", "Октябрь",
        "Ноябрь", "Декабрь"};

    output << monthName[d.month] << ' '
        << d.day << ", " << d.year;

    return output;                                // возможность сцепления
}

```

Рис. 8.6. Определения функций-элементов класса **Date** (часть 4 из 6)

Перегрузка операции префиксного инкремента очевидна. Программа вызывает функцию утилиту **helpIncrement** для того, чтобы выполнить сам инкремент. Эта функция обеспечивает переходы от конца месяца к началу, когда мы увеличиваем последний день месяца. Эти переходы требуют увеличения месяца. Если месяц уже равен 12, то тогда нужно также увеличить год. Функция **helpIncrement** использует функции **leapYear** и **endOfMonth**, чтобы правильно увеличить день.

Перегруженная операция префиксного инкремента возвращает *копию* текущего объекта с измененной датой. Это происходит, потому что текущий объект ***this**, возвращается как объект класса **Date**, что активизирует конструктор копии.

Перегрузка операции постфиксного инкремента немного более сложная. Чтобы эмулировать действие постфиксного инкремента, мы должны вернуть неизмененную копию объекта класса **Date**. При входе в **operator++** мы спасаем текущий объект (***this**) во временном объекте **temp**. Затем мы вызываем **helpIncrement**, чтобы инкрементировать объект. В итоге мы возвращаем неизмененную копию объекта в **temp**. Отметим, что эта функция не может вернуть ссылку на объект класса **Date**, потому что значение, которое надо вернуть, сохраняется в локальной переменной в определении функции. Локальные переменные уничтожаются, когда функция, в которой они объявлены, завершена. Таким образом, объявление типа, возвращаемого функцией, как **Date &**, привело бы к ссылке на объект, который после возвращения

больше не существует. Возвращение ссылки на локальную переменную является типичной ошибкой, которую трудно найти.

```
// FIG8 6.CPP
// Драйвер для класса Date
#include <iostream.h>
#include "date1.h"

main()
{
    Date d1, d2(12,27,1992), d3(0, 99, 8045);
    cout << "d1 равна " << d1 << endl
        << "d2 равна " << d2 << endl
        << "d3 равна " << d3 << endl << endl;

    cout << "d2 += 7 равна " << (d2 += 7) << endl << endl;

    d3.setDate(2, 28, 1992);
    cout << "d3 равна " << d3 << endl;
    cout << "+d3 равна " << ++d3 << endl << endl;

    Date d4(3, 18, 1969);

    cout << "Проверка операции префиксного инкремента:" << endl
        << "d4 равна " << d4 << endl;
    cout << "+d4 равна " << ++d4 << endl;
    cout << "d4 равна " << d4 << endl << endl;

    cout << "Проверка операции постфиксного инкремента:" << endl
        << "d4 равна " << d4 << endl;
    cout << "d4++ равна " << d4++ << endl;
    cout << "d4 равна " << d4 << endl << endl;

    return 0;
}
```

Рис. 8.6. Определения функций-элементов класса **Date** (часть 5 из 6)

```
d1 равна Январь 1, 1900
d2 равна Декабрь 27, 1992
d3 равна Январь 1, 1900

d2 += 7 равна Январь 3, 1993

d3 равна Февраль 28, 1992
++d3 равна Февраль 29, 1992

Проверка операции префиксного инкремента:
d4 равна Март 18, 1969
++d4 равна Март 19, 1969
d4 равна Март 19, 1969

Проверка операции постфиксного инкремента:
d4 равна Март 19, 1969
d4++ равна Март 19, 1969
d4 равна Март 20, 1969
```

Рис. 8.6. Выходные данные драйвера класса **Date** (часть 6 из 6)

Резюме

- Операция << используется в C++ для многих целей: и как операция поместить в поток, и как операция сдвига влево. Это пример перегрузки операции. Подобным же образом перегружается операция >>; она используется и как операция взять из потока, и как операция сдвига вправо.
- C++ предоставляет программисту возможность перегружать большинство операций и делать их чувствительными к контексту, в котором они используются. Компилятор генерирует соответствующий код, основываясь на способе использования операции.
- Перегрузка операций способствует расширяемости C++.
- Операции перегружаются путем составления описания функции (с заголовком и телом). Имя функции состоит из ключевого слова **operator**, после которого записывается перегружаемая операция.
- Чтобы использовать операцию над объектами классов, эта операция должна быть перегружена, но есть два исключения. Операция присваивания (=) может быть использована с каждым классом без явной перегрузки, выполняя по умолчанию *побитовое копирование* данных-элементов класса. Операция адресации (&) также может быть использована с объектами любых классов без перегрузки; она просто возвращает адрес объекта в памяти.
- Перегрузка операций обеспечивает такие же краткие выражения для типов, определенных пользователем, какие C++ обеспечивает с помощью богатого набора операций для встроенных типов.
- Большинство операций C++ можно перегружать.
- Старшинство и ассоциативность операций не может быть изменено перегрузкой.
- При перегрузке операций нельзя использовать аргументы по умолчанию.
- Изменить количество operandов, которое берет операция, невозможно: перегруженные унарные операции остаются унарными, перегруженные бинарные операции остаются бинарными.
- Нельзя создавать обозначения для новых операций; перегружать можно только существующие операции.
- Перегрузка не может изменять смысл работы операций с объектами встроенного типа.
- При перегрузке (), [], -> или = функция перегрузки операции должна быть объявлена как элемент класса.
- Функции-операции могут быть, а могут и не быть функциями-элементами.
- Если функция-операция реализуется как функция-элемент, крайний левый operand должен быть объектом того класса (или ссылкой на объект того класса), элементом которого является функция.
- Если левый operand должен быть объектом другого класса, эта функция операция должна быть реализована не как функция-элемент.

- Функции-элементы операций вызываются только в случае, если левый операнд бинарной операции или единственный операнд унарной операции являются объектом того класса, элементом которого является функция.
- Можно выбирать для перегрузки операции функцию, не являющуюся элементом, чтобы обеспечить коммутативность операции (т.е. давая соответствующие определения перегруженной операции, можно иметь в качестве левого аргумента операции объект другого типа данных).
- Унарную операцию класса можно перегружать как нестатическую функцию-элемент без аргументов, либо как функцию, не являющуюся элементом, с одним аргументом; этот аргумент должен быть либо объектом класса, либо ссылкой на объект класса.
- Бинарную операцию можно перегружать как нестатическую функцию-элемент с одним аргументом, либо как функцию, не являющуюся элементом, с двумя аргументами (один из этих аргументов должен быть либо объектом класса, либо ссылкой на объект класса).
- Операция индексации массива [] не ограничена применением только к массивам; ее можно использовать для выделения элементов из других видов классов-контейнеров, таких, как связные списки, строки, словари и так далее. Кроме того, индексы — это не обязательно целые числа; можно использовать, например, символы и строки.
- Конструктор копии используется, чтобы в качестве начальных значений элементов одного объекта задать значения элементов другого объекта того же класса. Конструктор копии вызывается всякий раз, когда возникает необходимость в копировании объекта, например, при вызове по значению, когда объект возвращается из вызванной функции, или при инициализации объекта, который должен быть копией другого объекта того же самого класса. Конструкторы копии должны получать свои параметры по ссылке.
- Компилятор не может сам по себе знать, как осуществлять преобразования междустроенными типами и типами, определенными пользователем. Программист должен явно указать, как выполнять такие преобразования. Эти преобразования могут быть выполнены с помощью конструкторов преобразований — конструкторов с единственным аргументом, которые преобразуют объекты разных типов (включая встроенные типы) в объекты данного класса.
- Операция преобразования (называемая также операцией приведения) может быть использована для преобразования объекта одного класса в объект другого класса или в объект встроенного типа. Такая операция преобразования должна быть нестатической функцией-элементом; операция преобразования этого вида не может быть дружественной функцией.
- Конструктор преобразования — это конструктор с единственным аргументом, используемый для преобразования аргумента в объект класса данного конструктора. Компилятор может вызывать такой конструктор неявно.
- Операция присваивания — наиболее часто перегружаемая операция. Обычно она используется для того, чтобы присвоить один объект дру-

гому объекту того же самого класса, но с помощью конструкторов преобразования она может также использоваться для присваиваний объектов различных классов.

- Если перегруженная операция присваивания не определена, присваивание все-таки возможно, но по умолчанию при этом проводится побитовое копирование каждого элемента данных. Для объектов, которые содержат указатели на динамически выделенные области памяти, подобное копирование приводит к тому, что два разных объекта указывают на одну и ту же динамически выделенную область памяти. Выполнение деструктора любого из этих объектов освобождает эту динамически выделенную область памяти. Если теперь обратиться к другому объекту, указывающему на эту освобожденную область памяти, то результат будет непредсказуемым.
- Чтобы перегрузить операцию инкремента для получения возможности использования и префиксной, и постфиксной форм, каждая из этих двух перегруженных функций-операций должна иметь разную сигнатуру, чтобы компилятор имел возможность определить, какая версия `++` имеется в виду в каждом конкретном случае. Префиксный вариант перегружается точно так же, как любая другая префиксная унарная операция. Обеспечение уникальной сигнатуры функции операции постфиксной формы инкремента достигается с помощью фиктивного аргумента типа `int`. Пользователь может не задавать никакого значения этому аргументу. Он вводится просто для того, чтобы помочь компилятору различать префиксные и постфиксные формы операций инкремента и декремента.

Терминология

встроенный тип
класс `Array`
класс `Date`
класс `String`
ключевое слово `operator`
конструктор копии
конструктор преобразования
неперегружаемые операции
неявные преобразования типов
операции, допускающие перегрузку
операция преобразования
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
перегрузка
перегрузка бинарной операции
перегрузка операций
перегрузка постфиксной операции
перегрузка префиксной операции
перегрузка унарной операции
преобразование, определенное
пользователем
преобразования между
встроенными типами и классами
преобразования между типами
классов
цепленные вызовы функций
тип, определенный пользователем
функции-операции
функция преобразования
явные преобразования типа
(с помощью приведений типов)

Типичные ошибки программирования

- 8.1 Попытка перегрузить операцию, запрещенную для перегрузки.
- 8.2 Попытка создавать новые операции.
- 8.3 Попытка изменить работу операции с объектами встроенного типа.
- 8.4 Предположение, что перегрузка операции (такой как `+`) автоматически перегружает связанные с ней операции (такие как `+=`). Операции можно перегружать только явно; неявной перегрузки не существует.
- 8.5 Отсутствие проверки самоприсваивания при перегрузке оператора присваивания для класса, имеющего указатель на динамическую область памяти.
- 8.6 Не предусматривают перегруженную операцию присваивания и конструктор копии для класса, когда объекты этого класса содержат указатели на динамически распределенную память.

Хороший стиль программирования

- 8.1 Используйте перегрузку операций, если она делает программу более ясной по сравнению с применением явных вызовов функций для выполнения тех же операций.
- 8.2 Избегайте чрезмерного или непоследовательного использования перегрузки операций, так как это может сделать программу непонятной и затруднит ее чтение.
- 8.3 Перегружайте операции, чтобы они выполняли над объектами класса ту же функцию или близкие к ней функции, что и операции, выполняемые над объектами встроенных типов.
- 8.4 Перед написанием программы на C++ с перегруженными операциями обратитесь к руководству по вашему компилятору C++, чтобы осознать разнообразные специфические ограничения и требования отдельных операций.
- 8.5 Чтобы обеспечить согласованность связанных операций используйте одни из них для реализации других (т.е. используйте перегруженную операцию `+` для реализации перегруженной операции `+=`).
- 8.6 При перегрузке унарных операций предпочтительнее создавать функции-операции, являющиеся элементами класса, вместо дружественных функций, не являющихся элементами. Дружественных функций и дружественных классов лучше избегать до тех пор, пока они не станут абсолютно необходимыми. Использование друзей нарушает инкапсуляцию класса.
- 8.7 Конструктор, деструктор, перегруженную операцию присваивания и конструктор копии обычно применяют совместно для класса, который использует динамически распределенную память.

Советы по повышению эффективности

- 8.1 Можно было бы перегружать операцию не как элемент и не как дружественную функцию, но такая функция, нуждающаяся в доступе к закрытым или защищенным данным класса, потребовала бы использования функций `set` или `get`, предусмотренных открытым интерфейсом этого класса. Накладные расходы на вызовы этих функций могли бы вызвать ухудшение производительности.
- 8.2 Наличие перегруженной операции сцепления `+=`, которая получала бы один аргумент типа `const char*`, более эффективно, чем выполнение сначала неявного преобразования, а затем сцепления. С другой стороны, неявные преобразования требуют меньшего объема кодирования и вызывают меньше ошибок.

Замечания по технике программирования

- 8.1 Перегрузка операций способствует расширяемости C++, являясь, несомненно, одним из наиболее привлекательных свойств этого языка.
- 8.2 По меньшей мере один аргумент функции-операции должен быть объектом класса или ссылкой на объект класса. Это предохраняет программиста от изменения работы операции с объектом встроенного типа.
- 8.3 Новые возможности ввода-вывода для типов, определенных пользователем, могут быть добавлены в C++ без изменения объявлений или закрытых данных-элементов для классов `ostream` и `istream`. Это еще один пример расширяемости языка программирования C++.
- 8.4 Можно запретить присваивание одного объекта класса другому. Это делается путем определения операции присваивания как закрытого элемента класса.
- 8.5 Можно предохранить объекты класса от копирования; для этого просто делают закрытыми и перегруженную операцию присваивания, и конструктор копии.
- 8.6 При применении конструктора преобразования для выполнения неявного преобразования типов C++ может использовать неявный вызов только одного конструктора, чтобы попытаться удовлетворить требование другой перегруженной операции. Невозможно удовлетворять требования перегруженных операций путем выполнения последовательности неявных определенных пользователем преобразований.

Упражнения для самопроверки

- 8.1 Заполнить пробелы в следующих утверждениях:
 - а) Предположим, что `a` и `b` — целые переменные и мы формируем сумму `a + b`. Теперь предположим, что `c` и `d` — переменные с плавающей запятой и мы формируем сумму `c + d`. Очевидно, что две

операции + использованы здесь в разных целях. Это пример

- b) Ключевое слово _____ вводит определение перегруженной функции-операции.
- c) Для использования операций над объектами класса они должны быть перегружены за исключением операций _____ и _____.
- d) _____ и _____ операции не могут изменяться при ее перегрузке.

- 8.2 Объясните множество значений операций << и >> в C++.
- 8.3 В каком контексте в C++ могло бы быть использовано имя operator/?
- 8.4 Верно или нет, что в C++ можно перегружать только существующие операции?
- 8.5 В каком соотношении находятся старшинство перегруженных и старшинство исходных операций?

Ответы на упражнения для самопроверки

- 8.1 a) перегрузки операций. b) operator. c) присваивания (=), адресации (&). d) Старшинство, ассоциативность.
- 8.2 Операция >> в зависимости от ее контекста может быть и операцией сдвига вправо, и операцией взять из потока. Операция << в зависимости от ее контекста может быть и операцией сдвига влево, и операцией поместить в поток.
- 8.3 При перегрузке операций: это могло бы быть имя функции, которая обеспечивает новую версию операции /.
- 8.4 Верно.
- 8.5 Они идентичны.

Упражнения

- 8.6 Дайте ряд примеров неявной перегрузки операций в С. Дайте ряд примеров неявной перегрузки операций в C++. Дайте убедительный пример ситуации, когда вы хотели бы явно перегрузить операцию в C++.
- 8.7 В C++ нельзя перегружать операции ___, ___, ___, ___ и ___.
- 8.8 Сцепление строк требует двух operandов — двух строк, которые должны быть склеены. Мы показали в тексте, как реализовать перегруженную операцию склейки, которая подцепляет второй объект класса String справа к первому объекту класса String, изменяя при этом первый объект. В некоторых приложениях желательно создать склеенный объект класса String, не изменяя ни одного из двух склеиваемых объектов. Реализуйте функцию operator+, способную выполнять такие операции как

```
string1 = string2 + string3;
```

8.9 (*Упражнение на перегрузку основных операций*). Чтобы прочувствовать осторожность, с которой надо подходить к выбору операций для перегрузки, составьте список перегруженных операции C++ и для каждой из них перечислите их возможные значения. Сделайте то же самое для нескольких перечисленных ниже классов, изученных вами в этом курсе:

- a) Массив
- b) Стек
- c) Стока

После того, как вы это сделаете, прокомментируйте, какие операции имеют значение для большинства классов. Какие операции представляются наименее важными для перегрузки? Какие операции представляются сомнительными?

8.10 Теперь подойдите к предыдущей задаче с другой стороны. Составьте список перегруженных операции C++. Для каждой из них перечислите, какие варианты этих «основных операций», как вам кажется, было бы полезно иметь.

8.11 (*Проект*) C++ — эволюционирующий язык. Как и все новые языки он постоянно совершенствуется. Какие дополнительные операции вы бы рекомендовали добавить в C++ или в будущий язык, подобный C++; эти операции должны поддерживать как процедурное, так и объектно-ориентированное программирование? Напишите тщательное обоснование. Вы могли бы попробовать послать ваши предложения в комитет ANSI по C++.

8.12 Прекрасным примером перегрузки операции вызова функции () является возможность более общей формы двойной индексации массива. Перегрузите операцию вызова функции так, чтобы вместо записи массива в виде

```
chessBoard[row][column]
```

можно было бы записывать альтернативную форму:

```
chessBoard[row, column]
```

8.13 Перегрузите операцию индексации так, чтобы она возвращала заданный элемент связного списка.

8.14 Перегрузите операцию индексации так, чтобы она возвращала наибольший элемент набора, второй наибольший, третий наибольший и т.д.

8.15 Рассмотрите класс **Complex**, показанный на рис.8.7. Класс позволяет работать с операциями над так называемыми *комплексными числами*. Они представляются в виде **realPart + imaginaryPart*i**, где **i** имеет значение корня квадратного из -1.

- а) Измените этот класс так, чтобы иметь возможность вводить и выводить комплексные числа посредством перегруженных операций >> и << соответственно (вы должны заменить функцию печати класса).
- б) Перегрузите операцию умножения так, чтобы иметь возможность перемножать два комплексных числа, как в алгебре.

c) Перегрузите операции == и != так, чтобы иметь возможность сравнивать два комплексных числа.

```
// 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;                            // печать
private:
    double real;                                // действительная часть
    double imaginary;                           // мнимая часть
};

#endif
```

Рис. 8.7. Определение класса **Complex** (часть 1 из 5)

```
// COMPLEX1.CPP
// Определения функций-элементов 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;
}
```

Рис. 8.7. Определения функций-элементов класса **Complex** (часть 2 из 5)

```
// Перегруженная операция присваивания
Complex& Complex::operator=(const Complex &right)
{
    real = right.real;
    imaginary = right.imaginary;
    return *this;                                // возможность сцепления
}

// Отображение объекта Complex в форме: (a, b)
void Complex::print() const
{ cout << '(' << real << ", " << imaginary << ')'; }
```

Рис. 8.7. Определения функций-элементов класса **Complex** (часть 3 из 5)

```
// FIGS 7.CPP
// Драйвер 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 << endl << "y: ";
    y.print();
    cout << endl << "z: ";
    z.print();

    x = y + z;
    cout << endl << endl << "x = y + z: " << endl;
    x.print();
    cout << " = ";
    y.print();
    cout << " + ";
    z.print();

    x = y - z;
    cout << endl << endl << "x = y - z: " << endl;
    x.print();
    cout << " = ";
    y.print();
    cout << " - ";
    z.print();
    cout << endl;

    return 0;
}
```

Рис. 8.7. Драйвер для класса **Complex** (часть 4 из 5)

```

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. 8.7. Выходные данные драйвера класса **Complex** (часть 5 из 5)

8.16 Машина с 32-битовыми целыми может представлять целые в диапазоне примерно от -2 миллиардов до +2 миллиардов. Ограничение этого фиксированного диапазона редко доставляет беспокойство. Но существует много приложений, в которых вы хотели бы иметь возможность представления целых в гораздо более широком диапазоне. Вот для этого и был создан C++ — для создания мощных новых типов данных. Рассмотрите класс **HugeInt** на рис.8.8. Тщательно изучите класс, а затем:

- Точно опишите, как он работает.
- Какие ограничения имеет класс?
- Измените класс так, чтобы можно было работать с произвольно большими числами. (Подсказка: используйте связанный список для представления **HugeInt**).
- Перегрузите операцию умножения *.
- Перегрузите операцию деления /.
- Перегрузите операции отношения и проверки на равенство.

```

// 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 &); // сложение
private:
    short integer[30];
};

#endif

```

Рис. 8.8. Определенный пользователем класс больших целых (часть 1 из 4)

```
// HUGEINT1.CPP
// Определения функций элементов и дружественных функций
// класса HugeInt
#include <string.h>
#include "hugeint1.h"

// конструктор преобразования
HugeInt::HugeInt(long val)
{
    for (int i = 0; i <= 29; i++)
        integer[ i ] = 0;           // задание массиву нулевых
                                     // начальных значений
    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;

    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
```

Рис. 8.8. Определенный пользователем класс больших целых (часть 2 из 4)

```

if ( i == 30)
    output << 0;
else
    for ( ; i <= 29; i++)
        output << num.integer[ i ];

return output;
}

```

Рис. 8.8. Определенный пользователем класс больших целых (часть 3 из 4)

```

// FIGS_8.CPP
// Проверка драйвера класса HugeInt
#include <iostream.h>
#include "hugeint1.h"

main()
{
    HugeInt    n1(7654321), n2{7891234},
                n3("99999999999999999999999999999999"),
                n4 (" 1 "), n5;

    cout << "n1 равен " << n1 << endl << "n2 равен " << n2
        << endl << "n3 равен " << n3 << endl << "n4 равен " << n4
        << endl << "n5 равен " << n5 << endl << endl;

    n5 = n1 + n2;
    cout << n1 << " + " << n2 << " = " << n5 << endl << endl;

    cout << n3 << " + " << n4 << endl << "= " << (n3 + n4)
        << endl << endl;

    n5 = n1 + 9;
    cout << n1 << " + " << 9 << " = " << n5 << endl << endl;

    n5 = n2 + "10000";
    cout << n2 << " + " << "10000" << " = " << n5 << endl
        << endl;

    return 0;
}

```

```

n1 равен 7654321
n2 равен 7891234
n3 равен 99999999999999999999999999999999
n4 равен 1
n5 равен 0

```

7654321 + 7891234 = 15545555

99999999999999999999999999999999 + -1
= 100000000000000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234

Рис. 8.8. Определенный пользователем класс больших целых (часть 4 из 4)

8.17 Создайте класс `rationalNumber` (дроби) со следующими возможностями:

- Создайте конструктор, который предотвращает равенство нулю знаменателя дроби, сокращает или упрощает дроби, если они не в сокращенной форме, и исключает отрицательные знаменатели.
- Перегрузите операции сложения, вычитания, умножения и деления для этого класса.
- Перегрузите операции отношения и проверки на равенство для этого класса.

8.18 Изучите функции библиотеки обработки строк С и реализуйте каждую из этих функций как часть класса `String`. Используйте затем эти функции для выполнения операций с текстами.

8.19 Разработайте класс `Polinomials` (полиномы). Внутренним представлением класса `Polinomials` является массив членов полинома. Каждый член содержит коэффициент и показатель степени. Член $2x^4$

имеет коэффициент 2 и показатель степени 4. Разработайте полный класс, содержащий соответствующие функции конструктора, деструктора, а также функции `set` и `get`. Класс должен обеспечивать путем использования перегруженных операций следующие возможности:

- Перегрузить операцию сложения (+), чтобы складывать два объекта класса `Polinomials`.
- Перегрузить операцию вычитания (-), чтобы вычесть два объекта класса `Polinomials`.
- Перегрузить операцию присваивания (=), чтобы присваивать один объект класса `Polynomial` другому.
- Перегрузить операцию умножения (*), чтобы перемножать два объекта класса `Polinomials`.
- Перегрузить операцию сложения с присваиванием (+=), операцию вычитания с присваиванием (-=), операцию умножения с присваиванием (*=).

9

Наследование



Цели

- Научиться создавать новые классы путем наследования существующих классов.
- Понять, как наследование способствует повторному использованию программного обеспечения.
- Понять формы записи базовых классов и производных классов.
- Научиться использовать множественное наследование для порождения классов из нескольких основных классов.

План

- 9.1. Введение
- 9.2. Базовые классы и производные классы
- 9.3. Защищенные элементы
- 9.4. Приведение типов указателей базовых классов к указателям производных классов
- 9.5. Использование функций-элементов
- 9.6. Переопределение элементов базового класса в производном классе
- 9.7. Открытые, защищенные и закрытые базовые классы
- 9.8. Прямые и косвенные базовые классы
- 9.9. Использование конструкторов и деструкторов в производных классах
- 9.10. Неявное преобразование объектов производных классов в объекты базовых классов
- 9.11. Проектирование программного обеспечения с помощью наследования
- 9.12. Композиция и наследование
- 9.13. Отношения «использует А» и «знает А»
- 9.14. Учебный пример: точка, круг, цилиндр
- 9.15. Множественное наследование

Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

9.1. Введение

В этой и следующей главах мы обсудим две наиболее важных особенности объектно-ориентированного программирования — *наследование* и *полиморфизм*. Наследование — это способ повторного использования программного обеспечения, при котором новые классы создаются из уже существующих классов путем заимствования их атрибутов и функций и обогащения этими возможностями новых классов. Повторное использование кодов экономит время при разработке программ. Это способствует повторному использованию

проверенного и отлаженного программного обеспечения и таким образом уменьшает число проблем, возникающих после того, как система начинает функционировать. Это захватывающие возможности. Полиморфизм позволяет нам обычным образом писать программы, обрабатывающие широкий круг как существующих, так и еще даже не описанных родственных классов. Наследование и полиморфизм представляют собой эффективную технику преодоления сложности программного обеспечения.

При создании нового класса вместо написания полностью новых данных-элементов и функций-элементов программист может указать, что новый класс является *наследником* данных-элементов и функций-элементов ранее определенного *базового класса*. Этот новый класс называется *производным классом*. Каждый производный класс сам является кандидатом на роль базового класса для будущих производных классов. При *простом наследовании* класс порождается *одним базовым классом*. При *множественном наследовании* производный класс наследует нескольким базовым классам (возможно неродственным).

Производный класс обычно добавляет свои собственные данные-элементы и функции-элементы, так что производный класс в общем случае больше своего базового класса. Производный класс более специфичен по своему назначению, более узок, чем его базовый класс, и представляет меньшую группу объектов. При простом наследовании предполагается, что производный класс будет выполнять примерно те же функции, что и базовый класс. Истинная сила наследования заключается в способности определять в производном классе добавления, замены или усовершенствования черт, унаследованных от базового класса.

Каждый объект производного класса является также объектом соответствующего базового класса. Однако, обратное неверно: объект базового класса не является объектом классов, порожденных этим базовым классом. Мы будем использовать достоинства этой связи — «объект производного класса является объектом базового класса» — для выполнения некоторых интересных операций. Например, мы можем объединить широкий круг различных объектов, связанных наследованием, в связный список объектов базового класса. Тогда мы сможем обрабатывать разные объекты одним общим способом. Как мы увидим в этой и следующей главах, это является ключевым моментом *объектно-ориентированного программирования*.

В данной главе мы узнаем о новой форме управления доступом к элементам, а именно, о *защищенном доступе*. Производные классы и их друзья получают приоритет перед другими функциями в доступе к защищенным элементам базового класса.

Опыт построения систем программного обеспечения показывает, что имеются большие фрагменты программы, которые имеют дело с тесно связанными частными случаями. В таких системах становится трудно видеть «картину в целом», потому что проектировщик и программист оказываются озабоченными частными специальными случаями. *Объектно-ориентированное программирование* обеспечивает несколько способов «увидеть лес за деревьями» — этот процесс иногда называют *абстрагированием*.

Если программа насыщена тесно связанными частными случаями, то в ней часто встречается оператор *переключения switch*, который отделяет частные случаи друг от друга и обеспечивает логику их индивидуальной обработки. В главе 10 мы покажем, как использовать наследование и полиморфизм для замены этой логики переключений более простой логикой.

Мы делаем различие между отношениями «является» и «содержит». «Является» — это наследование. При отношении «является» объект типа производного класса является также объектом типа базового класса (может рассматриваться как такой объект). «Содержит» — это композиция (см. рис. 7.4). При отношении «содержит» объект класса включает один или более объектов других классов в качестве своих элементов.

Производный класс не может иметь доступ к закрытым элементам своего базового класса; разрешение такого доступа явилось бы нарушением инкапсуляции базового класса. Производный класс может, однако, иметь доступ к открытым и защищенным элементам своего базового класса. Элементы базового класса, которые не должны быть доступны производному классу через наследование, объявляются в базовом классе закрытыми. Производный класс может иметь доступ к закрытым элементам своего базового класса только посредством функций доступа, предусмотренных в открытом интерфейсе базового класса.

Одной из проблем наследования является то, что производный класс может наследовать открытые функции-элементы, когда в этом нет необходимости или когда это недопустимо. Имеется возможность избежать этого. Когда какой-то элемент базового класса не подходит для задач производного класса, этот элемент может быть соответствующим образом переопределён в производном классе.

Может быть, наиболее захватывающей является идея наследования новыми классами существующих библиотек классов. Организации разрабатывают свои собственные библиотеки классов и могут при этом заимствовать достоинства различных распространенных в мире библиотек. В скором времени программное обеспечение будет конструироваться из стандартизованных повторно используемых компонентов точно так же, как сегодня конструируется аппаратное обеспечение. Это поможет ответить на будущие требования разработки все более мощного программного обеспечения.

9.2. Базовые классы и производные классы

Часто объекты одного класса на самом деле являются также и объектами другого класса. Прямоугольник является также и четырехугольником (как и квадрат, и параллелограмм, и трапеция). Таким образом, можно сказать, что класс `Rectangle` (прямоугольник) является *наследником* класса `Quadrilateral` (четырехугольник). В этом контексте класс `Quadrilateral` является *базовым классом*, а класс `Rectangle` — *производным классом*. Прямоугольник является специальным типом четырехугольника, но неверно утверждать, что четырехугольник является прямоугольником. На рис. 9.1 показано несколько простых примеров наследования.

Другие объектно-ориентированные языки программирования, такие как `Smalltalk`, используют отличную от данной терминологию: в наследовании базовый класс называется *суперклассом* (представляет супермножество объектов), а производный класс называется *подклассом* (представляет подмножество объектов). Поскольку наследование обычно создает производный класс с *большим* количеством свойств, чем базовый класс, термины суперкласс и подкласс могут вызвать путаницу; мы будем избегать этих терминов.

Наследование формирует древовидные иерархические структуры. Базовый класс находится в иерархических отношениях со своими производными

классами. Класс, конечно, может существовать сам по себе, но когда класс используется в механизме наследования, этот класс становится либо базовым классом, который снабжает атрибутами и функциями другие классы, либо производным классом, который наследует эти атрибуты и функции.

Разработаем простую иерархию наследования. Типичное университетское сообщество состоит из тысяч людей, являющихся его членами. Эти люди делятся на студентов и служащих. Служащие — либо члены факультета, либо вспомогательный персонал. Члены факультета — либо администраторы (деканы и начальники курсов), либо преподаватели факультета. Это дает иерархию наследования, показанную на рис. 9.2.

Базовый класс	Производный класс
Студент	Выпускник Не окончивший институт
Форма	Круг Треугольник Прямоугольник
Ссуда	Ссуда на автомобиль Ссуда на ремонт дома Ипотечная ссуда
Служащий	Член факультета Вспомогательный персонал
Счет	Чековый счет Сберегательный счет

Рис. 9.1. Некоторые простые примеры наследования

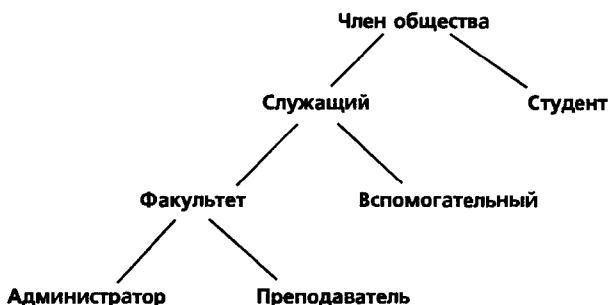


Рис. 9.2. Иерархия наследования для членов университетского сообщества

Другой реальной иерархией наследования является иерархия Shape (форма) на рис. 9.3. Типичным для студентов, впервые изучающих объектно-ориентированное программирование, является то, что они не привыкли воспринимать богатство примеров иерархии наследования в реальном мире в рассматриваемых нами категориях; так что нужна некоторая корректировка их мышления.

Чтобы указать, что класс **CommissionWorker** порожден классом **Employee**, класс **CommissionWorker** должен быть определен следующим образом:

```

class CommissionWorker : public Employee {
    ...
};
  
```

Это называется *открытым наследованием* (*public inheritance*). Мы обсудим также *закрытое наследование* (*private inheritance*) и *защищенное наследование* (*protected inheritance*). При открытом наследовании открытые и защищенные элементы базового класса наследуются как открытые и защищенные элементы производного класса соответственно. Помните, что закрытые элементы базового класса не доступны в производных классах.

Можно рассматривать объекты базового и производного классов как подобные; их общность выражается в атрибутах и функциях базового класса. Все объекты любых классов, открыто порожденных общим базовым классом, могут рассматриваться как объекты этого базового класса.

Мы рассмотрим много примеров, в которых мы сможем на основе этого отношения добиться простоты программирования, что недоступно для не объектно-ориентированных языков, таких, как С.



Рис. 9.3. Часть иерархии класса **Shape**

9.3. Защищенные элементы

Открытые элементы базового класса доступны всем функциям программы. Закрытые элементы базового класса доступны только функциям-элементам и друзьям базового класса.

Защищенный уровень доступа служит промежуточным уровнем защиты между открытым доступом и закрытым доступом. Защищенные элементы базового класса могут быть доступны только элементам и друзьям базового класса и элементам и друзьям производного класса.

Элементы производного класса могут ссылаться на открытые и защищенные элементы базового класса простым использованием имен этих элементов. Защищенные данные «взламывают» инкапсуляцию: изменение защищенных элементов базового класса может потребовать модификации всех производных классов.

9.4. Приведение типов указателей базовых классов к указателям производных классов

Объекты открыто порожденного класса могут также рассматриваться как объекты соответствующего ему базового класса. Это ведет к некоторым интересным следствиям. Например, вопреки тому факту, что объекты различных классов, порожденных одним базовым классом, могут существенно отличаться друг от друга, мы можем создать их связный список, поскольку мы рассмат-

риваем их как объекты базового класса. Но обратное неверно: объекты базового класса не являются автоматически объектами производного класса.

Типичная ошибка программирования 9.1

Рассмотрение объектов базового класса как объектов производного класса может вызвать ошибки.

Программист может, однако, использовать явное приведение типов для преобразования указателей базового класса в указатели производного класса. Но будьте внимательны: если такой указатель должен разыменовываться, программисту надо быть уверенным, что тип указателя соответствует типу объекта, на который он указывает.

Типичная ошибка программирования 9.2

Явное приведение типа указателя базового класса, который указывает на объект базового класса, к типу указателя производного класса и затем ссылка на элементы производного класса, которые не существуют в этом объекте.

Наш первый пример показан на рис. 9.4, части с 1 по 5. Части 1 и 2 показывают определение класса **Point** (точка) и определения функций-элементов класса **Point**. Части 3 и 4 показывают определение класса **Circle** (круг) и определение функций-элементов класса **Circle**. Часть 5 показывает программу драйвер, в которой мы демонстрируем присваивание указателей производного класса указателям базового класса и приведение типов указателей базового класса к указателям производного класса.

Исследуем сначала определение класса **Point** (Рис. 9.4, часть 1). Открытый интерфейс класса **Point** содержит функции-элементы **setPoint**, **getX** и **getY**. Данные-элементы **x** и **y** класса **Point** указаны как **protected** — защищенные. Это запрещает клиентам объектов класса **Point** прямой доступ к данным, но позволяет классам, порожденным классом **Point**, иметь непосредственный доступ к унаследованным данным-элементам. Если бы эти данные были определены как **private**, то для доступа к ним даже в производном классе надо было использовать открытые функции-элементы класса **Point**.

```
// 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 с умолчанием
    void setPoint(float, float);           // установка координат
    float getX() const { return x; }       // получение координаты x
    float getY() const { return y; }       // получение координаты y
protected:
    float x, y;                          // доступно для производных классов
};                                     // координаты Point x и y

#endif
```

Рис. 9.4. Определение класса **Point** (часть 1 из 5)

Заметим, что перегруженная функция-операция поместить в поток (рис. 9.4, часть 2) способна ссылаться на переменные *x* и *y* непосредственно, так как эта перегруженная функция-операция является другом класса *Point*. Заметим также, что ссылаться на *x* и *y* необходимо через объекты, например, *p.x* и *p.y*. Это объясняется тем, что перегруженная функция-операция поместить в поток не является функцией-элементом класса *Point*.

```
// POINT.CPP
// Функции-элементы класса Point
#include <iostream.h>
#include "point.h"

// Конструктор класса Point
Point::Point(float a, float b) { setPoint(a, b); }

// Установка координат x и y в Point
void Point::setPoint(float a, float b)
{
    x = a;
    y = b;
}

// Вывод данных Point (с помощью операции поместить в поток)
ostream &operator<<(ostream &output, const Point &p)
{
    output << '[' <<p.x << ", " <<p.y << ']';
    return output; // возможность сцепленных вызовов
}
```

Рис. 9.4. Определения функций-элементов класса **Point** (часть 2 из 5)

```
// 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 Point
    friend ostream &operator<<(ostream &, const Circle &);
public:
    // конструктор с умолчанием
    Circle(float r = 0.0, float x = 0, float y = 0);

    void setRadius(float); // установка радиуса
    float getRadius() const; // возвращение радиуса
    float area() const; // вычисление площади
protected:
    float radius;
};

#endif
```

Рис. 9.4. Определение класса **Circle** (часть 3 из 5)

```

// CIRCLE.CPP
// Определения функций-элементов класса Circle
#include "circle.h"

// Конструктор Circle вызывает конструктор Point
// с инициализаторами элементов, затем инициализирует радиус.
Circle::Circle(float r, float a, float b)
    : Point (a, b)           // вызов конструктора базового класса
{ radius = r; }

// Установка радиуса круга
void Circle::setRadius(float r) { radius = r; }

// Получение радиуса круга
float Circle::getRadius() const { return radius; }

// Вычисление площади круга
float Circle::area() const
{ return 3.14159 * radius * radius; }

// Вывод данных Circle в форме:
// Центр = [x, y]; Радиус = #.##
ostream &operator << (ostream &output, const Circle &c)
{
    output << "Центр = [" << c.x << ", " << c.y
           << "]; Радиус = " << setiosflags(ios::showpoint)
           << setprecision(2) << c.radius;
    return output;           // возможность сцепленных вызовов
}

```

Рис. 9.4. Определение функций-элементов класса **Circle** (часть 4 из 5)

Класс **Circle** (рис. 9.4, часть 3) наследует классу **Point** путем открытого наследования. Это указано в первой строке определения класса

```
class Circle : public Point {           // Circle наследует Point
```

Двоеточие в заголовке определения класса указывает наследование. Ключевое слово **public** указывает тип наследования (смотри раздел 9.7). Все элементы класса **Point** унаследованы в классе **Circle**. Это означает, что открытый интерфейс класса **Circle** включает открытые функции-элементы класса **Point** и добавляет свои функции-элементы **area**, **setRadius** и **getRadius**.

Конструктор **Circle** (рис. 9.4, часть 4) должен активизировать конструктор **Point** для задания начальных значений той части объекта **Circle**, которая относится к базовому классу. Это осуществляется списком инициализаторов элементов, рассмотренным в главе 7:

```
Circle::Circle(float r, float a, float b)
    : Point (a, b)           // вызов конструктора базового класса
```

Вторая строка заголовка функции конструктора активизирует конструктор класса **Point** по имени. Значения **a** и **b** передаются из конструктора класса **Circle** конструктору класса **Point**, чтобы задать начальные значения элементам **x** и **y** базового класса. Если конструктор **Circle** не активизирует конструктор **Point** явно, то конструктор **Point** активизируется со значениями **x** и **y** по умолчанию (т.е. 0 и 0). Если класс **Point** не снабжен конструктором с умолчанием, компилятор генерирует сообщение об ошибке.

```

// FIG9_4.CPP
// Приведение типов указателей базового класса к
// типам указателей производного класса
#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 << endl << "Circle c: " << c << endl;

    // Рассмотрение Circle как Point (видит только часть
    // базового класса)
    pointPtr = &c;           // присваивание адреса Circle
                            // указателю pointPtr
    cout << endl << "Circle c (через *pointPtr): "
        << *pointPtr << endl;

    // Рассмотрение Circle как Point (с помощью приведения типов)
    pointPtr = &c;           // присваивание адреса Circle
                            // указателю pointPtr
    circlePtr = (Circle *) pointPtr; // приведение типов
                                    // базового к производному
    cout << endl << "Circle c (через *circlePtr): " << endl
        << *circlePtr << endl << "Площадь с (через circlePtr): "
        << circlePtr->area() << endl;

    // ОПАСНОСТЬ: рассмотрение Point как Circle
    pointPtr = &p;           // присваивание адреса Point
                            // указателю pointPtr
    circlePtr = (Circle *) pointPtr; // приведение типов
                                    // базового к производному
    cout << endl << "Point p (через *circlePtr): "
        << endl << *circlePtr << endl
        << "Площадь объекта circlePtr указывает на: "
        << circlePtr->area() << endl;

    return 0;
}

```

```

Point p: [3.5, 5.3]
Circle c: Центр = [1.2, 8.9]; Радиус = 2.70

```

```

Circle c (через *pointPtr): [1.20, 8.90]

```

```

Circle c (через *circlePtr):
Центр = [1.20, 8.90]; Радиус = 2.70
Площадь с (через circlePtr): 22.90

```

```

Point p (через *circlePtr):
Центр = [3.50, 5.30]; Радиус = 4.02e-38
Площадь объекта circlePtr указывает на: 0.00

```

Рис. 9.4. Приведение типов указателей базового класса к типам указателей производного класса (часть 5 из 5)

Заметим, что перегруженная функция-операция поместить в поток класса `Circle` способна ссылаться на `x` и `y` непосредственно, потому что они являются защищеннымми элементами базового класса `Point`. Заметим также, что ссылаться на `x` и `y` необходимо через объекты, например, `s.x` и `s.y`. Это объясняется тем, что перегруженная функция-операция поместить в поток является не элементом класса `Circle`, а другом этого класса.

Программа драйвер (рис. 9.4, часть 5) создает `pointPtr` — указатель на объект класса `Point`, и создает экземпляр класса `Point` — объект `r`. Затем создается `circlePtr` — указатель на объект класса `Circle`, и создается `c` — объект класса `Circle`. Объекты `r` и `c` выводятся с помощью перегруженных операций поместить в поток, чтобы показать, что они инициализированы правильно.

Далее драйвер демонстрирует присваивание указателя производного класса (адреса объекта `c`) указателю базового класса `pointPtr` и выводит объект с класса `Circle`, используя перегруженную операцию поместить в поток для класса `Point` и разыменование указателя `*pointPtr`. Отметим, что отображается только часть объекта с класса `Circle`, относящаяся к классу `Point`. Присваивание указателя производного класса указателю базового класса всегда выполняется правильно, поскольку объект производного класса *является* в то же самое время объектом базового класса. Указатель базового класса «видит» только часть производного класса, относящуюся к базовому классу. Компилятор выполняет неявное преобразование указателя производного класса в указатель базового класса.

Далее драйвер демонстрирует присваивание указателя производного класса (адреса объекта `c`) указателю базового класса `pointPtr` и затем приведение типа `pointPtr` обратно к `Circle *`. Результат этой операции приведения типа присваивается `circlePtr`. Объект с класса `Circle` выводится, с помощью перегруженной операции поместить в поток класса `Circle` и разыменования указателя `*circlePtr`. Площадь объекта с класса `Circle` выводится через `circlePtr`. Результат содержит правильное значение площади, потому что эти указатели всегда указывают на объект класса `Circle`.

Указатель базового класса не может быть непосредственно присвоен указателю производного класса, потому что это опасное присваивание: компилятор считает, что указатели производного класса должны указывать на объекты производного класса. В подобном случае компилятор не выполнит неявное преобразование. Использование операции явного приведения типов информирует компилятор, что программист знает об опасности такого типа преобразования указателей и несет ответственность за неправильное использование указателя.

Далее драйвер демонстрирует присваивание указателя базового класса (адреса объекта `r`) указателю базового класса `pointPtr` и обратное приведение `pointPtr` к типу `Circle*`. Результат операции приведения типа присваивается указателю `circlePtr`. Объект `r` класса `Point` выводится с помощью перегруженной операции поместить в поток для класса `Circle` и разыменования указателя `*circlePtr`. Отметим странное выходное значение для элемента `radius`. Выведение объекта класса `Point` как `Circle` дает в результате неправильное значение для `radius`, потому что указатель указывает на объект класса `Point`. А объект класса `Point` не имеет элемента `radius`. Поэтому программа выводит некоторое значение, оказавшееся в памяти в той области, в которой `circlePtr` ожидает элемент данных `radius`. Площадь объекта, на который указывает `circlePtr` (объект `r` класса `Point`), также выводится через `circlePtr`. Отметим,

что значение площади равно 0.00, потому что ее вычисление основано на несуществующем значении `radius`. Очевидно, что доступ к данным-элементам, которых в действительности нет, в данном случае не опасен. Но вызов несуществующих функций-элементов может вызвать необратимую ошибку в программе.

9.5. Использование функций-элементов

Функции-элементы производного класса могут нуждаться в доступе к определенным функциям-элементам и данным-элементам базового класса.

Замечание по технике программирования 9.1

Производный класс не имеет прямого доступа к закрытым элементам своего базового класса.

Это решающий аспект разработки программного обеспечения на C++. Если бы закрытые элементы базового класса были доступны производному классу, это разрушило бы инкапсуляцию базового класса. Недоступность закрытых элементов — огромное подспорье в тестировании, отладке и правильной модификации систем. Если бы закрытые элементы базового класса были доступны производному классу, то доступ к ним стал бы возможным для классов, порожденных этим производным классом и так далее. Это передавало бы по наследству доступ к тому, что по замыслу должно быть закрытыми данными, и выгоды инкапсуляции были бы потеряны из-за иерархии классов.

9.6. Переопределение элементов базового класса в производном классе

Производный класс может переопределить функцию-элемент базового класса. При описании в производном классе функции с тем же именем, версия функции производного класса переопределяет версию базового класса. Чтобы сделать доступной для производного класса версию функции базового класса, нужно использовать операцию разрешения области действия.

Типичная ошибка программирования 9.3

При переопределении в производном классе функции-элемента базового класса при-нято вызывать версию базового класса и после этого выполнять некоторые дополнительные операции. При этом ссылка на функцию-элемент базового класса без использования операции разрешения области действия может вызвать бесконечную рекурсию, потому что функция-элемент производного класса будет в действительности вызывать сама себя.

Замечание по технике программирования 9.2

При переопределении в производном классе функции-элемента базового класса нет необходимости получать такую же сигнатуру, как у функции-элемента базового класса.

Рассмотрим упрощенный класс **Employee** (служащие). Он хранит имена **firstName** и фамилии **lastName** служащих. Эта информация — общая для всех служащих, включая находящихся в классах, порожденных из класса **Employee**. На основе класса **Employee** порождены классы **HourlyWorker** (работники с почасовой оплатой), **PieceWorker** (работники со сделкой оплатой), **Boss** (управляющие) и **CommissionWorker** (работники с комиссионной оплатой). **HourlyWorker** получают почасовую оплату из расчета 40 часов в неделю и полуторную почасовую оплату за работу сверх 40 часов. **PieceWorker** работают сделко и получают фиксированную плату за единицу произведенной продукции (для простоты допустим, что эти работники делают только один тип продукции, так что закрытыми данными-элементами являются количество произведенных единиц продукции и ставка за единицу продукции). **Boss** получает фиксированную зарплату за неделю. **CommissionWorker** получает маленькую еженедельную фиксированную основную зарплату плюс процент от своих оптовых продаж за неделю. Для простоты мы изучим только класс **Employee** и производный класс **HourlyWorker**.

Наш следующий пример показан на рис. 9.5, части с 1 по 5. Части 1 и 2 показывают определение класса **Employee** и определения функций-элементов **Employee**. Части 3 и 4 показывают определение класса **HourlyWorker** и определение функции-элемента **HourlyWorker**. Часть 5 показывает программу драйвер для иерархии наследования **Employee** — **HourlyWorker**, которая просто создает объект **HourlyWorker**, задает ему начальные значения и вызывает функцию-элемент **print** класса **HourlyWorker** для вывода данных объекта.

Определение класса **Employee** (рис. 9.5, часть 1) содержит два закрытых элемента данных типа **char*** — **firstName** и **lastName**, и три функции-элемента: конструктор, деструктор и **print**. Функция конструктор (рис. 9.5, часть 2) получает две строки и динамически выделяет память для их хранения. Заметим, что макрос **assert** (обсуждается в главе 18 «Другие темы») используется для того, чтобы определить, была ли выделена память для **firstName** и **lastName**. Если нет, программа завершается сообщением об ошибке, указывающим проверенное условие, номер строки, в которой появилось условие, и файл, в котором локализовано условие.

```
// 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 размещенная строка
    char *lastName; //динамически размещенная строка
};

#endif
```

Рис. 9.5. Определение класса **Employee** (часть 1 из 5)

```

// EMPLOY.CPP
// Определения функций-элементов класса Employee.
#include <string.h>
#include <iostream.h>
#include <assert.h>
#include "employ.h"

// Конструктора динамически распределяет память для имени
// и фамилии и использует strcpy для их копирования в объект.
Employee::Employee(const char *first, const char *last)
{
    firstName = new char[ strlen(first) + 1 ];
    assert(firstName != 0); // завершение, если память
                           // не выделена
    strcpy(firstName, first);

    lastName = new char[ strlen(last) + 1 ];
    assert(lastName != 0); // завершение, если память
                           // не выделена
    strcpy(lastName, last);
}

// Вывод имени служащего
void Employee::print() const
{ cout << firstName << ' ' << lastName; }

// Деструктор освобождает динамически выделенную память
Employee::~Employee()
{
    delete [] firstName; //освобождение динамической памяти
    delete [] lastName; //освобождение динамической памяти
}

```

Рис. 9.5. 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); // вычисление и возвращение
    float getPay() const; // зарплаты
    void print() const; // переопределенная печать
                        // базового класса
private:
    float wage; // зарплата за час
    float hours; // рабочие часы за неделю
};

#endif

```

Рис. 9.5. Определение класса **HourlyWorker** (часть 3 из 5)

```

// HOURLY_B.CPP
// Определения функций-элементов класса HourlyWorker
#include <iostream.h>
#include <iomanip.h>
#include "hourly.h"

// Конструктор класса HourlyWorker
HourlyWorker::HourlyWorker(const char *first, const char *last,
                           float initHours, float initWage)
    : Employee(first, last)                         // вызов конструктора
                                                    // базового класса
{
    hours = initHours;
    wage = initWage;
}

// получение оплаты HourlyWorker
float HourlyWorker::getPay() const { return wage * hours; }

// печать имени и оплаты
void HourlyWorker::print() const
{
    cout << "HourlyWorker::print()" << endl;

    Employee::print();                            // вызов функции print базового класса

    cout << " является рабочим почасовиком с оплатой "
         << "$" << setiosflags(ios::fixed | ios::showpoint)
         << setprecision(2) << getPay() << endl;
}

```

Рис. 9.5. Определения функций-элементов **HourlyWorker** (часть 4 из 5)

```

// FIG9_5.CPP
// Переопределение функции-элемента базового класса
// в производном классе
#include <iostream.h>
#include "hourly.h"

main()
{
    HourlyWorker h("Боб", "Смит", 40.0, 7.50);
    h.print();
    return 0;
}

HourlyWorker::print()
Боб Смит является рабочим почасовиком с оплатой $300.00

```

Рис. 9.5. Переопределение функции-элемента базового класса в производном классе (часть 5 из 5)

Поскольку данные класса **Employee** имеют тип **private**, доступ к данным возможен только посредством функции-элемента **print**, которая просто выводит имя и фамилию служащего. Функция деструктор возвращает системе динамически выделенную память.

Класс `HourlyWorker` (рис. 9.5, часть 3) наследует классу `Employee` путем открытого наследования. Это указано в первой строке определения класса в следующем виде:

```
class HourlyWorker : public Employee
```

Открытый интерфейс `HourlyWorker` включает функцию `print` класса `Employee` и функции-элементы `getPay` и `print` класса `HourlyWorker`. Обратите внимание, что класс `HourlyWorker` определяет свою собственную функцию `print`. Поэтому класс `HourlyWorker` имеет доступ к двум функциям `print`. Класс `HourlyWorker` содержит также закрытые данные-элементы `wage` и `hours` для вычисления недельной зарплаты служащих.

Конструктор класса `HourlyWorker` (рис. 9.5, часть 4) использует список инициализаторов элементов для передачи строк `first` и `last` конструктору класса `Employee` для задания начальных значений элементам базового класса, а затем задает начальные значения элементам `wage` и `hours`. Функция-элемент `getPay` вычисляет зарплату `HourlyWorker`.

Функция-элемент `print` класса `HourlyWorker` является примером функции-элемента базового класса, переопределенной в производном классе. Функции-элементы базового класса часто переопределяются в производном классе для выполнения каких-то специфических операций. Переопределенные функции-элементы иногда вызывают версию функции базового класса, чтобы выполнить часть новой задачи. В этом примере функция `print` производного класса вызывает функцию `print` базового класса чтобы вывести имя служащего (функция `print` базового класса — единственная функция, имеющая доступ к закрытым данным базового класса). Функция `print` производного класса выводит также оплату служащих. Обратите внимание, как вызывается весия функции `print` базового класса:

```
Employee::print();
```

Поскольку функции базового класса и производного класса имеют одинаковые имена и сигнатуры, функции базового класса должно предшествовать имя класса и операция разрешения области действия. В противном случае вызовется опять версия функции производного класса, что приведет к бесконечной рекурсии (функция `print` класса `HourlyWorker` будет вызывать сама себя).

9.7. Открытые, защищенные и закрытые базовые классы

При порождении класса из базового класса этот базовый класс может наследоваться как `public`, `protected` или `private`. Защищенное и закрытое наследование встречаются редко и каждое из них нужно использовать с большой осторожностью; в этой книге мы используем только открытое наследование.

При порождении класса как `public` открытые элементы базового класса становятся открытыми элементами производного класса, а защищенные элементы базового класса становятся защищенным элементами производного класса. Закрытые элементы базового класса никогда не бывают доступны для производного класса.

При защищенном наследовании открытые и защищенные элементы базового класса становятся защищенными элементами производного класса. При закрытом наследовании открытые и защищенные элементы базового класса становятся закрытыми элементами производного класса. При закрытом и защищенном наследовании не справедливо отношение, что объект производного класса является объектом базового класса.

Рис. 9.6 обобщает доступ элементов базового класса из производного класса, основывающийся на спецификаторах доступа к элементам в базовом классе и типе наследования. Первый столбец содержит спецификаторы доступа к элементам в базовом классе. Первая строка содержит типы наследования. Остальная часть таблицы указывает спецификаторы доступа к элементам базового класса, которые применимы в производном классе, и краткое описание того, как можно осуществить доступ к элементам базового класса.

Спецификатор доступа к элементам в базовом классе	Тип наследования		
	public открытое наследование	protected защищенное наследование	private закрытое наследование
public	public в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам, дружественным функциям и функциям, не являющимся элементами.	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	private в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.
protected	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	private в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.
private	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.

Рис. 9.6. Сводка доступности элементов базового класса в производном классе

9.8. Прямые и косвенные базовые классы

Базовый класс может быть *прямым* или *косвенным базовым классом* производного класса. Прямой базовый класс явно перечисляется в заголовке при объявлении производного класса. Косвенный базовый класс явно не перечисляется в заголовке производного класса; он наследуется через два или более уровней иерархии классов.

9.9. Использование конструкторов и деструкторов в производных классах

Поскольку производный класс наследует элементы базового класса, то при создании объекта производного класса должен быть вызван конструктор базового класса для задания начальных значений элементам базового класса, содержащимся в объекте производного класса. В конструкторе производного класса при явном вызове конструктора базового класса может быть предусмотрен список инициализаторов элементов; в противном случае конструктор производного класса будет неявно вызывать конструктор базового класса с умолчанием.

Конструкторы и операции присваивания не наследуются производными классами. Однако, конструкторы и операции присваивания производного класса могут вызывать конструкторы и операции присваивания базового класса.

Конструктор производного класса всегда сначала вызывает конструктор своего базового класса для задания начальных значений тем элементам производного класса, которые идентичны элементам базового класса. Если конструктор производного класса отсутствует, то конструктор по умолчанию производного класса вызывает конструктор базового класса. Деструкторы вызываются в последовательности, обратной вызовам конструкторов, так что деструктор производного класса вызывается раньше соответствующего деструктора базового класса.

Замечание по технике программирования 9.3

При создании объекта производного класса первым выполняется конструктор базового класса, затем конструкторы объектов-элементов производных классов, затем конструктор производного класса. Деструкторы вызываются в последовательности, обратной той, в которой вызывались соответствующие конструкторы.

Замечание по технике программирования 9.4

Последовательность, в которой конструируются объекты-элементы, – это последовательность, в которой эти объекты объявлены в определении класса. На это не влияет последовательность, в которой перечислены инициализаторы элементов.

Замечание по технике программирования 9.5

При наследовании конструкторы базовых классов вызываются в той последовательности, в которой указано наследование в определении производного класса. На это не влияет последовательность, в которой указаны конструкторы базовых классов в описании конструктора производного класса.

Программа на рис. 9.7 демонстрирует последовательность, в которой вызываются конструкторы и деструкторы производного класса. Программа состоит из 5 частей. Части 1 и 2 показывают простой класс **Point**, содержащий конструктор, деструктор и защищенные данные-элементы **x** и **y**. Конструктор и деструктор печатают объект класса **Point**, для которого они активизированы. Части 3 и 4 показывают простой класс **Circle**, наследующий **Point** открытым наследованием, содержащий конструктор, деструктор и закрытый элемент данных **radius**. Конструктор и деструктор печатают объект класса **Circle**, для которого они активизированы. Конструктор **Circle** активизирует также конструктор класса **Point**, используя список инициализаторов элементов, и передает значения **a** и **b** для задания начальных значений элементов-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
                                                // с умолчанием
    ~Point();                                // деструктор
protected:
    float x, y;                            // x и y - координаты Point
};

#endif
```

Рис. 9.7. Определение класса **Point** (часть 1 из 5)

```
// POINT2.CPP
// Определения функций-элементов класса Point
#include <iostream.h>
#include "point2.h"

// Конструктор класса Point
Point::Point(float a, float b)
{
    x= a;
    y=b;

    cout << "Конструктор Point: "
        << '[' << x << ", " << y << ']' << endl;
}

// Деструктор класса Point
Point::~Point()
{
    cout << "Деструктор Point: "
        << '[' << x << ", " << y << ']' << endl;
}
```

Рис. 9.7. Определения функций-элементов класса **Point** (часть 2 из 5)

```
// CIRCLE2.H
// Определение класса Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"
#include <iomanip.h>

class Circle : public Point {
public:
    // конструктор с умолчанием
    Circle(float r = 0.0, float x = 0, float y = 0);

    ~Circle(); // деструктор
private:
    float radius; //радиус Circle
};

#endif
```

Рис. 9.7. Определение класса **Circle** (часть 3 из 5)

В части 5 представлена программа драйвер для иерархии **Point — Circle**. Программа начинается созданием объекта класса **Point** со своей собственной областью определения внутри **main**. Управление входит и сразу выходит из области определения этого объекта, так что вызываются и конструктор, и деструктор объекта. Затем программа создает объект **circle1** класса **Circle**. Это активизирует конструктор класса **Point**, осуществляющий вывод значений, переданных ему из конструктора класса **Circle**, затем выполняется вывод, указанный в конструкторе класса **Circle**. Следующим возникает объект **circle2** класса **Circle**. Снова вызываются конструкторы классов **Point** и **Circle**. Отметим, что тело конструктора **Point** выполняется раньше тела конструктора **Circle**. При достижении конца **main** должны быть вызваны деструкторы объектов **circle1** и **circle2**. Деструкторы вызываются в последовательности, обратной вызовам соответствующих им конструкторов. Поэтому деструктор **Point** и деструктор **Circle** вызываются сначала для объекта **circle2**, а потом для объекта **circle1**.

```
// CIRCLE2.CPP
// Определение функций-элементов класса Circle
#include "circle2.h"

// Конструктор Circle вызывает конструктор Point
Circle::Circle(float r, float a, float b)
    : Point(a, b) // вызов конструктора базового класса
{
    radius = r;

    cout << "Конструктор Circle: радиус равен "
        << radius << "[" << a << ", " << b << ']' << endl;
}

// Деструктор класса Circle
Circle::~Circle()
{
    cout << "Деструктор Circle: радиус равен: "
        << radius << "[" << x << ", " << y << ']' << endl;
}
```

Рис. 9.7. Определения функций элементов класса **Circle** (часть 4 из 5)

```
// FIG9_7.CPP
// Демонстрация вызовов конструкторов и деструкторов
// базового и производного классов.

#include <iostream.h>
#include "point2.h"
#include "circle2.h"

main()
{
// Демонстрация вызовов конструктора и деструктора 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;
}
```

```
Конструктор Point: [1.1, 2.2]
Деструктор Point: [1.1, 2.2]
```

```
Конструктор Point: [7.2, 2.9]
Конструктор Circle: радиус равен 4.5 [7.2, 2.9]
```

```
Конструктор Point: [5, 5]
Конструктор Circle: радиус равен 10 [5, 5]
```

```
Деструктор Circle: радиус равен: 10 [5, 5]
Деструктор Point: [5, 5]
Деструктор Circle: радиус равен: 4.5 [7.2, 2.9]
Деструктор Point: [7.2, 2.9]
```

Рис. 9.7. Последовательность вызовов конструкторов и деструкторов базового и производного классов (часть 5 из 5)

9.10. Неявное преобразование объектов производных классов в объекты базовых классов

Несмотря на тот факт, что объекты производных классов «являются» также и объектами базового класса, типы объектов производного и базового классов различны. Объекты производных классов можно рассматривать как объекты базового класса. Это имеет смысл, потому что производный класс имеет элементы, соответствующие каждому из элементов базового класса. Но напомним, что производный обычно имеет большее количество элементов, чем его базовый класс. Поэтому присваивание объекта базового класса объекту производного класса оставило бы неопределенными добавочные элементы производного класса. Хотя такое присваивание «естественным путем» не

разрешено, его можно было бы выполнить законным путем, предусмотрев соответствующую перегруженную операцию присваивания или конструктор преобразования.

Типичная ошибка программирования 9.4

Присваивание объекта производного класса объекту соответствующего базового класса и затем попытка сослаться в этом новом объекте базового класса на элементы, имеющиеся только в объектах производного класса.

Указатель на объект производного класса может быть неявно преобразован в указатель на объект базового класса, потому что объект производного класса «является» объектом базового класса.

Существует четыре возможных способа комбинирования и попарного сопоставления указателей и объектов базового и производного классов:

1. Ссылка на объект базового класса с помощью указателя базового класса очевидна.
2. Ссылка на объект производного класса с помощью указателя производного класса очевидна.
3. Ссылка на объект производного класса с помощью указателя базового класса не опасна, потому что объект производного класса является также и объектом своего базового класса. Если производится ссылка с помощью указателя базового класса на элементы, имеющиеся только в объектах производного класса, компилятор сообщит о синтаксической ошибке.
4. Ссылка на объект базового класса с помощью указателя производного класса является синтаксической ошибкой. Указатель производного класса сначала должен быть приведен к типу указателя базового класса.

Типичная ошибка программирования 9.5

Приведение типа указателя базового класса к типу указателя производного класса может вызвать ошибки, если этот указатель используется затем для ссылки на объект базового класса, который не имеет требуемых элементов производного класса.

Удобно было бы рассматривать объекты производного класса как объекты базового класса и оперировать со всеми этими объектами с помощью указателей базового класса, однако здесь имеется проблема. В бухгалтерских системах, например, нам хотелось бы иметь возможность просматривать связный список служащих и вычислять для каждого из них его недельную зарплату. Но использование указателей базового класса позволяет программе вызывать для обработки платежных ведомостей только процедуру базового класса (если такая процедура в базовом классе существует). Нам нужен был бы способ активизировать процедуру обработки платежных ведомостей для любого объекта, будь он производного класса или базового класса, и делать это просто использованием указателей базового класса. Решением является использование виртуальных функций и полиморфизма; как это делать описано в главе 10.

9.11. Проектирование программного обеспечения с помощью наследования

Мы можем использовать наследование для настройки существующего программного обеспечения. Мы наследуем атрибуты и функции существующего класса, а затем добавляем атрибуты и функции, необходимые для настройки класса под наши нужды. Это делается в C++ без доступа производного класса к исходному тексту базового класса, но производный класс должен иметь возможность компоноваться с объектным кодом базового класса. Такая возможность, заложенная в C++, привлекательна для независимых продавцов программного обеспечения (НППО). НППО могут разрабатывать собственные классы для продажи или лицензирования и делать эти классы доступными пользователям в форме объектного кода. Пользователи затем могут быстро создавать новые классы из этих библиотечных классов без доступа к собственным исходным кодам НППО. Все, что требуется от НППО, — это снабжать объектные коды заголовочными файлами.

Студентам может быть трудно проникнуться проблемами, с которыми сталкиваются разработчики и создатели крупных программных проектов. Опытные разработчики таких систем будут неизменно заявлять, что ключом к улучшению процесса разработки программного обеспечения является повторное использование программных кодов. Объектно-ориентированное программирование вообще, и на C++ в частности, несомненно обеспечивают это.

Доступность важных и полезных библиотек классов обеспечивает максимальные преимущества повторного использования кодов посредством наследования. Интерес к библиотекам классов будет расти так же, как интерес к C++. Создание и продажа библиотек классов становится такой же быстроразвивающейся индустрией, как и производимое независимыми продавцами компактное программное обеспечение в эпоху появления первых персональных компьютеров. Разработчики приложений будут строить свои приложения с помощью этих библиотек, а разработчики библиотек будут вознаграждены тем, что их библиотеки широко используются в приложениях. Библиотеки, непрерывно пополняемые с помощью компиляторов C++, имеют тенденцию становиться направленными на определенные сферы применения. Вот почему мы видим повсеместное стремление создавать библиотеки классов для огромного разнообразия конкретных применений.

Замечание по технике программирования 9.6

Создание производного класса не влияет на исходный или объектный код базового класса; сохранность базового класса оберегается наследованием.

Базовый класс описывает общие черты объектов. Все классы, порожденные базовым классом, наследуют возможности этого базового класса. В процессе объектно-ориентированного проектирования разработчик отыскивает общие черты объектов и выражает их в форме базового класса. Производные классы затем пополняются дополнительными по сравнению с унаследованными от базового класса возможностями.

Точно так же, как разработчик не объектно-ориентированных систем старается избежать не вызванного необходимостью быстрого роста числа функций, разработчик объектно-ориентированных систем должен избегать не вызванного необходимостью быстрого роста числа классов. Такое разрастание классов создает проблемы управления и может помешать повторному использованию кодов просто потому, что потенциальному повторному пользователю слишком трудно выделить нужный ему класс из огромного набора имеющихся. Выходом из положения является создание меньшего числа классов, но таких, каждый из которых обеспечен широкими функциональными возможностями. Такие классы могли бы быть несколько более дорогими для пользователей; они могут скрыть ненужные им избыточные функциональные возможности, приспособливая таким образом классы для удовлетворения своих потребностей.

Совет по повышению эффективности 9.1

Если классы, полученные путем наследования, более широкие, чем необходимо, это ведет к потере ресурсов памяти и производительности. Наследуйте из класса только самое необходимое.

Заметим, что чтение объявлений производных классов может приводить к недоразумениям, поскольку в них не показаны унаследованные элементы, которые тем не менее присутствуют в производных классах. Аналогичные проблемы могут существовать и с документацией на производные классы.

Замечание по технике программирования 9.7

В объектно-ориентированных системах классы часто тесно связаны. «Факторизуйте» общие атрибуты и функции и помещайте их в базовом классе. Затем используйте наследование для формирования производных классов.

Замечание по технике программирования 9.8

Производный класс содержит атрибуты и функции своего базового класса. Производный класс может также содержать дополнительные атрибуты и функции. При наследовании базовый класс может быть скомпилирован независимо от производного класса. При формировании производного класса нужно компилировать только дополнительные атрибуты и функции; их объединение с базовым классом сформирует производный класс.

Замечание по технике программирования 9.9

Изменения в базовом классе не требуют изменений в производных классах до тех пор, пока открытый интерфейс базового класса остается неизменным. Однако, производные классы могут нуждаться при этом в перекомпиляции.

9.12. Композиция и наследование

Мы обсудили отношения «является», которое поддерживается наследованием. Мы обсудили также отношения «содержит» (и видели примеры в предыдущих главах), при которых класс может содержать другие классы в

качестве элементов; такие отношения создают новые классы композицией существующих классов. Например, для классов `Employee` (служащий), `BirthDay` (день рождения) и `TelephoneNumber` (номер телефона) было бы неуместно говорить, что `Employee` «является» `BirthDay` или что `Employee` «является» `TelephoneNumber`.

Замечание по технике программирования 9.10

Программные модификации класса, который является элементом другого класса, не требуют изменения вмещающего класса до тех пор, пока остается неизменным открытый интерфейс класса-элемента. Отметим, однако, что класс композиции может нуждаться в перекомпиляции.

9.13. Отношения «использует А» и «знает А»

И наследование, и композиция способствуют повторному использованию кодов путем создания новых классов, которые имеют много общего с существующими классами. Имеются и другие способы использования услуг классов. Хотя объект человек не является автомобилем и не содержит автомобиль, но, конечно, объект человек *использует* автомобиль. Отношение «использует» объект осуществляется просто путем обращения к функции-элементу этого объекта.

Функция может «знать» другой объект. Базы знаний часто используют такие отношения. Для того, чтобы один объект «знал» о другом достаточно, чтобы объект содержал указатель или ссылку на этот другой объект. В этом случае говорят, что один объект находится в отношении «знает» с другим объектом.

9.14. Учебный пример: точка, круг, цилиндр

Рассмотрим теперь основное упражнение этой главы. Мы рассмотрим иерархию точка, круг, цилиндр. Сначала мы разработаем и используем класс `Point` (рис. 9.8). Затем мы представим пример, в котором класс `Circle` является производным классом от класса `Point` (рис. 9.9). В заключение, мы представим пример, в котором класс `Cylinder` является наследником класса `Circle` (рис. 9.10). Класс `Point` представлен на рис. 9.8. Часть 1 — это определение класса `Point`. Отметим, что элементы класса `Point` — защищенные. Таким образом, когда класс `Circle` наследует классу `Point`, функции-элементы класса `Circle` могут непосредственно ссылаться на координаты `x` и `y`, что предпочтительнее, чем использование функций доступа. Это дает лучшую производительность.

На рис. 9.8 часть 2 представлены определения функций-элементов класса `Point`, а на рис. 9.8 часть 3 представлена программа драйвер класса `Point`. Заметим, что `main` должна использовать функции доступа `getX` и `getY`, чтобы читать значения защищенных данных-элементов `x` и `y`; напомним, что защищенные данные-элементы доступны только элементам и друзьям их класса и производного класса.

```

// 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 с умолчанием
    void setPoint(float, float);          // установка координат
    float getX() const { return x; }      // получение координаты x
    float getY() const { return y; }      // получение координаты y
protected:
    float x, y;                         // доступно производным классам
};

#endif

```

Рис. 9.8. Определение класса **Point** (часть 1 из 3)

```

// POINT2.CPP
// Функции-элементы класса Point
#include <iostream.h>
#include "point2.h"

// конструктор класса Point
Point::Point(float a, float b) { setPoint(a, b); }

// установка координат x и y
void Point::setPoint(float a, float b)
{
    x = a;
    y = b;
}

// вывод Point
ostream &operator<< (ostream &output, const Point &p)
{
    output << '[' <<p.x <<", " <<p.y <<']';
    return output;                      // возможность сцепления
}

```

Рис. 9.8. Функции элементы класса **Point** (часть 2 из 3)

Наш следующий пример показан на рис. 9.9, части с 1 по 3. Здесь повторно использованы определение класса **Point** и определения функций-элементов из рис. 9.8. В частях с 1 по 3 показаны описание класса **Circle**, определения функций-элементов класса **Circle** и его программа драйвер. Заметим, что класс **Circle** наследуется от класса **Point** открытым наследованием. Это означает, что открытый интерфейс **Circle** включает функции-элементы **Point** и дополнительные функции-элементы **setRadius**, **getRadius** и **area**. Заметим, что перегруженная операция поместить в поток класса **Circle** способна ссылаться на переменные **x** и **y** непосредственно, потому что это — защищенные элементы базового класса **Point**. Заметим также, что необходимо ссылаться на переменные **x** и **y** через объект: **c.x** и **c.y**. Это вызвано тем, что перегруженная операция поместить в поток не является функцией-элементом класса **Circle**.

```
// FIG9_8.CPP
// Драйвер класса Point
#include <iostream.h>
#include "point2.h"

main()
{
    Point p(7.2, 11.5);           // создание объекта p класса Point

    // защищенные данные Point недоступны main
    cout << "Координата X равна " << p.getX()
        << endl << "Координата Y равна " << p.getY();

    p.setPoint(10, 10);
    cout << endl << endl << "Новая точка p равна "
        << p << endl;

    return 0;
}
```

```
Координата X равна 7.2
Координата Y равна 11.5
```

```
Новая точка p равна [10, 10]
```

Рис. 9.8. Драйвер класса **Point** (часть 3 из 3)

Программа драйвер создает объект класса **Circle**, затем использует функции доступа, чтобы получить информацию об объекте класса **Circle**. Функция **main** не является ни функцией-элементом, ни другом класса **Circle**, так что она не может непосредственно ссылаться на защищенные данные класса **Circle**. Затем программа драйвер использует функции установки **setRadius** и **setPoint** для переустановки радиуса и координат центра круга. В заключение драйвер инициализирует переменную ссылку **pRef** типа «ссылка на объект класса **Point**» (**Point &**), указывая на объект с класса **Circle**. Затем драйвер печатает **pRef**, который, несмотря на тот факт, что он инициализирован объектом класса **Circle**, «думает», что это объект **Point**, так что на самом деле объект класса **Circle** печатается как объект класса **Point**.

Наш последний пример показан на рис. 9.10, части с 1 по 3. Здесь повторно использованы определения классов **Point** и **Circle** и определения их функций-элементов на рис. 9.8 и 9.9. В частях с 1 по 3 показаны определение класса **Cylinder**, определения функций-элементов **Cylinder** и программа драйвер соответственно. Заметим, что класс **Cylinder** наследуется от класса **Circle** открытым наследованием. Это означает, что открытый интерфейс класса **Cylinder** включает функции-элементы класса **Circle** а также функции-элементы **setHeight**, **getHeight**, **area** (переопределенную) и **volume**. Заметим, что перегруженная операция поместить в поток класса **Cylinder** способна ссылаться на переменные **x**, **y** и **radius** непосредственно, так как это — защищенные элементы базового класса **Circle** (**x** и **y** были унаследованы **Circle** от **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 с умолчанием
    Circle(float r = 0.0, float x = 0, float y = 0);
    void setRadius(float);           // установка радиуса
    float getRadius() const;        // возвращение радиуса
    float area() const;             // вычисление площади
protected:
    float radius;                  // доступно производным классам
};                                // радиус круга

#endif

```

Рис. 9.9. Определение класса **Circle** (часть 1 из 3)

```

// CIRCLE2.CPP
// Определения функций-элементов класса Circle

#include <iostream.h>
#include <iomanip.h>
#include "circle2.h"

// Конструктор Circle вызывает конструктор Point с помощью
// инициализаторов элементов и задает начальное значение радиуса
Circle::Circle(float r, float a, float b)
    : Point (a, b)          // вызов конструктора базового класса
{ radius = r; }

// Установка радиуса
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 формате:
// Центр = [x, y]; Радиус = #.##
ostream &operator<< (ostream &output, const Circle &c)
{
    output << "Центр = [" << c.x << ", " << c.y
        << "]; Радиус = " << setiosflags(ios::showpoint)
        << setprecision(2) << c.radius;

return output;                      // возможность сцепленных вызовов
}

```

Рис. 9.9. Определения функций элементов класса **Circle** (часть 2 из 3)

```
// FIG9_9.CPP
// Драйвер класса Circle
#include <iostream.h>
#include <iomanip.h>
#include "point2.h"
#include "circle2.h"

main()
{
    Circle c(2.5, 3.7, 4.3);
    cout << "Координата X равна " << c.getX()
        << endl << "Координата Y равна " << c.getY()
        << endl << "Радиус равен " << c.getRadius();

    c.setRadius(4.25);
    c.setPoint(2, 2);
    cout << setiosflags(ios::fixed | ios::showpoint);
    cout << endl << endl << "Новые координаты точки и радиус равны"
        << endl << c << endl << "Площадь: " << c.area() << endl;

    Point &pRef = c;
    cout << endl << "Печать Circle как Point: "
        << pRef << endl;

    return 0;
}
```

```
Координата X равна 3.7
Координата Y равна 4.3
Радиус равен 2.5
```

```
Новые координаты точки и радиус равны
Центр = [2.000000, 2.000000]; Радиус = .25
Площадь: 56.74
```

```
Печать Circle как Point: [2.00, 2.00]
```

Рис. 9.9. Драйвер класса **Circle** (часть 3 из 3)

Заметим также, что на переменные **x**, **y** и **radius** необходимо ссылаться через объект: **c.x**, **c.y** и **c.radius**. Это вызвано тем, что перегруженная операция поместить в поток не является функцией-элементом класса **Cylinder**. Программа драйвер создает объект класса **Cylinder**, затем использует функции доступа для получения информации об объекте **Cylinder**. Как и раньше, **main** не является ни функцией-элементом, ни другом класса **Cylinder**, так что она не может непосредственно ссылаться на защищенные данные класса **Cylinder**. Затем программа драйвер использует функции установки **setHeight**, **setRadius** и **setPoint** для переустановки высоты, радиуса и координат цилиндра. В заключение драйвер задает начальное значение переменной ссылке **pRef** типа «ссылка на объект **Point**» (**Point &**), указывающее на объект **cyl** класса **Cylinder**. Драйвер печатает **pRef**, который, несмотря на то, что он инициализирован объектом класса **Cylinder**, «думает», что это объект **Point**, так что на самом деле объект класса **Cylinder** печатается как объект класса **Point**. Затем драйвер задает начальное значение переменной ссылке **cRef** типа

«ссылка на объект Circle» (`Circle &`), указывающее на объект `cyl1` класса `Cylinder`. Драйвер печатает `cRef`, который, несмотря на то, что он инициализирован объектом класса `Cylinder`, «думает», что это объект класса `Circle`, так что объект класса `Cylinder` печатается как объект класса `Circle`. Выводится также площадь объекта класса `Circle`.

Этот пример наглядно демонстрирует открытое наследование, определения и ссылки на защищенные данные-элементы. Теперь читатель должен чувствовать себя уверенно в вопросах наследования. В следующей главе мы покажем, как в общем случае программировать с помощью иерархического наследования, используя полиморфизм. Абстрагирование данных, наследование и полиморфизм — это суть объектно-ориентированного программирования.

```
// CYLINDR2.H
// Определение класса Cylinder
#ifndef CYLINDR2_H
#define CYLINDR2_H

#include "circle2.h"

class Cylinder : public Circle{
    friend ostream& operator<<(ostream&, const Cylinder&);

public:
    // конструктор с умолчанием
    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;   // возвращение высоты
    float area() const;        // вычисление и возвращение площади
    float volume() const;      // вычисление и возвращение объема

protected:
    float height;             // высота цилиндра
};

#endif
```

Рис. 9.10. Определение класса `Cylinder` (часть 1 из 3)

9.15. Множественное наследование

До сих пор в этой главе мы обсуждали простое наследование, когда каждый класс порождается только от одного базового класса. Однако, класс может порождаться более, чем от одного базового класса; такое порождение называется *множественным наследованием*. Множественное наследование означает, что производный класс наследует элементы нескольких базовых классов. Эта мощная возможность способствует интересным формам повторного использования кодов, но может вызвать и многообразные проблемы, связанные с некоторой неопределенностью.

```
// CYLINDR2.CPP
// Определения элементов и дружественных функций 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 конструктора базового класса
{ 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 его 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 Cylinder& c)
{
    output << "Центр = [ " << c.x << ", " << c.y
        << "]; Радиус = " << setiosflags(ios::showpoint)
        << setprecision(2) << c.radius
        << "; Высота = " << c.height;

    return output;                // возможность сцепленных вызовов
}
```

Рис. 9.10. Определения элементов и дружественных функций класса **Cylinder** (часть 2 из 3)

Хороший стиль программирования 9.1

Множественное наследование является мощной возможностью при правильном использовании. Множественное наследование должно использоваться, когда между новым типом и двумя или более существующими типами имеется отношение «является» (т.е. тип А «является» типом В и «является» типом С).

```

// FIG9_10.CPP
// Драйвер класса Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "point2.h"
#include "circle2.h"
#include "cylindr2.h"

main()
{
    // создание объекта класса Cylinder
    Cylinder cyl(5.7, 2.5, 1.2, 2.3);

    // использование функций get для отображения цилиндра
    cout << "Координата X равна " << cyl.getX() << endl
        << "Координата Y равна " << cyl.getY() << endl
        << "Радиус равен " << cyl.getRadius() << endl
        << "Высота равна " << cyl.getHeight() << endl << endl;

    // использование функций set для изменения атрибутов цилиндра
    cyl.setHeight(10);
    cyl.setRadius(4.25);
    cyl.setPoint(2, 2);
    cout << "Новые координаты, радиус "
        << "и высота цилиндра:" << endl << cyl << endl;

    // отображение цилиндра как Point
    Point &pRef = cyl;           // pRef "думает", что это Point
    cout << endl << "Cylinder, напечатанный как Point:"
        << pRef << endl << endl;

    // отображение цилиндра как Circle
    Circle &cRef = cyl;          // cRef "думает", что это Circle
    cout << "Cylinder, напечатанный как Circle:" << endl
        << cRef << endl << "Площадь: " << cRef.area() << endl;
    return 0;
}

```

Координата X равна 1.2

Координата Y равна 2.3

Радиус равен 2.5

Высота равна 5.7

Новые координаты, радиус и высота цилиндра:

Центр = [2, 2]; Радиус = 4.25; Высота = 10.00

Cylinder, напечатанный как Point:[2.00, 2.00]

Cylinder, напечатанный как Circle:

Центр = [2.00, 2.00]; Радиус = 4.25

Площадь: 56.74

Рис. 9.10. Драйвер класса **Cylinder** (часть 3 из 3)

Рассмотрим пример множественного наследования на рис. 9.11. Класс **Base1** содержит один защищенный элемент данных — **int value**. **Base1** содержит конструктор, который устанавливает **value**, и открытую функцию-элемент **getData**, которая возвращает **value**.

```
// BASE1.H
// Определение класса Basel
#ifndef BASE1_H
#define BASE1_H

class Basel {
public:
    Basel(int x){ value = x; }
    int getData() const { return value; }
protected:
    int value;           // доступно производным классам
};                      // наследуется производными классами

#endif
```

Рис. 9.11. Определение класса **Base1** (часть 1 из 6)

```
// 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:
    char letter;          // доступно производным классам
};                      // наследуется производными классами

#endif
```

Рис. 9.11. Определение класса **Base2** (часть 2 из 6)

```
// DERIVED.H
// Определение класса Derived, с множественным наследованием
// базовых классов (Base1 и Base2).

#ifndef DERIVED_H
#define DERIVED_H

#include "base1.h"
#include "base2.h"

// множественное наследование
class Derived : public Basel, 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. 9.11. Определение класса **Derived** (часть 3 из 6)

Класс `Base2` аналогичен классу `Base1`, за исключением того, что его защищенным данным является `char letter`. `Base2` имеет также открытую функцию-элемент `getData`, но эта функция возвращает значение `char letter`.

Класс `Derived` порождается двумя классами — `Base1` и `Base2` посредством множественного наследования. Класс `Derived` имеет закрытый элемент данных `float real` и открытую функцию-элемент `getReal`, которая читает значение `float real`.

```
// DERIVED.CPP
// Определения функций-элементов класса Derived
#include <iostream.h>
#include "derived.h"

// Конструктор Derived вызывает конструкторы класса Base1
// и класса Base2.
Derived::Derived(int i, char c, float f)
    : Base1(i), Base2(c)      // вызов конструкторов обоих базовых
                               // классов
{ real = f; }

// Возвращение значения real
float Derived::getReal() const { return real; }

// Отображение всех элементов данных Derived
ostream &operator<<(ostream &output, const Derived &d)
{
    output << "        Целое: " << d.value << endl
        << "        Символ: " << d.letter << endl
        << "Действительное: " << d.real;

    return output;           // возможность сцепленных вызовов
}
```

Рис. 9.11. Определения функций элементов класса `Derived` (часть 4 из 6)

Заметьте, что множественное наследование указывается двоеточием (`:`) после имени класса (`class Derived`) с последующим перечислением через запятую базовых классов. Отметим также, что конструктор `Derived` вызывает конструкторы каждого из своих базовых классов `Base1` и `Base2` с использованием списка инициализаторов элементов. Конструкторы базового класса вызываются в той последовательности, в которой определено наследование, но не в той, в которой эти конструкторы упоминаются.

Перегруженная операция поместить в поток класса `Derived` использует для печати `value`, `letter` и `real` запись с точкой производного объекта `d`. Эта функция-операция является другом `Derived`, так что `operator<<` может непосредственно иметь доступ к закрытому элементу данных `real` класса `Derived`. Поскольку эта операция — друг производного класса, она может иметь доступ также и к защищенным элементам `value` и `letter` классов `Base1` и `Base2` соответственно.

```
// FIG9_11.CPP
// Драйвер примера множественного наследования

#include <iostream.h>
#include "base1.h"
#include "base2.h"
#include "derived.h"

main()
{
    Base1 b1(10), *base1Ptr;      //создание объекта базового класса
    Base2 b2('Z'), *base2Ptr;    //создание объекта другого
                                 // базового класса

    Derived d(7, 'A', 3.5);     //создание объекта производного
                                 // класса
    // печать данных-элементов объектов базового класса
    cout << " Объект b1 содержит целое "
        << b1.getData() << endl
        << "Объект b2 содержит символ "
        << b2.getData() << endl
        << "Объект d содержит:" << endl << d << endl << endl;

    // печать данных-элементов объекта производного класса
    // операция разрешения области действия устраниет
    // неопределенность getData
    cout << "Элементы данных класса Derived могут быть "
        << "доступны индивидуально:" << endl
        << "           Целое: " << d.Base1::getData() << endl
        << "           Символ: " << d.Base2::getData() << endl
        << "Действительное: " << d.getReal() << endl << endl;

    cout << "Derived может трактоваться как объект "
        << "обоих базовых классов:" << endl;

    // трактовка Derived как объекта класса Base1
    base1Ptr = &d;
    cout << "Результат base1Ptr->getData() "
        << base1Ptr->getData() << endl;

    // трактовка Derived как объекта класса Base2
    base2Ptr = &d;
    cout << "Результат base2Ptr->getData() "
        << base2Ptr->getData() << endl;

    return 0;
}
```

Рис. 9.11. Драйвер примера множественного наследования (часть 5 из 6)

```
Объект b1 содержит целое 10
Объект b2 содержит символ Z
Объект d содержит:
    Целое: 7
    Символ: A
Действительное: 3.5
```

```
Элементы данных класса Derived могут быть доступны индивидуально:
    Целое: 7
    Символ: A
Действительное: 3.5
```

```
Derived может трактоваться как объект обоих базовых классов:
Результат base1Ptr->getData() 7
Результат base2Ptr->getData() A
```

Рис. 9.11. Пример множественного наследования (часть 6 из 6)

Теперь давайте рассмотрим программу драйвер в `main`. Мы создаем объект `b1` класса `Base1` и задаем целому начальное значение 10. Затем мы создаем объект `b2` класса `Base2` и задаем символу начальное значение 'Z'. Наконец, мы создаем объект `d` класса `Derived` и задаем начальные значения 7 для целого, 'A' для символа и 3.5 для действительного элементов.

Содержание каждого из объектов базового класса печатается путем вызовов функции-элемента `getData` для каждого объекта. Несмотря на существование двух функций `getData`, вызовы не являются неопределенными, потому что они ссылаются непосредственно на версию функции `getData` объекта `b1` и на версию функции `getData` объекта `b2`.

Далее мы печатаем содержимое объекта `d` класса `Derived` с помощью статического связывания. Но здесь мы имеем проблему неопределенности, потому что один объект содержит две функции `getData`, одну, унаследованную от `Base1`, и одну, унаследованную от `Base2`. Эта проблема легко решается с помощью бинарной операции разрешения области действия: `d.Base1::getData()`, чтобы напечатать целое `value`, и `d.Base2::getData()`, чтобы напечатать символ `letter`. Значение `float` в `real` печатается без неопределенности с помощью вызова `d.getReal()`.

Далее мы показываем, что отношения «является» простого наследования применимы и к множественному наследованию. Мы присваиваем адрес производного объекта `b` базового класса указателю `base1Ptr` и печатаем целое `value`, активизируя функцию-элемент `getData` класса `Base1` через `base1Ptr`. Аналогично мы присваиваем адрес производного объекта `d` базового класса указателю `base2Ptr` и печатаем символ `letter`, активизируя функцию элемент `getData` класса `Base1` через `base2Ptr`.

Этот пример показывает механизм множественного наследования и знакомит с простейшей проблемой неопределенности. Множественное наследование — сложная тема, рассматриваемая более подробно в более серьезных руководствах по C++.

Хороший стиль программирования 9.2

Множественное наследование — мощное свойство, но оно может внести в систему сложности. Требуется большая осторожность при проектировании систем для извлечения выгоды из использования множественного наследования; использовать его не следует, если можно ограничиться простым наследованием.

Резюме

- Одним из ключей к могуществу объектно-ориентированного программирования является достижение повторного использования кодов путем наследования.
- Программист может указать, что новый класс наследует данные-элементы и функции-элементы ранее определенного базового класса. В этом случае новый класс называется производным классом.
- При простом наследовании класс порождается только одним базовым классом. При множественном наследовании производный класс наследует нескольким базовым классам (возможно, неродственным).
- Производный класс обычно добавляет свои собственные данные-элементы и функции-элементы, так что производный класс в общем случае больше своего базового класса. Производный класс более специфичен по своему назначению, более узок, чем его базовый класс, и представляет меньшую группу объектов.
- Производный класс не может иметь доступ к закрытым элементам своего базового класса; разрешение доступа нарушило бы инкапсуляцию базового класса. Однако, производный класс может иметь доступ к открытым и защищенным элементам своего базового класса.
- Конструктор производного класса всегда прежде всего вызывает конструктор своего базового класса для создания элементов базового класса, имеющихся в производном, и задания им начальных значений.
- Деструкторы вызываются в последовательности, обратной вызовам конструкторов, так что деструктор производного класса вызывается раньше деструктора базового класса.
- Наследование создает возможность повторного использования кодов, что экономит время разработки и способствует использованию проверенного и отлаженного высококачественного программного обеспечения.
- Наследование может осуществляться путем использования существующих библиотек классов.
- Иногда значительная часть программное обеспечение может быть сконструирована из стандартизованных, повторно используемых компонентов точно так же, как сегодня конструируется многое из аппаратного обеспечения.
- Разработчик производного класса не нуждается в доступе к исходному коду базового класса, но нуждается в доступе к интерфейсу и объектному коду базового класса.
- Объект производного класса может рассматриваться как объект соответствующего ему базового класса. Однако, обратное неверно.
- Базовый класс существует в иерархическом взаимоотношении с классом, порожденным от него простым наследованием.
- Класс может существовать сам по себе. Если этот класс используется в механизме наследования, то он становится либо базовым классом, который снабжает атрибутами и функциями другие классы, либо производным классом, который наследует эти атрибуты и функции.

- Иерархия наследования может быть произвольно глубокой в пределах физических ограничений конкретной системы.
- Иерархии являются полезным инструментом для понимания и управления сложностью. В связи с ростом сложности программного обеспечения C++ обеспечивает механизмы поддержки иерархических структур посредством наследования и полиморфизма.
- Для преобразования типа указателя базового класса в тип производного класса можно использовать явное приведение типов. Такой указатель не должен разыменовываться, если он действительно не указывает на объект типа производного класса.
- Защищенный уровень доступа служит промежуточным уровнем защиты между открытым доступом и закрытым доступом. Защищенные элементы базового класса могут быть доступны только элементам и друзьям базового класса и элементам и друзьям производного класса; никакие другие функции не могут иметь доступа к защищенным элементам базового класса.
- Защищенные элементы используются для расширения привилегий производных классов; этих привилегий лишены функции, не относящиеся к классу или не являющиеся дружественными ему.
- Множественное наследование указывается двоеточием (:) после имени производного класса и перечислением списка разделенных запятыми базовых классов после двоеточия. Для вызова конструкторов базового класса в конструкторе производного класса используется список инициализаторов элементов.
- При порождении класса базовый класс может быть объявлен как **public** (открытое наследование), **protected** (защищенное наследование) или **private** (закрытое наследование).
- При порождении класса как **public** открытые элементы базового класса становятся открытыми элементами производного класса, а защищенные элементы базового класса становятся защищенными элементами производного класса.
- При защищенном наследовании открытые и защищенные элементы базового класса становятся защищенными элементами производного класса.
- При закрытом наследовании открытые и защищенные элементы базового класса становятся закрытыми элементами производного класса.
- Базовый класс может быть прямым или косвенным базовым классом производного класса. Прямой базовый класс явно перечисляется в заголовке при объявлении производного класса. Косвенный базовый класс явно в заголовке производного класса не перечисляется; он наследуется через два или более уровней иерархии классов.
- Если элемент базового класса не подходит для производного класса, мы можем просто переопределить этот элемент в производном классе.
- Важно различать отношения «является» и «содержит». При отношении «содержит» объект одного класса содержит объекты другого класса как элементы. При отношении «является» объект типа производного

класса может также рассматриваться как объект базового класса типа. «Является» — это наследование. «Содержит» — это композиция.

- Объект производного класса можно присваивать объекту базового класса. Этот вид присваивания имеет смысл, потому что производный класс имеет элементы, соответствующие каждому из элементов базового класса.
- Указатель на объект производного класса может быть неявно преобразован в указатель на объект базового класса.
- Указатель базового класса можно преобразовать в указатель производного класса, используя явное приведение типов. Он должен указывать на объект производного класса.
- Базовый класс описывает общие черты объектов. Все классы, порожденные базовым классом, наследуют возможности этого базового класса. В процессе объектно-ориентированного проектирования разработчик отыскивает общие черты объектов и выражает их в форме базового класса. Производные классы затем пополняются дополнительными по сравнению с унаследованными от базового класса возможностями.
- Чтение объявлений производных классов может приводить к недоразумениям, поскольку в них не показаны унаследованные элементы, которые тем не менее присутствуют в производных классах.
- Отношение «содержит» дает примеры создания новых объектов путем объединения существующих классов
- Отношения «знает» — это объекты, содержащих указатели или ссылки на другие объекты, так что они могут связаться с этими объектами.
- Конструкторы объектов-элементов вызываются в той последовательности, в которой объявлены объекты. При наследовании конструкторы базового класса вызываются в той последовательности, в которой указано наследование, и до вызова конструктора объекта производного класса.
- Для объекта производного класса конструктор базового класса вызывается в первую очередь, а затем вызывается конструктор производного класса (который может вызвать конструкторы своих объектов-элементов).
- При уничтожении объекта производного класса деструкторы вызываются в последовательности, обратной вызовам конструкторов: сначала вызывается деструктор производного класса, а затем базового класса.
- Класс может быть порожден более, чем от одного базового класса; такое порождение называется множественным наследованием.
- Множественное наследование указывается двоеточием (:) с последующим списком базовых классов, разделенных запятыми.
- Конструктор производного класса вызывает конструкторы всех своих базовых классов, используя списки инициализаторов элементов. Конструкторы базового класса вызываются в той последовательности, в которой эти базовые классы объявляются в процессе наследования.

Терминология

абстрагирование	объект-элемент
базовый класс	объектно-ориентированное программирование
библиотеки классов	открытое наследование
деструктор базового класса	открытый базовый класс
деструктор производного класса	отношение «знает»
друзья базового класса	отношение «использует»
друзья производных классов	отношение «содержит»
закрытое наследование	отношение «является»
закрытый базовый класс	ошибка бесконечной рекурсии
защищенное порождение	переопределенный элемент
защищенный базовый класс	базового класса
защищенный элемент класса	повторное использование кода
иерархическое соотношение	подкласс
иерархия классов	производный класс
инициализатор базового класса	простое наследование
класс-элемент	прямое наследование базовый класс
клиент класса	прямой ациклический граф
ключевое слово protected	стандартизованные компоненты
композиция	программного обеспечения
конструктор базового класса	суперкласс
конструктор по умолчанию базового класса	указатель на объект базового класса
конструктор производного класса	указатель на объект производного класса
косвенный базовый класс	указатель базового класса
множественное наследование	указатель производного класса
наследование	управление доступом к элементу
настройка программного обеспечения	
неопределенность в множественном наследовании	

Типичные ошибки программирования

- 9.1. Рассмотрение объектов базового класса как объектов производного класса может вызвать ошибки.
- 9.2. Явное приведение типа указателя базового класса, который указывает на объект базового класса, к типу указателя производного класса и затем ссылка на элементы производного класса, которые не существуют в этом объекте.
- 9.3. При переопределении в производном классе функции-элемента базового класса принято вызывать версию базового класса и после этого выполнять некоторые дополнительные операции. При этом ссылка на функцию-элемент базового класса без использования операции разрешения области действия может вызвать бесконечную рекурсию, потому что функция-элемент производного класса будет в действительности вызывать сама себя.
- 9.4. Присваивание объекта производного класса объекту соответствующего базового класса и затем попытка сослаться в этом новом объекте базового класса на элементы, имеющиеся только в объектах производного класса.

- 9.5. Приведение типа указателя базового класса к типу указателя производного класса может вызвать ошибки, если этот указатель используется затем для ссылки на объект базового класса, который не имеет требуемых элементов производного класса.

Хороший стиль программирования

- 9.1. Множественное наследование является мощной возможностью при правильном использовании. Множественное наследование должно использоваться, когда между новым типом и двумя или более существующими типами имеется отношение «является» (т.е. тип А «является» типом В и «является» типом С).
- 9.2. Множественное наследование — мощное свойство, но оно может внести в систему сложности. Требуется большая осторожность при проектировании систем для извлечения выгоды из использования множественного наследования; использовать его не следует, если можно ограничиться простым наследованием.

Советы по повышению эффективности

- 9.1. Если классы, полученные путем наследования, более широкие, чем необходимо, это ведет к потере ресурсов памяти и производительности. Наследуйте из класса только самое необходимое.

Замечания по технике программирования

- 9.1. Производный класс не имеет прямого доступа к закрытым элементам своего базового класса.
- 9.2. При переопределении в производном классе функции-элемента базового класса нет необходимости получать такую же сигнатуру, как у функции-элемента базового класса.
- 9.3. При создании объекта производного класса первым выполняется конструктор базового класса, затем конструкторы объектов-элементов производных классов, затем конструктор производного класса. Деструкторы вызываются в последовательности, обратной той, в которой вызывались соответствующие конструкторы.
- 9.4. Последовательность, в которой конструируются объекты-элементы, — это последовательность, в которой эти объекты объявлены в определении класса. На это не влияет последовательность, в которой перечислены инициализаторы элементов.
- 9.5. При наследовании конструкторы базовых классов вызываются в той последовательности, в которой указано наследование в определении производного класса. На это не влияет последовательность, в которой указаны конструкторы базовых классов в описании конструктора производного класса.
- 9.6. Создание производного класса не влияет на исходный или объектный код базового класса; сохранность базового класса оберегается наследованием.

- 9.7. В объектно-ориентированных системах классы часто тесно связаны. «Факторизуйте» общие атрибуты и функции и помещайте их в базовом классе. Затем используйте наследование для формирования производных классов.
- 9.8. Производный класс содержит атрибуты и функции своего базового класса. Производный класс может также содержать дополнительные атрибуты и функции. При наследовании базовый класс может быть скомпилирован независимо от производного класса. При формировании производного класса нужно компилировать только дополнительные атрибуты и функции; их объединение с базовым классом сформирует производный класс.
- 9.9. Изменения в базовом классе не требуют изменений в производных классах до тех пор, пока открытый интерфейс базового класса остается неизменным. Однако, производные классы могут нуждаться при этом в перекомпиляции.
- 9.10. Программные модификации класса, который является элементом другого класса, не требуют изменения вмещающего класса до тех пор, пока остается неизменным открытый интерфейс класса-элемента. Отметим, однако, что класс композиции может нуждаться в перекомпиляции.

Упражнения для самопроверки

- 9.1. Заполнить пробелы в следующих утверждениях:
- Если класс **Alpha** наследует классу **Beta**, класс **Alpha** называется _____ классом, а класс **Beta** — _____ классом.
 - C++ обеспечивает _____, которое позволяет производному классу наследовать нескольким базовым классам, даже если эти базовые классы не родственные.
 - Наследование предоставляет возможность _____, что экономит время разработки и способствует использованию проверенного и высококачественного программного обеспечения.
 - Объект _____ класса можно рассматривать как объект соответствующего ему _____ класса.
 - Для преобразования типа указателя базового класса в тип производного класса, должно быть использовано _____, потому что компилятор считает такую операцию опасной.
 - Существуют три спецификатора доступа к элементу: _____, _____ и _____.
 - При порождении класса от базового класса открытым наследованием открытые элементы базового класса становятся _____ элементами производного класса, а защищенные элементы базового класса становятся _____ элементами производного класса.
 - При порождении класса от базового класса защищенным наследованием открытые элементы базового класса становятся _____ элементами производного класса, а защищенные элементы базового класса становятся _____ элементами производного класса.

- i) Отношение между классами «содержит» представляет _____, а отношение «является» представляет _____.

Ответы на упражнения для самопроверки

- 9.1. a) производным, базовым. b) множественное наследование. c) повторного использования программного обеспечения. d) производного, базового. e) приведение типа. f) `public`, `protected`, `private`. g) открытыми, защищенными. h) защищенными, защищенными. i) композицию, наследование.

Упражнения

- 9.2. Рассмотрите класс `bycicle` (велосипед). Демонстрируя ваши знания о некоторых типичных компонентах велосипеда, покажите иерархию классов, в которой класс `bycicle` наследует другим классам, которые, в свою очередь, наследуют еще каким-то другим классам. Обсудите возникновение различных объектов класса `bycicle`. Обсудите наследование классу `bycicle` других тесно связанных производных классов.
- 9.3. Кратко определите каждый из следующих терминов: наследование, множественное наследование, базовый класс и производный класс.
- 9.4. Обсудите, почему преобразование типа указателя базового класса в тип производного класса компилятор считает опасным.
- 9.5. Укажите различия простого и множественного наследования.
- 9.6. (*Верно, или неверно*) Производный класс часто называют подклассом, потому что он представляет подмножество своего базового класса, т.е. производный класс в общем случае меньше, чем его базовый класс.
- 9.7. (*Верно, или неверно*) Объект производного класса является также объектом своего базового класса.
- 9.8. Некоторые программисты предпочитают не использовать защищенный доступ, потому что он нарушает инкапсуляцию базового класса. Обсудите относительные достоинства защищенного доступа по сравнению с использованием закрытого доступа в базовом классе.
- 9.9. Многие программы, написанные с применением наследования, могли бы быть реализованы с применением композиции и наоборот. Обсудите относительные достоинства этих двух походов в контексте иерархии классов `Point`, `Circle`, `Cylinder`, рассмотренных в этой главе. Перепишите программу на рис. 9.10 (и поддерживающие классы) с использованием композиции вместо наследования. После этого оцените заново относительные достоинства обоих подходов для проблемы `Point`, `Circle`, `Cylinder`, а также для объектно-ориентированных программ вообще.
- 9.10. Перепишите программу `Point`, `Circle`, `Cylinder` на рис. 9.10 как программу `Point`, `Square` (квадрат), `Cube` (куб). Сделайте это двумя способами: один раз с наследованием и один раз с композицией.

- 9.11.** В этой главе мы заявили: «Если элемент базового класса не подходит для производного класса, этот элемент можно переопределить в производном классе и изменить его реализацию». Если это сделать, сохранится ли отношение «производный класс является объектом базового класса»? Объясните свой ответ.
- 9.12.** Изучите иерархию наследования на рис. 9.2. Для каждого класса укажите некоторые общие атрибуты и функции, определяющие иерархию. Добавьте некоторые другие классы, такие, как *UndergraduateStudent* (студент), *GraduateStudent* (аспирант), *Freshman* (первокурсник), *Sophomore* (второкурсник), *Junior* (младшекурсник), *Senior* (старшекурсник) и т.д., чтобы обогатить иерархию.
- 9.13.** Напишите иерархию наследования для класса *Quadrilateral* (четырехугольник), *Trapezoid* (трапеция), *Parallelogram* (параллелограмм), *Rectangle* (прямоугольник) и *Square* (квадрат). Используйте *Quadrilateral* как базовый класс иерархии. Сделайте иерархию настолько глубокой (т.е. настолько многоуровневой), насколько это возможно. Закрытыми данными класса *Quadrilateral* должны быть пары координат (x, y) четырех угловых точек *Quadrilateral*. Напишите программу драйвер, который создает и отображает объекты каждого из этих классов.
- 9.14.** Напишите все двумерные и трехмерные формы, которые вы сможете придумать, и сформируйте из них иерархию форм. Ваша иерархия должна иметь базовый класс *Shape*, порождающий классы *TwodimensionalShape* и *ThreedimensionalShape*. Когда разработаете иерархию, определите каждый из классов в этой иерархии. Мы будем использовать эту иерархию в упражнениях главы 10 для обработки всех форм как объектов базового классовой *Shape*. Эта техника называется полиморфизмом.

г л а в а

10

Виртуальные функции и полиморфизм



Ц е л и

- Понять способы записи полиморфизма.
- Понять, как объявлять и использовать виртуальные функции для полиморфизма.
- Понять различие между абстрактными и конкретными классами.
- Научиться объявлять чистые виртуальные функции для создания абстрактных классов.
- Понять, почему полиморфизм делает системы легко расширяемыми и сопровождаемыми.

План

- 10.1. Введение**
- 10.2. Поля типов и операторы *switch***
- 10.3. Виртуальные функции**
- 10.4. Абстрактные классы и конкретные классы**
- 10.5. Полиморфизм**
- 10.6. Учебный пример: система расчета заработной платы**
- 10.7. Новые классы и динамическое связывание**
- 10.8. Виртуальные деструкторы**
- 10.9. Учебный пример: интерфейс наследования и его реализация**

Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

10.1 Введение

С помощью *виртуальных функций* и *полиморфизма* можно разрабатывать и реализовывать системы, которые являются более *расширяемыми*. Программы могут быть написаны для обобщенной обработки объектов всех существующих в иерархии классов как объектов базового класса. Если в момент разработки программы отсутствуют какие-либо полезные классы, их можно в дальнейшем добавлять с незначительными изменениями или вообще без изменений общей части программы при условии, что эти новые классы являются частью уже обрабатываемой иерархии. Единственными частями программы, которые необходимо модифицировать, являются части, требующие непосредственного знания конкретного класса, добавляемого в иерархию.

10.2. Поля типов и операторы switch

Одним из средств управления объектами различных типов является применение оператора **switch**, который может по-разному обрабатывать различные объекты в зависимости от их типов. Например, в иерархии форм, в которой каждая форма хранит свой тип в некотором поле — элементе данных, структура выбора **switch** способна определить по значению этого поля, какую из функций **print** надо вызвать для работы с объектом данного типом. Однако, при использовании логики оператора **switch** возникает множество проблем. Программист может забыть произвести проверку типа объекта, когда она необходима. Он также может забыть проверить все возможные случаи в операторе **switch**. Если программа, основанная на использовании оператора **switch**, модифицируется и в нее добавляются новые типы объектов, то программист может забыть вставить новые случаи в существующие операторы **switch**. Каждое добавление и удаление класса требует изменения каждого соответствующего оператора **switch** в системе. Отслеживание всего этого требует немалого времени и может являться источником ошибок.

Как мы увидим, виртуальные функции и полиморфное программирование может исключить потребность в применении логики оператора **switch**. Программист может использовать механизм виртуальных функций для автоматического выполнения той же логики, избегая при этом различных ошибок, типичных для применения оператора **switch**.

Замечание по технике программирования 10.1

Интересным следствием использования виртуальных функций и полиморфизма является то, что программы приобретают более простой вид. Они включают меньше логических ветвлений и больше простого последовательного кода. Это упрощение облегчает тестирование, отладку и сопровождение программ.

10.3. Виртуальные функции

Предположим, что ряд классов форм, таких, как **Circle** (круг), **Triangle** (треугольник), **Rectangle** (прямоугольник), **Square** (квадрат) и т.д., являются производными от базового класса **Shape** (форма). В объектно-ориентированном программировании каждый из этих классов может быть наделен способностью нарисовать свою форму. Хотя каждый класс имеет свою функцию рисования **draw**, для разных форм эти функции совершенно различны. При рисовании любой формы, какая бы она ни была, было бы прекрасно иметь возможность работать со всеми этими формами в целом как с объектами базового класса **Shape**. Тогда для рисования любой формы мы могли бы просто вызвать функцию **draw** базового класса **Shape** и предоставить программе динамически (т.е. во время выполнения программы) определять, какую из функций **draw** производного класса следует использовать.

Для того, чтобы предоставить такого рода возможность, объявим функцию **draw** *виртуальной функцией* и затем *переопределим* функцию **draw** в каждом производном классе, чтобы она рисовала соответствующую форму.

Функция объявляется виртуальной с помощью ключевого слова `virtual`, предшествующего прототипу функции в базовом классе. Например, в базовом классе `Shape` можно написать

```
virtual void draw() const;
```

Этот прототип объявляет, что функция `draw` является константной функцией, которая не принимает никаких аргументов, ничего не возвращает и является виртуальной функцией.

Замечание по технике программирования 10.2

Если функция однажды объявлена виртуальной, то она остается виртуальной на любом более низком уровне иерархической структуры.

Хороший стиль программирования 10.1

Несмотря на то, что некоторые функции могут быть неявно виртуальными, поскольку они объявлены такими на более высоком уровне иерархии, некоторые программисты предпочитают явно объявлять функции виртуальными на каждом уровне иерархии, чтобы обеспечить ясность программы.

Замечание по технике программирования 10.3

Если в производном классе решено не описывать виртуальную функцию, то производный класс непосредственно наследует описание виртуальной функции из базового класса.

Если функция `draw` в базовом классе объявлена как `virtual` и если мы затем вызываем функцию `draw` через указатель базового класса, указывающий на объект производного класса (например, `shapePtr->draw()`), то программа будет динамически (т.е. во время выполнения программы) выбирать соответствующую функцию `draw` производного класса. Это называется *динамическим связыванием* и будет продемонстрировано в учебных примерах разделов 10.6 и 10.9.

Замечание по технике программирования 10.4

Переопределенная виртуальная функция должна иметь тот же самый тип возвращаемого значения и ту же сигнатуру, что и виртуальная функция базового класса.

Типичная ошибка программирования 10.1

Если в производном классе переопределяется виртуальная функция базового класса и данная функция не имеет тот же тип возвращаемого значения и ту же сигнатуру, что и соответствующая функция базового класса, то возникает синтаксическая ошибка.

Когда виртуальная функция вызывается путем обращения к заданному объекту по имени и при этом используется операция доступа к элементу точка (например, `squareObject.draw()`), тогда эта ссылка разрешается (обрабатывается) во время компиляции (это называется *статическим связыванием*) и в качестве вызываемой определяется функция класса данного объекта (или наследуемая этим классом).

10.4. Абстрактные базовые классы и конкретные классы

Когда мы думаем о классе как о типе, мы предполагаем, что будут создаваться объекты этого типа. Однако, имеются случаи, в которых полезно определять классы, для которых программист не намерен создавать какие-либо объекты. Такие классы называются *абстрактными классами*. Поскольку они применяются в качестве базовых классов в процессе наследования, мы обычно будем называть их *абстрактными базовыми классами*. Объекты абстрактного базового класса не могут быть реализованы.

Единственным назначением абстрактного класса является создание соответствующего базового класса, от которого другие классы могут унаследовать интерфейс и реализацию. Классы, объекты которых могут быть реализованы, называются *конкретными классами*.

У нас может быть абстрактный базовый класс `TwoDimensionalObject` (двумерный), из которого мы можем получить конкретные классы, такие, как `Square`, `Circle`, `Triangle` и т.д. У нас может быть также абстрактный базовый класс `ThreeDimensionalObject` (трехмерный), из которого можно получить конкретные классы, такие, как `Cube`, `Sphere`, `Cylinder` и т.д. Абстрактные базовые классы являются слишком общими для определения реальных объектов; нам требуется больше определенности, чтобы можно было думать о реализации объектов. Для этого предназначены конкретные классы; они обладают необходимой спецификой, делающей реальным создание объектов.

Класс делается абстрактным путем объявления одной или более его виртуальных функций чисто виртуальными. Чистой виртуальной функцией является такая функция, у которой в ее объявлении тело определено как 0 (инициализатор равен 0); например:

```
virtual float earnings( ) const = 0;      // чистая виртуальная
                                            // функция
```

Замечание по технике программирования 10.5

Если класс является производным от класса с чистой виртуальной функцией и если эта чистая виртуальная функция не описана в производном классе, тогда функция остается чистой виртуальной и в производном классе. Следовательно, такой производный класс также является абстрактным классом.

Типичная ошибка программирования 10.2

Попытка создать объект абстрактного класса (т.е. класса, который содержит хотя бы одну чистую виртуальную функцию) является синтаксической ошибкой.

Иерархия не требует обязательного включения каких-либо абстрактных классов. Но, как мы увидим далее, многие хорошие программы, использующие объектно-ориентированное программирование, имеют иерархию классов, порожденную абстрактным базовым классом. В некоторых случаях абстрактные классы составляют несколько верхних уровней иерархии. Наглядным примером этого является иерархия форм. Иерархия может порождаться абстрактным базовым классом `Shape`. При переходе на один уровень ниже мы может иметь на два абстрактных класса больше, например, абстрактные ба-

зовые классы **TwoDimensionalObject** и **ThreeDimensionalObject**. При переходе еще на один уровень ниже мы смогли бы определить конкретные классы для таких двумерных форм, как круги и квадраты, а также определить конкретные классы для таких трехмерных форм, как сфера и куб.

10.5. Полиморфизм

C++ включает такое свойство, как *полиморфизм* — возможность для объектов разных классов, связанных с помощью наследования, реагировать различным образом при обращении к одной и той же функции-элементу. Если, например, класс **Rectangle** (прямоугольник) является производным от класса **Quadrilateral** (четырехугольник), то тогда объект класса **Rectangle** является более конкретизированным вариантом объекта класса **Quadrilateral**. Операция (такая, как вычисление периметра или площади), которая может быть выполнена для объекта класса **Quadrilateral**, может быть также выполнена и для объекта класса **Rectangle**.

Полиморфизм реализуется посредством виртуальных функций. Если при использовании виртуальной функции запрос осуществляется с помощью указателя базового класса (или ссылки), то C++ выбирает правильную переопределенную функцию в соответствующем производном классе, связанном с данным объектом.

Иногда функция-элемент определена в базовом классе не как виртуальная, но переопределена в производном классе. Если такая функция-элемент вызывается через указатель базового класса, то используются версия базового класса. Если же эта функция-элемент вызывается через указатель производного класса, то используется версия производного класса. Это не полиморфное поведение.

Рассмотрим следующий пример, в котором использованы базовый класс **Employee** (служащие) и производный класс **HourlyWorker** (почасовики), приведенные на рис. 9.5.

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print; // вызов функции-элемента print базового класса
hPtr->print; // вызов функции-элемента print производного класса
```

Наш базовый класс **Employee** и производный класс **HourlyWorker** имеют свои собственные функции **print**, которые определены в этих классах. Поскольку эти функции не объявлены виртуальными и имеют одинаковую сигнатуру, то вызов функции **print** через указатель класса **Employee** приводит к вызову функции **print** из базового класса **Employee**, т.е. выполняется операция **Employee::print()**, а вызов функции **print** через указатель класса **HourlyWorker** приводит к вызову функции **print** из производного класса **HourlyWorker**. Функция **print** базового класса также является элементом производного класса, но при вызове функции **print** для объекта производного класса вариант функции базового класса должен быть вызван явным образом, как показано ниже :

```
hPtr->Employee::print();
```

Эта запись явным образом определяет, что должна быть вызвана функция **print** базового класса.

Благодаря применению виртуальных функций и полиморфизму вызов функции-элемента может привести к различным действиям, которые зависят от типа вызываемого объекта (мы увидим впоследствии, что это требует незначительного количества дополнительных затрат). Подобное полиморфное поведение предоставляет программисту огромные возможности. В нескольких следующих разделах мы увидим примеры, которые демонстрируют, каким мощным средством являются полиморфизм и виртуальные функции.

Замечание по технике программирования 10.6

Благодаря виртуальным функциям и полиморфизму программист может управлять общими свойствами объектов, предоставляя возможность программной среде во время выполнения программы самой «заботиться» о специфике объектов. Программист может управлять широким спектром объектов, даже не зная об их типах, причем управление будет автоматически учитывать специфику этих объектов.

Замечание по технике программирования 10.7

Полиморфизм способствует расширяемости: программное обеспечение, использующее полиморфный механизм, пишется независимо от типов объектов, которым отправляются сообщения. Таким образом, новые типы объектов, которые должны реагировать на соответствующие сообщения, могут включаться в такую систему без модификации ее основы. За исключением кода пользователя, который создает новые объекты, программа не потребует перекомпиляции.

Замечание по технике программирования 10.8

Абстрактный класс определяет интерфейс для разных типов элементов иерархии классов. Абстрактный класс включает чистые виртуальные функции, которые будут определены в производных классах. Все функции в иерархии могут применять один и тот же интерфейс, используя полиморфизм.

Хотя мы не можем создавать объекты абстрактного базового класса, мы *можем* объявить указатели на абстрактный базовый класс. Эти указатели могут быть затем использованы, чтобы предоставить возможность для полиморфного оперирования объектами производных конкретных классов.

Рассмотрим применение полиморфизма и виртуальных функций. Экранному администратору необходимо отображать множество объектов, включая новые типы объектов, которые будут включаться в систему даже после того, как будет написана сама программа экранного администратора. Системе может потребоваться отображать разные формы (базовым классом для них является класс *Shape*), такие, как квадраты, окружности, треугольники, прямоугольники и т.п. Каждый из этих классов форм является производным от базового класса *Shape*. Экранный администратор использует указатели базового класса (указатели на *Shape*) для управления всеми отображаемыми на экране объектами. Чтобы нарисовать объект (безотносительно уровня, на котором находится объект в иерархии наследования), экранный администратор использует указатель базового класса на объект и просто посыпает сообщение объекту: «нарисовать». Функция *draw* объявляется чистой виртуальной функцией в базовом классе *Shape* и должна быть переопределена в каждом из производных классов. Каждый объект базового класса *Shape* «знает как себя нарисовать». Экранному администратору не надо заботиться

о том, какого типа каждый объект и вообще имел ли дело когда-нибудь экранный администратор с объектами такого типа; просто экранный администратор «приказывает» каждому объекту изобразить себя на экране.

Полиморфизм особенно эффективен при реализации многоуровневых систем программного обеспечения. В операционных системах, например, каждый тип физического устройства может работать совершенно отлично от других. Несмотря на это, команды чтения данных с таких физических устройств *read* и команды записи данных на физические устройства *write* могут иметь определенное единство. Сообщение *write*, посланное объекту драйверу устройства, интерпретируется специальным образом в зависимости от используемого драйвера и от того, каким образом этот драйвер управляет устройствами конкретного типа. Тем не менее, этот вызов *write* сам по себе действительно не отличается от вызовов *write* любого другого устройства в системе; вызов просто перемещает некоторое число байтов из памяти в конкретное устройство. Объектно-ориентированные операционные системы могут использовать абстрактные базовые классы для того, чтобы реализовать интерфейс, пригодный для драйверов всех устройств. Затем с помощью наследования этих абстрактных базовых классов образуются производные классы, которые работают одинаковым образом. Указанные возможности (т.е. открытый интерфейс), предоставленные драйверам устройств, обеспечиваются чистыми виртуальными функциями абстрактных базовых классов. Реализации этих виртуальных функций обеспечиваются в производных классах и соответствуют конкретным типам драйверов устройств.

В главе 7 мы ввели понятие итераторов. Обыкновенно определяют *класс итераторов*, который обеспечивает обход всех объектов набора. Если, например, вы хотите напечатать перечень объектов в связном списке, то объект итератор может быть реализован таким образом, что он будет возвращать следующий элемент связного списка при каждом вызове этого итератора. Итераторы обычно применяются в полиморфном программировании для обхода в связном списке объектов различных иерархических уровней. Все указатели в таком списке могут быть указателями базового класса (см. главу 15 «Структуры данных», в которой рассматриваются связные списки). Список объектов класса *TwoDimensionalShape* может включать объекты из классов *Square*, *Circle*, *Triangle* и т.д. При использовании полиморфизма отправка сообщения «нарисовать» каждому объекту списка обеспечивала бы изображение на экране правильной картинки.

10.6. Учебный пример : система расчета заработной платы

Давайте используем виртуальные функции и полиморфизм, чтобы выполнить расчет заработной платы, в котором учитывается тип служащего (см. рис. 10.1). Воспользуемся базовым классом *Employee*. Производными классами от базового класса *Employee* являются: класс *Boss* — служащим начисляется еженедельный фиксированный оклад независимо от числа проработанных часов; класс *CommissionWorker* — служащим начисляется базовая заработка плата плюс комиссионный процент от продаж; класс *PieceWorker* — служащим начисляется сдельная плата по количеству изготовленных изделий; класс *HourlyWorker* — служащим начисляется почасовая

заработка платы за часы, отработанные в основное время, и повышенная плата за часы, отработанные сверхурочно.

Вызов функции `earnings` используется для всех служащих. Но способы, которыми начисляется заработка плата, зависят, конечно, от классов служащих и все эти классы являются производными от базового класса `Employee`. Поэтому функция `earnings` объявляется в базовом классе `Employee` как виртуальная, а соответствующие ее реализации обеспечиваются в каждом производном классе. Тогда, чтобы вычислить заработную плату какого-либо служащего, программа просто использует указатель базового класса на объект, соответствующий этому служащему, и вызывает функцию `earnings`. В реальной системе начисления заработной платы разные объекты служащих могли бы указываться отдельными элементами массива указателей типа `Employee *`. Программа могла бы просто проходить по этому массиву, используя указатели `Employee *` для вызов функций `earnings` каждого объекта.

Давайте рассмотрим класс `Employee` (рис. 10.1, части 1 и 2). Открытые функции-элементы включают: конструктор, который принимает в качестве аргументов имя и фамилию; деструктор, который освобождает динамически выделенную память; функцию доступа `get`, которая возвращает имя; функцию доступа `get`, которая возвращает фамилию; две чистые виртуальные функции `earnings` и `print`. Почему эти функции объявлены чистыми виртуальными? Ответ состоит в том, что не имеет никакого смысла реализовывать эти функции в классе `Employee`. Мы не можем начислять заработную плату абстрактному служащему: мы должны, сначала определить тип служащего; мы не можем также печатать заработную плату абстрактного служащего. Делая эти функции чистыми виртуальными, мы показываем, что они должны быть реализованы в производных классах, а не в базовом.

```
// 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 абстрактного базового класса
    // Employee
    virtual float earnings() const = 0;           // чистая виртуальная
    virtual void print() const = 0;                 // чистая виртуальная

private:
    char *firstName;
    char *lastName;
};

#endif
```

Рис. 10.1. Абстрактный базовый класс `Employee` (часть 1 из 12)

```

// EMPLOY2.CPP
// Определения функций-элементов
// абстрактного базового класса Employee
//
// Замечание: не дается никаких определений чистых
// виртуальных функций
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ2.h"

// Конструктор динамически выделяет память для имени и фамилии
// и использует функцию strcpy для копирования их в объект
Employee::Employee(const char *first, const char *last)
{
    firstName = new char[ strlen(first) + 1 ];
    assert(firstName != 0);                                // проверка работы new

    strcpy(firstName, first);

    lastName = new char[ strlen(last) + 1 ];
    assert(lastName != 0);                                // проверка работы new
    strcpy(lastName, last);
}

// Деструктор освобождает динамически выделенную память
Employee::~Employee()
{
    delete [] firstName;
    delete [] lastName;
}

// Возвращение указателя на имя
const char *Employee::getFirstName() const
{
    // const препятствует модификации скрытых данных
    // со стороны вызывающей программы.
    // Чтобы не допустить неопределенного указателя, вызывающая
    // программа должна копировать возвращаемую строку до того, как
    // деструктор освободит динамически выделенную область памяти

    return firstName;           // Вызывающая программа должна
                               // освободить память
}

// Возвращение указателя на фамилию
const char *Employee::getLastName() const
{
    // const препятствует модификации скрытых данных со стороны
    // вызывающей программы. Чтобы не допустить неопределенного
    // указателя, вызывающая программа должна копировать
    // возвращаемую строку до того, как деструктор освободит
    // динамически выделенную область памяти

    return lastName;           // Вызывающая программа должна освободить
                               // память
}

```

Рис. 10.1. 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. 10.1. Класс **Boss**, производный от абстрактного базового класса **Employee** (часть 3 из 12)

```
// BOSS1.CPP
// Определения 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
{ setWeeklySalary(s); }

// Установка еженедельного оклада для класса Boss
void Boss::setWeeklySalary(float s)
{ weeklySalary = s > 0 ? s : 0; }

// Начисление заработной платы в классе Boss
float Boss::earnings() const { return weeklySalary; }

// Печать имени служащего из класса Boss
void Boss::print() const
{
    cout << endl << "                  Администратор: "
        << getFirstName()
        << ' ' << getLastName();
}
```

Рис. 10.1. Определения функций-элементов класса **Boss** (часть 4 из 12)

Класс **Boss** (рис. 10.1, части 3 и 4) является производным от класса **Employee** с открытым наследованием. Открытые функции-элементы включают: конструктор, который принимает в качестве аргументов имя, фамилию и еженедельный оклад, а также передает имя и фамилию конструктору **Employee** для инициализации элементов **firstName** и **lastName** той части объекта производного класса, которая совпадает с базовым классом; функцию **set**, которая присваивает новое значение скрытому элементу данных **weeklySalary**; виртуальную функцию **earnings**, в которой определено, как начис-

лять заработную плату в классе **Boss**; виртуальную функцию **print**, которая выводит тип служащего и его имя.

```
// COMMIS1.H
// Класс CommissionWorker, производный от 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, unsigned = 0);

    void setSalary(float);
    void setCommission(float);
    void setQuantity(unsigned);
    virtual float earnings() const;
    virtual void print() const;

private:
    float salary;           // Базовая заработная плата за неделю
    float commission;       // Комиссионные от продажи одного изделия
    unsigned quantity;      // Количество проданных изделий за неделю
};

#endif
```

Рис. 10.1. 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, unsigned q)
: Employee(first, last) // вызов конструктора базового класса
{
    salary = s > 0 ? s : 0;
    commission = c > 0 ? c : 0;
    quantity = q > 0 ? q : 0;
}

// Установка еженедельного основного оклада для класса
// CommissionWorker
void CommissionWorker::setSalary(float s)
{ salary = s > 0 ? s : 0; }

// Установка комиссионных для класса CommissionWorker
void CommissionWorker::setCommission(float c)
{ commission = c > 0 ? c : 0; }

// Установка количества продаж изделий
void CommissionWorker::setQuantity(unsigned q)
```

```

    { quantity = q > 0 ? q : 0; }

// Определение заработной платы служащих из класса ComissionWorker
float CommissionWorker::earnings() const
{ return salary + commission * quantity; }

// Печать имени служащего из класса CommissionWorker
void CommissionWorker::print() const
{
    cout << endl << "    Служащий на комиссионных: "
        << getFirstName() << ' ' << getLastName();
}

```

Рис. 10.1. Определения функций-элементов класса **ComissionWorker** (часть 6 из 12)

Класс **ComissionWorker** (части 5 и 6 рис. 10.1) является производным от класса **Employee** открытым наследованием. Открытые функции-элементы включают: конструктор, который принимает в качестве аргументов имя, фамилию, базовую недельную заработную плату, комиссионное вознаграждение и количество проданных изделий, а также передает имя и фамилию конструктору **Employee**; функцию *set*, которая присваивает новые значения скрытым данным-элементам **salary**, **comission** и **quantity**; виртуальную функцию **earnings**, в которой определено, как начислять заработную плату в классе **ComissionWorker**; виртуальную функцию **print**, которая выводит тип служащего и его имя.

Класс **PieceWorker** (рис. 10.1, части 7 и 8) является производным от класса **Employee** с открытым наследованием. Открытые функции-элементы включают: конструктор, который принимает в качестве аргументов имя, фамилию, оплату за единицу продукции и количество произведенной продукции за неделю, а также передает имя и фамилию конструктору **Employee**; функцию *set*, которая присваивает новые значения скрытым данным-элементам **wagePerPiece** и **quantity**; виртуальную функцию **earnings**, в которой определено, как начислять заработную плату в классе **PieceWorker**; виртуальную функцию **print**, которая выводит тип служащего и его имя.

```

// PIECE1.H
// Класс PieceWorker, производный от Employee
#ifndef PIECE1_H
#define PIECE1_H
#include "employ2.h"

class PieceWorker : public Employee {
public:
    PieceWorker(const char*, const char*, float = 0.0, unsigned = 0);
    void setWage(float);
    void setQuantity(unsigned);
    virtual float earnings() const;
    virtual void print() const;
private:
    float wagePerPiece;           // Оплата единицы продукции
    unsigned quantity;           // Число единиц продукции за неделю
};

#endif

```

Рис. 10.1. Класс **PieceWorker**, производный от абстрактного базового класса **Employee** (часть 7 из 12)

```

// PIECEL.CPP
// Определения функций-элементов класса PieceWorker
#include <iostream.h>
#include "piece1.h"

// Конструктор класса PieceWorker
PieceWorker::PieceWorker (const char *first, const char *last,
                          float w, unsigned q)
    : Employee(first, last) // Вызов конструктора базового класса
{
    wagePerPiece = w > 0 ? w : 0;
    quantity = q > 0 ? q : 0;
}

// Установка сдельной заработной платы
void PieceWorker::setWage(float w)
    { wagePerPiece = w > 0 ? w : 0; }

// Установка количество произведенных единиц продукции
void PieceWorker::setQuantity(unsigned q)
    { quantity = q > 0 ? q : 0; }

// Определение заработной платы служащих из класса PieceWorker
float PieceWorker::earnings() const
    { return quantity * wagePerPiece; }

// Печать имени служащего из класса PieceWorker
void PieceWorker::print() const
{
    cout << endl << "Служащий на сдельной оплате: "
        << getFirstName()
        << ' ' << getLastName();
}

```

Рис. 10.1. Определения функций-элементов класса **PieceWorker** (часть 8 из 12)

```

// HOURLY1. H
// Класс HourlyWorker, производный от Employee
#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;           // Заработка плата за час
    float hours;          // Количество отработанных часов за неделю
};

#endif

```

Рис. 10.1. Класс **HourlyWorker**, производный от абстрактного базового класса **Employee** (часть 9 из 12)

```
// HOURLY1.CPP
// Определения 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 почасовой оплаты
void HourlyWorker::setWage(float w)
{ wage = w > 0 ? w : 0; }

// Установка числа отработанных часов
void HourlyWorker::setHours(float h)
{ hours = h >= 0 && h < 168 ? h : 0; }

// Определение заработной платы служащих из класса HourlyWorker
float HourlyWorker::earnings() const { return wage * hours; }

// Печать имени служащего из класса HourlyWorker
void HourlyWorker::print() const
{
    cout << endl << "Служащий с почасовой оплатой: "
        << getFirstName() << ' ' << getLastName();
}
```

Рис. 10.1. Определения функций-элементов класса **HourlyWorker** (часть 10 из 12)

Класс **HourlyWorker** (рис. 10.1, части 9 и 10) является производным от класса **Employee** с открытым наследованием. Открытые функции-элементы включают: конструктор, который принимает в качестве аргументов имя, фамилию, почасовую оплату и количество отработанных часов, а также передает имя и фамилию конструктору **Employee**; функцию *set*, которая присваивает новые значения скрытым данным-элементам *wage* и *hours*; виртуальную функцию *earnings*, в которой определено, как начислять заработную плату в классе **HourlyWorker**; виртуальную функцию *print*, которая выводит тип служащего и его имя.

Программа драйвер (рис. 10.1, части 11 и 12) начинается с объявления указателя базового класса *ptr* типа **Employee ***. Последующие три сегмента кода в функции **main** сходны друг с другом, поэтому мы обсудим только первый сегмент, который связан с объектом **Boss**.

```
// FIG10 1.CPP
// Драйвер иерархии Employee

#include <iostream.h>
#include <iomanip.h>
#include "employ2.h"
#include "boss1.h"
#include "commis1.h"
#include "piece1.h"
#include "hourly1.h"

main()
{
    // Установка выходного формата
    cout << setiosflags(ios::showpoint) << setprecision(2);

    Employee *ptr;           // Указатель базового класса

    Boss b("John", "Smith", 800.0);
    ptr = &b;                // Указатель базового класса на объект
                            // производного класса
    ptr->print();           // Динамическое связывание
    cout << " заработал $" << ptr->earnings(); // Динамическое
                                                // связывание
    b.print();               // Статическое связывание
    cout << " заработал $" << b.earnings(); // Статическое
                                                // связывание

    CommissionWorker c("Sue", "Jones", 200.0, 3.0, 150);
    ptr = &c;                // Указатель базового класса на объект
                            // производного класса
    ptr->print();           // Динамическое связывание
    cout << " заработал $" << ptr->earnings(); // Динамическое
                                                // связывание
    c.print();               // Статическое связывание
    cout << " заработал $" << c.earnings(); // Статическое
                                                // связывание

    PieceWorker p("Bob", "Lewis", 2.5, 200);
    ptr = &p;                // Указатель базового класса на объект
                            // производного класса
    ptr->print();           // Динамическое связывание
    cout << " заработал $" << ptr->earnings(); // Динамическое
                                                // связывание
    p.print();               // Статическое связывание
    cout << " заработал $" << p.earnings(); // Статическое
                                                // связывание

    HourlyWorker h("Karen", "Price", 13.75, 40);
    ptr = &h;                // Указатель базового класса на объект
                            // производного класса
    ptr->print();           // Динамическое связывание
    cout << " заработал $" << ptr->earnings(); // Динамическое
                                                // связывание
    h.print();               // Статическое
                            // связывание
```

```

cout << " заработал $" << h.earnings();           // Статическое
// связывание

cout << endl;

return 0;
)

```

Рис. 10.1. Иерархия на основе абстрактного базового класса **Employee** (часть 11 из 12)

```

Администратор: John Smith заработал $800.00
Администратор: John Smith заработал $800.00
Служащий на комиссионных: Sue Jones заработал $650.00
Служащий на комиссионных: Sue Jones заработал $650.00
Служащий на сдельной оплате: Bob Lewis заработал $500.00
Служащий на сдельной оплате: Bob Lewis заработал $650.00
Служащий с почасовой оплатой: Karen Price заработал $550.00
Служащий с почасовой оплатой: Karen Price заработал $650.00

```

Рис. 10.1. Иерархия на основе абстрактного базового класса **Employee** (часть 12 из 12)

Строка

```
Boss b("John", "Smith", 800.00);
```

создает объект **b** производного класса **Boss** и снабжает конструктор аргументами, включающими имя, фамилию и фиксированный еженедельный оклад.

Строка

```
ptr = &b;           // Указатель базового класса на объект
// производного класса
```

помещает в указатель базового класса **ptr** адрес объекта производного класса **b**. Это то, что мы должны сделать для реализации полиморфного поведения.

Строка

```
ptr->print();           // динамическое связывание
```

вызывает функцию-элемент **print** того объекта, на который указывает **ptr**. Поскольку функция **print** объявлена в базовом классе как виртуальная, система вызывает функцию **print** объекта производного класса, как и положено при полиморфном поведении. Это обращение к функции является примером динамического связывания: функция вызывается через указатель базового класса, поэтому выбор того, какую функцию вызвать, откладывается до времени выполнения программы.

Строка

```
cout << " заработал $" << ptr->earnings();      // Динамическое
// связывание
```

вызывает функцию-элемент **earnings** того объекта, на который указывает **ptr**. Поскольку функция **earnings** объявлена в базовом классе как виртуальная, система вызывает функцию **earnings** объекта производного класса. Это также пример динамического связывания.

Строка

```
b.print(); // Статическое связывание
```

явным образом вызывает функцию-элемент `print` класса `Boss`, используя операцию доступа к элементу точка из заданного объекта `b` класса `Boss`. Это является примером статического связывания, поскольку тип объекта, для которого вызывается функция, известен во время компиляции. Этот вызов включен с целью сравнения, чтобы показать, правильная ли функция `print` вызывается при использовании динамического связывания.

Строка

```
cout << " заработал $" << b.earnings(); // Статическое связывание
```

явно вызывает функцию-элемент `earnings` класса `Boss` с помощью операции доступа к элементу точка из заданного объекта `b` класса `Boss`. Это также пример статического связывания. Этот вызов включается с целью сравнения, чтобы показать, правильная ли функция `earnings` вызывается при использовании динамического связывания.

10.7. Новые классы и динамическое связывание

Полиморфизм и виртуальные функции могут прекрасно работать, если все возможные классы известны заранее. Но они также работают, когда в систему добавляются новые типы классов.

Новые классы встраиваются при помощи динамического связывания (называемого также *поздним связыванием*). Во время компиляции нет необходимости знать тип объекта, чтобы скомпилировать вызов виртуальной функции. Во время выполнения программы вызов виртуальной функции будет соответствовать функции-элементу вызванного объекта.

Например, программа экранного администратора может обрабатывать новые экранные объекты, так как при их добавлении перекомпиляция системы не требуется. Вызов функции `draw` остается прежним. Новые объекты сами содержат возможности для рисования своих форм. Это позволяет легко добавлять в систему новые возможности, минимально затрагивая ее структуру. Это способствует также повторному использованию программного обеспечения.

Динамическое связывание позволяет независимым дистрибутерам программного обеспечения распространять свою продукцию, не выдавая фирменных секретов. Распространяемое программное обеспечение может состоять только из заголовочных и объектных файлов. Чтобы не раскрывать секреты программного обеспечения, не должно прикладываться никаких исходных текстов. Разработчики программного обеспечения могут использовать наследование для создания новых производных классов на основе тех классов, которые предоставлены им дистрибутерами программного обеспечения. Программные средства, которые работают с классами, предоставленными дистрибутерами, будут продолжать работать и с производными классами, используя (с помощью динамического связывания) переопределенные виртуальные функции, имеющиеся в этих классах.

Ниже показано, каким образом компилятор и программа (во время выполнения) управляют полиморфизмом с незначительными затратами.

Динамическое связывание требует, чтобы во время выполнения программы вызов виртуальной функции-элемента был бы направлен варианту вир-

туальной функции соответствующего класса. Для этого служит *таблица виртуальных методов* или *table*, которая реализуется в виде массива, содержащего указатели на функции. У каждого класса, который содержит виртуальные функции, имеется таблица виртуальных методов. Для каждой виртуальной функции в классе таблица имеет элемент, содержащий указатель на вариант виртуальной функции, используемый в объектах данного класса. Виртуальная функция, используемая в некотором классе, может быть определена в этом классе или прямо или косвенно наследоваться из базового класса, стоящего выше в иерархии.

Если базовый класс имеет виртуальную функцию-элемент, то производные классы могут переопределить эту функцию, но они могут этого и не делать. Таким образом, производный класс может использовать вариант виртуальной функции-элемента базового класса и это будет отражено в таблице виртуальных методов.

Каждый объект класса, содержащего виртуальные функции, имеет указатель на таблицу виртуальных методов этого класса, недоступный для программиста. Во время выполнения программы полиморфные вызовы виртуальных функций осуществляются через разыменование указателя объекта на таблицу виртуальных методов, что дает доступ к таблице виртуальных методов класса. Затем в таблице виртуальных методов находится соответствующий указатель на функцию, он разыменовывается, что и завершает вызов виртуальной функции во время выполнения программы. Просмотр таблицы виртуальных методов и операция разыменования указателя требуют минимальных затрат времени выполнения.

Совет по повышению эффективности 10.1

Полиморфизм, реализуемый с помощью виртуальных функций и динамического связывания, очень эффективен. Программисты могут использовать это средство при ничтожном влиянии на производительность системы.

Совет по повышению эффективности 10.2

Виртуальные функции и динамическое связывание позволяют использовать полиморфное программирование как альтернативу логике оператора **switch**. Оптимизирующие компиляторы C++ обычно генерируют код, который исполняется по крайней мере так же эффективно, как код, построенный вручную на основе логики оператора выбора **switch**.

10.8. Виртуальные деструкторы

При использовании полиморфизма для обработки динамически размещенных объектов иерархии классов может появиться одна проблема. Если объект уничтожается явным использованием операции **delete** над указателем базового класса на объект, то вызывается деструктор базового класса данного объекта. Это происходит вне зависимости от типа объекта, на который указывает указатель базового класса и вне зависимости от того факта, что деструкторы каждого класса имеют разные имена.

Существует простое решение этой проблемы: объявление деструктора базового класса виртуальным. Это автоматически приведет к тому, что все деструкторы производных классов станут виртуальными, даже если они

имеют имена, отличные от имени деструктора базового класса. В этом случае, если объект в иерархии уничтожен явным использованием операции `delete`, примененной к указателю базового класса на объект производного класса, то будет вызван деструктор соответствующего класса. Вспомним, что когда производный класс уничтожен, часть базового класса, содержащаяся в производном классе, также уничтожается. Деструктор базового класса автоматически выполняется после деструктора производного класса.

Хороший стиль программирования 10.2

Если у класса имеются виртуальные функции, предусматривайте создание виртуального деструктора, даже если он не требуется этому классу. Классы, производные от данного класса, могут содержать деструкторы, которые должны вызываться соответствующим образом.

Типичная ошибка программирования 10.3

Объявление конструктора виртуальной функцией. Конструкторы не могут быть виртуальными.

10.9. Учебный пример : интерфейс наследования и его реализация

В нашем следующем примере (рис. 10.2) повторно рассматривается иерархия форм точка, круг и цилиндр из предыдущей главе. Но мы дополнительно создаем для этих форм головной элемент этой иерархии в виде абстрактного базового класса `Shape`. У класса `Shape` имеются две чистых виртуальных функции `printShapeName` и `print`, так что `Shape` является абстрактным базовым классом. Класс `Shape` включает также еще две другие виртуальные функции — `area` и `volume`, каждая из которых имеет в классе реализацию, возвращающую нулевое значение. Класс `Point` наследует эти реализации от класса `Shape`. Это имеет смысл, поскольку и площадь, и объем точки равны нулю. Класс `Circle` наследует функцию `volume` класса `Point`, но имеет собственную реализацию функции `area`. Класс `Cylinder` имеет собственные реализации функций `area` и `volume`.

```
// 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
    virtual void print() const = 0; // Чистая виртуальная функция
};

#endif
```

Рис. 10.2. Определение абстрактного базового класса `Shape` (часть 1 из 9)

Заметим, что хотя класс **Shape** является абстрактным базовым классом, он, однако, содержит реализации некоторых функций-элементов; и эти реализации являются наследуемыми. Класс **Shape** предоставляет наследуемый интерфейс в виде четырех виртуальных функций, которые будут входить во все элементы иерархии. В классе **Shape** представлены также некоторые реализации, которые будут использоваться производными классами нескольких первых уровней иерархии.

```
// POINT1.H
// Определение класса Point
#ifndef POINT1_H
#define POINT1_H
#include <iostream.h>
#include "shape.h"

class Point : public Shape {
    friend ostream &operator<<(ostream &, const Point &);
public:
    Point(float = 0, float = 0);           // Конструктор с умолчанием
    void setPoint(float, float);
    float getX() const { return x; }
    float getY() const { return y; }
    virtual void printShapeName() const { cout << "Точка: "; }
    virtual void print() const;
private:
    float x, y;                         // Координаты x и y
                                         // класса Point
};
#endif
```

Рис. 10.2. Определение класса **Point** (часть 2 из 9)

```
// POINT1.CPP
// Определения функций-элементов класса Point
#include <iostream.h>
#include "point1.h"

Point::Point(float a, float b) { setPoint(a, b); }

void Point::setPoint(float a, float b)
{
    x = a;
    y=b;
}

void Point::print() const { cout << '[' <<x <<", " << y << ']'; }

ostream &operator<< (ostream &output, const Point &p)
{
p.print();                      // вызов функции print для печати объекта
return output;                  // допускает сцепленные вызовы
}
```

Рис. 10.2. Определения функций-элементов класса **Point** (часть 3 из 9)

```
// 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 с умолчанием
    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 << "Круг: ";}
    virtual void print() const;
private:
    float radius;           // Радиус круга
};

#endif
```

Рис. 10.2. Определение класса **Circle** (часть 4 из 9)

```
// CIRCLE1.CPP
// Определение функций-элементов класса Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle1.h"

Circle::Circle(float r, float a, float b)
    : Point(a, b)           // Вызов конструктора
                           // базового класса
{ radius = r > 0 ? r : 0; }
void Circle::setRadius(float r) { radius = r > 0 ? r : 0; }

float Circle::getRadius() const { return radius; }

float Circle::area() const { return 3.14159 * radius * radius; }

void Circle::print() const
{
    cout << '[' << getX() << ", " << getY()
        << "]; Радиус = " << setiosflags(ios::showpoint)
        << setprecision(2) << radius;
}

ostream &operator<< (ostream &output, const Circle &c)
{
    c.print();             // вызов функции print для печати объекта
    return output;         // допускает сцепленные вызовы
}
```

Рис. 10.2. Определения функций-элементов класса **Circle** (часть 5 из 9)

```

// CYLINDR1.H
// Определение класса Cylinder
#ifndef CYLINDR1_H
#define CYLINDR1_H
#include "circle1.h"

class Cylinder : public Circle {
    friend ostream &operator<< (ostream &, const Cylinder &);
public:
    // Конструктор с умолчанием
    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 << "Цилиндр: " ; }
    virtual void print() const;

private:
    float height;      // Высота цилиндра
};

#endif

```

Рис. 10.2. Определение класса **Cylinder** (часть 6 из 9)

```

// CYLINDR1.CPP
// Определение функций-элементов и дружественных функций 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 цилиндра
    return 2 * Circle::area() +
        2 * 3.14159 * Circle::getRadius() * height;
}

float Cylinder::volume() const { return Circle::area() * height; }

void Cylinder::print() const
{
    Circle::print();
    cout << "; Высота = " << height;
}

ostream &operator << (ostream &output, const Cylinder& c)
{
    c.print();                 // вызов функции print для печати объекта
    return output;             // допускает сцепленные вызовы
}

```

Рис. 10.2. Определения функций-элементов класса **Cylinder** (часть 7 из 9)

В этом учебном примере делается упор на то, что классы могут наследовать и интерфейс, и реализацию базового класса. В иерархиях, проектируемых для наследования реализации, стремятся обеспечить функциональные возможности на возможно более высоких уровнях, чтобы каждый новый производный класс мог наследовать функции-элементы, описанные в базовом классе, и использовать эти описания. В иерархиях, проектируемых для наследования интерфейса, стремятся обеспечить функциональные возможности на возможно более низких уровнях: базовый класс объявляет функции, которые должны вызываться идентично для каждого объекта в иерархии (т.е. иметь одинаковую сигнатуру), а производные классы обеспечивают для них свои собственные реализации.

Базовый класс `Shape` (рис. 10.2, часть 1) состоит из четырех открытых виртуальных функций и не содержит никаких-либо данных. Функции `printShapeName` и `print` являются чисто виртуальными, так как они переопределяются в каждом производном классе. Функции `area` и `volume` определены, причем по определению они возвращают 0.0. Эти функции переопределяются в тех производных классах, в которых требуется использовать их для соответствующих вычислений площади и объема.

Класс `Point` (рис. 10.2, части 2 и 3) является производным от класса `Shape` с открытым наследованием. Точка не имеет ни площади, ни объема, так что функции-элементы базового класса `area` и `volume` в данном случае не переопределены; они наследуют их определения из базового класса `Shape`. Функции `printShapeName` и `print` являются реализациями виртуальных функций, которые были определены в базовом классе как чисто виртуальные. Другими функциями-элементами являются функция `set`, присваивающая новые значения координатам точки `x` и `y`, и функция `get`, возвращающая координаты `x` и `y`.

Класс `Circle` (рис. 10.2, части 4 и 5) является производным от класса `Point` с открытым наследованием. Круг не имеет объема, так что функция-элемент базового класса `volume` для класса `Circle` не переопределяется; она наследуется из базового класса `Shape` (через класс `Point`). Круг имеет площадь; поэтому функция `area` в классе `Circle` переопределяется. Функции `printShapeName` и `print` являются реализациями виртуальных функций, которые были определены в базовом классе как чисто виртуальные. Если бы эти функции здесь не переопределялись, то наследовались бы версии этих функций из класса `Point`. Другими функциями-элементами являются функция `set`, присваивающая новое значение радиусу `radius`, и функция `get`, возвращающая значение радиуса.

Класс `Cylinder` (рис. 10.2, части 6 и 7) является производным от класса `Circle` с открытым наследованием. Цилиндры имеют площадь и объем; поэтому в классе `Cylinder` обе функции `area` и `volume` переопределены. Функции `printShapeName` и `print` являются реализациями виртуальных функций, которые были определены в базовом классе как чисто виртуальные. Если бы эти функции здесь не переопределялись, то наследовались бы версии этих функций из класса `Circle`. Другими функциями-элементами являются функция `set`, присваивающая новое значение высоты цилиндра `height`, и функция `get`, возвращающая значение высоты.

```
// FIG10_2.CPP
// Драйвер иерархии точка, круг, цилиндр
#include <iostream.h>
#include <iomanip.h>
#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"

main ( )
{
    Point point(7, 11);           // Создается объект класса Point
    Circle circle(3.5, 22, 8);    // Создается объект класса Circle
    Cylinder cylinder(10, 3.3, 10, 10); // Создается
                                       // объект класса Cylinder

    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 *arrayOfShapes[3]; // Массив указателей базового класса
    // Задание элементу arrayOfShapes[0] объекта производного
    // класса Point
    // Задание элементу arrayOfShapes[1] объекта производного
    // класса Circle
    // Задание элементу arrayOfShapes[2] объекта производного
    // класса Cylinder
    arrayOfShapes[0] = &point;
    arrayOfShapes[1] = &circle;
    arrayOfShapes[2] = &cylinder;

    // Цикл по arrayOfShapes и печать имени формы, площади и объема
    // каждого объекта, на который указывает элемент
    // (используется динамическое связывание)
    for (int i = 0; i < 3; i++) {
        arrayOfShapes[i]->printShapeName();
        cout << endl;
        arrayOfShapes[i]->print();
        cout << "\nПлощадь = " << arrayOfShapes[i]->area()
            << "\nОбъем = " << arrayOfShapes[i]->volume()
            << endl << endl;
    }

    return 0;
}
```

Рис. 10.2. Драйвер для иерархии точка, круг, цилиндр (часть 8 из 9)

```

Точка: [7, 11]
Круг: [22, 8]; Радиус = 3.50
Цилиндр: [10.00, 10.00]; Радиус = 3.30; Высота = 10.00

Точка:
[7.00, 11.00]
Площадь = 0.00
Объем = 0.00

Круг:
[22.00, 8.00]; Радиус = 3.50
Площадь = 38.48
Объем = 0.00

Цилиндр:
[10.00, 10.00]; Радиус = 3.30; Высота = 10.00
Площадь = 275.77
Объем = 342.12

```

Рис. 10.2. Драйвер для иерархии точка, круг, цилиндр (часть 9 из 9)

Программа драйвер (рис. 10.2, части 8 и 9) начинает свою работу с создания объекта `point` класса `Point`, объекта `circle` класса `Circle` и объекта `cylinder` класса `Cylinder`. Для каждого объекта вызывается функция `printShapeName` и каждый объект выводится с помощью своей перегруженной операции поместить в поток, чтобы показать правильную инициализацию объектов. Затем объявляется массив `arrayOfShapes` типа `Shape *`. Этот массив указателей базового класса используется для того, чтобы указывать на каждый созданный объект производного класса. Сначала элементу массива `arrayOfShapes[0]` присваивается адрес объекта `point`, затем элементу `arrayOfShapes[1]` присваивается адрес объекта `circle`, а элементу `arrayOfShapes[2]` присваивается адрес объекта `cylinder`. После этого выполняется структура `for`, просматривающая массив `arrayOfShapes` и выполняющая на каждой итерации цикла следующие вызовы:

```

arrayOfShapes[i]->printShapeName;
arrayOfShapes[i]->print( )
arrayOfShapes[i]->area( )
arrayOfShapes[i]->volume( )

```

Каждый из приведенных выше вызовов активизирует эти функции для того объекта, на который указывает элемент массива `arrayOfShapes[i]`. Из выходных результатов, приведенных на рис. 10.2, часть 9, следует, что функции вызываются должным образом. Сначала выводится строка "Точка:" и координаты, хранящиеся в объекте `point`; площадь и объем равны 0.00. Затем выводится строка "Круг:", координаты центра круга и радиус, хранящиеся в объекте `circle`; потом вычисляется площадь круга, а его объем равен 0.00. И, наконец, выводится строка "Цилиндр", координаты центра, радиус и высота цилиндра, хранящиеся в объекте `cylinder`; потом вычисляются площадь его поверхности и объем. Все вызовы функций `printShapeName`, `print`, `area` и `volume` реализуются в процессе выполнения программы с помощью динамического связывания.

Резюме

- С помощью виртуальных функций и полиморфизма становится возможным создавать и внедрять системы, которые являются легко расширяемыми. Программы могут быть написаны так, чтобы обрабатывать объекты типов, которых даже еще может быть нет, пока программа находится в стадии разработки.
- Виртуальные функции и полиморфное программирование могут устранить потребность в использовании логики оператора выбора `switch`. Программист может использовать возможности виртуальных функций для автоматического выполнения аналогичной логики, избегая таким образом разного рода ошибок, присущих логике оператора `switch`. Коды, принимающие решения относительно типов объектов и их представления, свидетельствуют о плохом стиле проектирования классов.
- Виртуальная функция объявляется с помощью ключевого слова `virtual`, предшествующего прототипу функции в базовом классе.
- Производные классы могут, если надо, иметь собственные реализации виртуальных функций базового класса, но если они их не имеют, то используются реализации, описанные в базовом классе.
- Когда виртуальная функция вызывается путем обращения к заданному объекту по имени и при этом используется операция доступа к элементу точка, тогда эта ссылка обрабатывается во время компиляции (это называется статическим связыванием) и в качестве вызываемой определяется функция класса данного объекта (или наследуемая этим классом).
- Имеются случаи, в которых полезно определять классы, для которых программист не намерен создавать какие-либо объекты. Такие классы называются абстрактными классами. Поскольку они применяются в качестве базовых классов в процессе наследования, мы обычно будем называть их абстрактными базовыми классами. Объекты абстрактного базового класса не могут быть реализованы.
- Классы, объекты которых могут быть реализованы, называются конкретными классами.
- Класс делается абстрактным путем объявления одной или более его виртуальных функций чисто виртуальными. Чистой виртуальной функцией является такая функция, у которой в ее объявлении тело определено как 0 (инициализатор равен 0).
- Если класс является производным от класса с чистой виртуальной функцией и если эта чистая виртуальная функция не определена в производном классе, то функция остается чистой виртуальной и в производном классе. Тогда производный класс также является абстрактным классом (и не может иметь каких-либо объектов).
- C++ включает такое свойство, как полиморфизм — возможность для объектов разных классов, связанных с помощью наследования, реагировать различным образом при обращении к одной и той же функции-элементу.
- Полиморфизм реализуется с помощью виртуальных функций.

- Если при использовании виртуальной функции запрос осуществляется с помощью указателя базового класса (или ссылки), то C++ выбирает правильную переопределенную функцию в соответствующем производном классе, связанном с данным объектом.
- Благодаря использованию виртуальных функций и полиморфизму, один и тот же вызов функции-элемента может привести к различным действиям в зависимости от типа объекта, принимающего этот вызов.
- Хотя мы не можем создавать объекты абстрактного базового класса, мы *можем* объявить указатели на абстрактный базовый класс. Эти указатели могут быть затем использованы, чтобы предоставить возможность для полиморфного оперирования объектами производных конкретных классов.
- В системы постоянно добавляются новые типы классов. Новые классы встраиваются при помощи динамического связывания, называемого также поздним связыванием. Во время компиляции нет необходимости знать тип объекта, чтобы скомпилировать вызов виртуальной функции. Во время выполнения программы вызов виртуальной функции будет соответствовать функции-элементу вызванного объекта.
- Динамическое связывание позволяет независимым дистрибутерам программного обеспечения распространять свою продукцию, не выдавая фирменных секретов. Распространяемое программное обеспечение может состоять только из заголовочных и объектных файлов. Чтобы не раскрывать секреты программного обеспечения, не должно прикладываться никаких исходных текстов. Разработчики программного обеспечения могут использовать наследование для создания новых производных классов на основе тех классов, которые предоставлены им дистрибутерами программного обеспечения. Программные средства, которые работают с классами, предоставленными дистрибутерами, будут продолжать работать и с производными классами, используя (с помощью динамического связывания) переопределенные виртуальные функции, имеющиеся в этих классах.
- Динамическое связывание требует, чтобы во время выполнения программы вызов виртуальной функции-элемента был бы направлен на вариант виртуальной функции соответствующего класса. Для этого служит таблица виртуальных методов, которая реализуется в виде массива, содержащего указатели на функции. У каждого класса, который содержит виртуальные функции, имеется таблица виртуальных методов. Для каждой виртуальной функции в классе таблица имеет элемент, содержащий указатель на вариант виртуальной функции, используемый в объектах данного класса. Виртуальная функция, используемая в некотором классе, может быть определена в этом классе или прямо или косвенно наследоваться из базового класса, стоящего выше в иерархии.
- Если базовый класс имеет виртуальную функцию-элемент, то производные классы могут переопределить эту функцию, но они могут этого и не делать. Таким образом, производный класс может использовать вариант виртуальной функции-элемента базового класса и это будет отражено в таблице виртуальных методов.

- Каждый объект класса, содержащего виртуальные функции, имеет указатель на таблицу виртуальных методов этого класса, недоступный для программиста. Во время выполнения программы вызовы виртуальных функций осуществляются через разыменование соответствующего указателя в таблице виртуальных методов. Просмотр таблицы виртуальных методов и операция разыменования указателя требуют минимальных затрат времени выполнения, обычно значительно меньших, чем самые лучшие коды клиентов.
- Если у класса имеются виртуальные функции, предусматривайте создание виртуального деструктора. Это приведет к тому, что все деструкторы производных классов станут виртуальными, даже если они имеют имена, отличные от имени деструктора базового класса. В этом случае, если объект в иерархии уничтожен явным использованием операции `delete`, примененной к указателю базового класса на объект производного класса, то будет вызван деструктор соответствующего класса.

Терминология

абстрактный базовый класс	полиморфизм
абстрактный класс	преобразование указателя
виртуальная функция	производного класса к указателю
виртуальная функция базового	базового класса
класса	производный класс
виртуальный деструктор	прямой базовый класс
динамическое связывание	раннее связывание
иерархия классов	расширяемость
исключение операторов <code>switch</code>	ссылка на абстрактный класс
ключевое слово <code>virtual</code>	ссылка на базовый класс
конструктор производного класса	ссылка на производный класс
косвенный базовый класс	статическое связывание
логика оператора <code>switch</code>	таблица виртуальных методов <code>vtable</code>
наследование	указатель на абстрактный класс
неявное преобразование указателя	указатель на базовый класс
переопределенная виртуальная	указатель на производный класс
функция	чистая виртуальная функция (<code>=0</code>)
повторное использование кодов	явное преобразование указателей
позднее связывание	

Типичные ошибки программирования

- 10.1. Если в производном классе переопределяется виртуальная функция базового класса и данная функция не имеет тот же тип возвращаемого значения и ту же сигнатуру, что и соответствующая функция базового класса, то возникает синтаксическая ошибка.
- 10.2. Попытка создать объект абстрактного класса (т.е. класса, который содержит хотя бы одну чистую виртуальную функцию) является синтаксической ошибкой.
- 10.3. Объявление конструктора виртуальной функцией. Конструкторы не могут быть виртуальными.

Хороший стиль программирования

- 10.1. Несмотря на то, что некоторые функции могут быть неявно виртуальными, поскольку они объявлены такими на более высоком уровне иерархии, некоторые программисты предпочитают явно объявлять функции виртуальными на каждом уровне иерархии, чтобы обеспечить ясность программы.
- 10.2. Если у класса имеются виртуальные функции, предусматривайте создание виртуального деструктора, даже если он не требуется этому классу. Классы, производные от данного класса, могут содержать деструкторы, которые должны вызываться соответствующим образом.

Советы по повышению эффективности

- 10.1. Полиморфизм, реализуемый с помощью виртуальных функций и динамического связывания, очень эффективен. Программисты могут использовать это средство при ничтожном влиянии на производительность системы.
- 10.2. Виртуальные функции и динамическое связывание позволяют использовать полиморфное программирование как альтернативу логике оператора `switch`. Оптимизирующие компиляторы C++ обычно генерирует код, который исполняется по крайней мере так же эффективно, как код, построенный вручную на основе логики оператора выбора `switch`.

Замечания по технике программирования

- 10.1. Интересным следствием использования виртуальных функций и полиморфизма является то, что программы приобретают более простой вид. Они включают меньше логических ветвлений и больше простого последовательного кода. Это упрощение облегчает тестирование, отладку и сопровождение программ.
- 10.2. Если функция однажды объявлена виртуальной, то она остается виртуальной на любом более низком уровне иерархической структуры.
- 10.3. Если в производном классе решено не описывать виртуальную функцию, то производный класс непосредственно наследует описание виртуальной функции из базового класса.
- 10.4. Переопределенная виртуальная функция должна иметь тот же самый тип возвращаемого значения и ту же сигнатуру, что и виртуальная функция базового класса.
- 10.5. Если класс является производным от класса с чистой виртуальной функцией и если эта чистая виртуальная функция не описана в производном классе, тогда функция остается чистой виртуальной и в производном классе. Следовательно, такой производный класс также является абстрактным классом.

- 10.6. Благодаря виртуальным функциям и полиморфизму программист может управлять общими свойствами объектов, предоставляя возможность программной среде во время выполнения программы самой «заботиться» о специфике объектов. Программист может управлять широким спектром объектов, даже не зная об их типах, причем управление будет автоматически учитывать специфику этих объектов.
- 10.7. Полиморфизм способствует расширяемости: программное обеспечение, использующее полиморфный механизм, пишется независимо от типов объектов, которым отправляются сообщения. Таким образом, новые типы объектов, которые должны реагировать на соответствующие сообщения, могут включаться в такую систему без модификации ее основы. За исключением кода пользователя, который создает новые объекты, программа не потребует перекомпиляции.
- 10.8. Абстрактный класс определяет интерфейс для разных типов элементов иерархии классов. Абстрактный класс включает чистые виртуальные функции, которые будут определены в производных классах. Все функции в иерархии могут применять один и тот же интерфейс, используя полиморфизм.

Упражнения для самопроверки

- 10.1. Заполните пробелы в следующих утверждениях:
- Использование наследования и полиморфизма помогает исключить логику _____ .
 - Чистая виртуальная функция задается с помощью размещения _____ в конце ее прототипа в определении класса.
 - Если класс содержит хотя бы одну чистую виртуальную функцию, он является _____ .
 - Вызов функции, обрабатываемый во время компиляции, называется _____ связыванием.
 - Вызов функции, обрабатываемый во время выполнения программы, называется _____ связыванием.

Ответы на упражнения для самопроверки

- 10.1. а) структуры switch. б) = 0. в) абстрактным базовым классом. г) статическим. е) динамическим.

Упражнения

- 10.2. Что такое виртуальные функции? Опишите случаи, в которых использование виртуальных функций было бы оправдано.
- 10.3. Учитывая, что конструкторы не могут быть виртуальными, опишите схему, по которой вы могли бы достичь аналогичного эффекта.

- 10.4. Как понимать, что полиморфизм позволяет вам осуществлять «общий подход» к программированию, «не вдаваясь в конкретику» классов. Приведите основные преимущества «общего подхода» к программированию.
- 10.5. Рассмотрите проблемы программирования, связанные с использованием логики оператора выбора `switch`. Объясните, почему полиморфизм является мощной альтернативой использованию логики оператора `switch`.
- 10.6. Выявите различия между статическим и динамическим связыванием. Объясните использование виртуальных функций и таблиц виртуальных методов при динамическом связывании.
- 10.7. Выявите различия между наследованием интерфейса и реализаций. Чем отличаются иерархические структуры, разработанные для наследования интерфейса, от иерархических структур, разработанных для наследования реализаций?
- 10.8. Выявите различия между виртуальными функциями и чистыми виртуальными функциями.
- 10.9. (*Правильно или ошибочно*) Все виртуальные функции в абстрактном базовом классе должны быть объявлены как чистые виртуальные функции.
- 10.10. Предложите один или более дополнительных уровней абстрактных базовых классов для иерархии `Shape`, рассмотренной в данной главе (первый уровень — `Shape`, второй — классы `TwoDimensionalShape` и `TwoDimensionalShape`).
- 10.11. Каким образом полиморфизм способствует расширяемости?
- 10.12. Создайте эскиз моделирующей программы, которая выводит сложные графические объекты. Объясните, почему полиморфное программное обеспечение будет особенно эффективно при решении задач такого рода.
- 10.13. Создайте базовый графический пакет. Используйте иерархию наследования класса `Shape` из главы 9. Ограничтесь двумерными графическими объектами, такими, как квадраты, прямоугольники, треугольники и окружности. Обеспечьте интерактивное взаимодействие с пользователем. Предоставьте возможность пользователю задавать позицию, размер, форму и заполняющий символ, используемый при изображении каждой формы. Пользователь должен иметь возможность задавать несколько объектов одной и той же формы. Когда вы создаете каждую форму, помешайте указатель `Shape *` на этот новый объект в массив. Каждый класс имеет собственную функцию-элемент `draw`. Напишите полиморфный экранный администратор, который будет проводить поиск в массиве (желательно с помощью итератора) и посыпать сообщения «нарисовать» (обращение к функции-элементу `draw`) каждому объекту в массиве для формирования экранного образа. Перерисовывайте весь экран каждый раз, когда пользователь задает очередную форму.

- 10.14.** Модифицируйте систему расчета заработной платы на рис 10.1, добавив скрытые данные-элементы `birthDate` (дата рождения, объект `Date`) и `DepartmentCode` (код отдела типа `int`) в класс `Employee`. Предполагайте, что расчет заработной платы производится один раз в месяц. Пусть ваша программа производит расчет заработной платы каждого служащего (с помощью полиморфизма) и добавляет премию в 100 долларов к его зарплате, если на месяц оплаты выпадает день его рождения.
- 10.15.** В упражнении 9.14 вы разработали иерархию классов форм `Shape` и описали классы этой иерархии. Модифицируйте иерархию так, чтобы класс `Shape` был абстрактным базовым классом, содержащим интерфейс иерархии. Создайте классы `TwoDimensionalShape` и `ThreeDimensionalShape`, производные от класса `Shape`. Эти классы тоже должны быть абстрактными. Используйте виртуальную функцию `print` для вывода типа и размера объекта каждого класса. Добавьте также виртуальные функции `area` и `volume`, которые выполняли бы вычисления для объектов каждого конкретного класса в иерархии. Напишите программу драйвер, которая проверяла бы созданную иерархию.

11

Потоки ввода-вывода в C++



Ц е л и

- Понять, как использовать объектно-ориентированные потоки ввода-вывода в C++.
- Научиться форматировать вводимые и выводимые данные.
- Понять иерархию классов потоков ввода-вывода.
- Понять, как вводить и выводить объекты типов, определенных пользователем.
- Научиться создавать определенные пользователем манипуляторы потока.
- Научиться определять, успешно или нет выполнилась операция ввода-вывода.
- Научиться связывать выходной поток с входным.

План

11.1. Введение

11.2. Потоки

11.2.1. Заголовочные файлы библиотеки ***iostream***

11.2.2. Классы и объекты потоков ввода-вывода

11.3. Вывод потоков

11.3.1. Операция поместить в поток

11.3.2. Сцепление операций поместить в поток и взять из потока

11.3.3. Вывод переменных типа ***char ****

11.3.4. Вывод символов с помощью функции-элемента ***put***; сцепленные выводы

11.4. Ввод потоков

11.4.1. Операция взять из потока

11.4.2. Функции-элементы ***get*** и ***getline***

11.4.3. Другие функции-элементы класса ***istream*** (***peek***, ***putback*** и ***ignore***)

11.4.4. Сохранение типов данных при вводе-выводе

11.5. Неформатированный ввод-вывод с использованием ***read***, ***gcount*** и ***write***

11.6. Манипуляторы потоков

11.6.1. Манипуляторы потоков ***dec***, ***oct***, ***hex*** и ***setbase***, задающие основание чисел

11.6.2. Точность чисел с плавающей запятой (***precision***, ***setprecision***)

11.6.3. Ширина поля (***setw*** и ***width***)

11.6.4. Манипуляторы, определяемые пользователем

11.7. Состояния формата потоков

11.7.1. Флаги состояний формата

11.7.2. Нулевые младшие разряды и десятичные точки (***ios::showpoint***)

11.7.3 Выравнивание (***ios::left***, ***ios::right*** и ***ios::internal***)

11.7.4. Заполнение (***fill*** и ***setfill***)

11.7.5. Основание системы счисления (***ios::dec***, ***ios::oct***, ***ios::hex***, ***ios::showbase***)

11.7.6. Числа с плавающей запятой; экспоненциальный формат (***ios::scientific***, ***ios::fixed***)

11.7.7. Управление выводом в нижнем и верхнем регистрах (***ios::uppercase***)

11.7.8. Установка и сброс флагов формата (***flags***, ***setiosflags*** и ***resetiosflags***)

11.8. Состояния потока ошибок

11.9. Ввод-вывод определенных пользователем типов данных

11.10. Связывание выходного потока с входным

Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

11.1. Введение

Стандартные библиотеки в языке C++ имеют расширенный набор средств ввода-вывода. В этой главе рассматривается комплекс средств, достаточный для выполнения большинства общепринятых операций ввода-вывода, и приводится обзор остальных возможностей. Некоторые средства, приведенные здесь, были рассмотрены в книге раньше, но эта глава позволяет провести более полное и упорядоченное рассмотрение возможностей ввода-вывода в C++.

Многие из описанных здесь средств ввода-вывода являются объектно-ориентированными. Читателю будет интересно увидеть, как они реализуются. Стиль ввода-вывода, приведенный в этой главе, позволяет использовать и другие свойства языка C++, такие, как ссылки, перегрузка функций и перегрузка операций.

Как мы увидим позже, в языке C++ предусмотрены средства *сохранения типов данных* при вводе-выводе. Каждая операция ввода-вывода осуществляется способом, чувствительным к типу данных. Если для какого-то типа данныхенным образом была определена функция ввода-вывода, то именно она автоматически вызывается для обработки этого типа данных. Если нет соответствия между типом реальных данных и функцией обработки этого типа данных, то компилятор обнаружит ошибку. Таким образом, несуществующие данные не могут проникнуть в систему (как может случиться в C, в котором есть лазейки, приводящие к возникновению неуловимых и странных ошибок).

Пользователи могут задавать ввод-вывод типов, определяемых пользователем, так же, как и стандартных типов. Такая *расширяемость* является самым ценным свойством языка C++.

Хороший стиль программирования 11.1

Используйте исключительно возможности языка C++ для организации ввода-вывода в программах, написанных на C++, хотя в этих программах доступен и стиль программирования языка C.

Замечание по технике программирования 11.1

Стиль программирования на языке C++ – это стиль, в котором предусмотрены средства сохранения типов данных

Замечание по технике программирования 11.2

Язык C++ предоставляет возможность для стандартной обработки ввода-вывода встроенных типов данных и типов данных, определенных пользователем. Это свойство облегчает разработку программного обеспечения вообще и повторное его использование в частности.

11.2. Потоки

В языке C++ производится ввод-вывод *потоков* байтов. Поток — это просто последовательность байтов. В операциях ввода байты пересыпаются от устройства (например, от клавиатуры, дисковода или соединения сети) в оперативную память. При выводе байты пересыпаются из оперативной памяти на устройство (например, на экран дисплея, принтер, дисковод или в соединение сети).

Приложение связывает значение элемента данных с байтами данных. Байты данных могут представляться в виде символов в коде ASCII, в виде внутреннего формата сырых данных, в виде графического изображения, отцифрованной речи, цифрового видеоизображения или в виде любой другой информации, которая может потребоваться приложению.

Механизм ввода-вывода заключается в пересылке байтов данных от устройств в оперативную память и обратно эффективным и надежным способом. Такая пересылка часто включает механические перемещения, например, вращение диска и ленты или нажатие клавиш клавиатуры. Время, затрачиваемое на такие перемещения, обычно значительно превышает время, в течение которого процессор обрабатывает данные. Таким образом, операции ввода-вывода требуют тщательного планирования для достижения максимальной эффективности. Вопросы, подобные этому, глубоко рассматриваются в руководствах по операционным системам (De90).

Язык C++ предоставляет возможности для ввода-вывода как на «низком», так и на «высоком» уровнях. Ввод-вывод на низком уровне (т.е. *неформатированный ввод-вывод*) обычно сводится к тому, что некоторое число байтов данных просто следует переслать от устройства в память или из памяти в устройство. При такой пересылке каждый байт является самостоятельным элементом данных. Передача на низком уровне позволяет осуществлять пересылку больших по объему потоков ввода-вывода с высокой скоростью, но такая передача обычно неудобна для программиста.

Программисты предпочитают иметь дело с представлением операций ввода-вывода на высоком уровне, т.е. с *форматированным вводом-выводом*, при котором байты группируются в такие значащие элементы данных, как, например, целые числа, числа с плавающей запятой, символы, строки, а также данные типов, определенных пользователем. Такие ориентированные на типы возможности более удобны для большинства операций ввода-вывода, кроме обработки файлов очень большого объема.

Совет по повышению эффективности 11.1

Используйте неформатированный ввод-вывод для достижения максимальной эффективности при обработке файлов большого объема.

11.2.1. Заголовочные файлы библиотеки *iostream*

Библиотека потоков *iostream* предоставляет сотни возможностей для выполнения операций ввода-вывода. Интерфейс библиотеки разбит на несколько заголовочных файлов.

Большая часть программ на языке C++ включает заголовочный файл *<iostream.h>*, который содержит основные сведения, необходимые для всех опера-

ций с потоками ввода-вывода. Заголовочный файл `<iostream.h>` включает объекты `cin`, `cout`, `cerr` и `clog`, которые соответствуют стандартному потоку ввода, стандартному потоку вывода, небуферизованному и буферизованному стандартным потокам вывода сообщений об ошибках. Предусмотрены возможности как для форматированного ввода-вывода, так и для неформатированного.

Заголовочный файл `<iomanip.h>` содержит информацию, полезную для обработки форматированного ввода-вывода при помощи так называемых *параметризованных манипуляторов потока*.

Заголовочный файл `<fstream.h>` содержит важную информацию для проведения операций с файлами, обработка которых осуществляется под контролем пользователя.

Заголовочный файл `<strstream.h>` содержит информацию, важную для выполнения *форматированного ввода-вывода в память*. Это похоже на обработку файлов, но операции ввода-вывода проводятся с символьными массивами, а не с файлами.

Заголовочный файл `<stdiostream.h>` включает нужные сведения для программ, использующих для выполнения операций ввода-вывода сочетание стилей языков С и С++. При создании новых программах следует избегать для операций ввода-вывода стиля языка С. Но для программистов, которым необходимо модифицировать уже существующие программы, написанные на С, эта предоставляемая в С++ возможность сочетания двух стилей окажется полезной.

Программы на С++ могут содержать также другие библиотеки, связанные с вводом-выводом, в которых предусмотрены, например, такие специфические для системы возможности, как управление специализированными устройствами для ввода-вывода аудио- и видеоданных.

11.2.2. Классы и объекты потоков ввода-вывода

Библиотека `iostream` содержит много классов для обработки широкого спектра операций ввода-вывода. Класс `istream` поддерживает операции по вводу потоков. Класс `ostream` поддерживает операции по выводу потоков. Класс `iostream` поддерживает оба типа операций: и ввод и вывод потоков.

Класс `istream` и класс `ostream` являются производными классами прямого наследования базового класса `ios`. Класс `iostream` является производным классом множественного наследования классов `istream` и `ostream`. Структура связей наследования представлена на рис. 11.1.

Перегрузка операций обеспечивает удобную запись операций ввода-вывода. Операция сдвига влево (`<<`) перегружена для обозначения вывода в поток и называется *операцией поместить в поток*. Операция сдвига вправо (`>>`) перегружена для обозначения ввода потока и называется *операцией взять из потока*. Эти операции применяются к объектам стандартных потоков `cin`, `cout`, `cerr` и `clog` и обычно используются также с объектами потоков, тип которых определен пользователем.

Объект стандартного потока ввода `cin` класса `istream`, как принято говорить, «привязан» (или «присоединен») к стандартному устройству ввода, обычно к клавиатуре. Операция *взять из потока*, показанная в приведенном ниже операторе, означает, что значение целой переменной `grade` (если полагать, что переменная `grade` объявлена как целая типа `int`) должно быть введено из объекта `cin` в память:

```
cin >> grade;
```

Заметим, что операция взять из потока является «достаточно интеллектуальной», чтобы определить тип используемых данных. Если переменная *grade* была должным образом объявлена, то не требуется никакой дополнительной информации для использования операции взять из потока (между прочим, в случае использования языка С такая информация требуется).

Объект стандартного потока вывода *cout* класса *ostream*, как принято говорить, «привязан» к стандартному устройству вывода, обычно к экрану дисплея. Операция поместить в поток, показанная в приведенном ниже операторе, означает, что значение целой переменной *grade* должно быть выведено из памяти на стандартное устройство вывода:

```
cout << grade;
```

Заметим, что операция поместить в поток также является «достаточно умной», чтобы определить тип переменной *grade* (полагая, что она должным образом объявлена); поэтому для использования операции поместить в поток никакой дополнительной информации не требуется.

Объект *cerr* класса *ostream* «привязан» к стандартному устройству вывода сообщений об ошибках. Выводимые потоки данных для объекта *cerr* являются небуферизованными. Это означает, что каждая операция поместить в *cerr* приводит к мгновенному появлению выводимых сообщений об ошибках; в этих случаях пользователь своевременно и должным образом информируется о неполадках в системе.

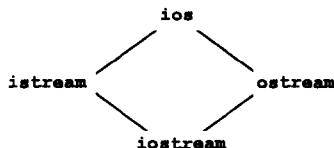


Рис. 11.1. Часть иерархии классов потоков ввода-вывода

Объект *clog* класса *ostream* «привязан» тоже к стандартному устройству вывода сообщений об ошибках. Выводимые потоки данных для объекта *clog* являются буферизованными. Это означает, что каждая операция поместить в *clog* может приводить к тому, что вывод хранится в буфере до тех пор, пока буфер полностью не заполнится или пока содержимое буфера не будет выведено принудительно.

При обработке файлов в C++ используются следующие классы:

- класс *ifstream*, выполняющий операции ввода из файлов;
- класс *ofstream*, выполняющий операции вывода в файлы;
- класс *fstream*, предназначенный для операций ввода-вывода файлов.

Класс *ifstream* наследует классу *istream*, класс *ofstream* наследует классу *ostream*, а класс *fstream* — классу *iostream*. Структура связей наследования классов потоков ввода-вывода приведена на рис. 11.2. В большинстве систем в полной иерархии классов потоков ввода-вывода поддерживается еще множество других классов, но и классы, приведенные выше, предоставляют широкие возможности, достаточные большинству программистов. Более подробные сведения об обработке файлов можно получить из справочного руководства по библиотекам классов C++ вашей системы.

11.3. ВЫВОД ПОТОКОВ

Класс `ostream` в C++ обеспечивает возможность реализации форматированного и неформатированного вывода потоков. Эта возможность позволяет осуществлять вывод следующих данных: вывод данных стандартных типов с помощью операции поместить в поток; вывод символов с помощью функции-элемента `put`; неформатированный вывод с помощью функции-элемента `write` (см. раздел 11.5); вывод целых чисел в десятичном, восьмеричном и шестнадцатеричном форматах (см. раздел 11.6.1); вывод значений с плавающей запятой с различной точностью (см. раздел 11.6.2), с указанием по выводу десятичной точки (см. раздел 11.7.2), в экспоненциальном формате или в формате с фиксированной точкой (см. раздел 11.7.6); вывод данных с выравниванием относительно какой-либо границы поля указанной ширины (см. раздел 11.7.3); вывод данных с полями, заполненными заданными символами (см. раздел 11.7.7); вывод буквами в верхнем регистре в экспоненциальном формате и при выводе шестнадцатеричных чисел (см. раздел 11.7.7).

11.3.1. Операция поместить в поток

Вывод в потоки может быть выполнен с помощью операции `поместить в поток`, т.е. перегруженной операции `<<`. Операция `<<` перегружена для вывода элементов данных встроенных типов, для вывода строк и вывода значений указателей. В разделе 11.9 показано, как перегрузить операцию `<<` для вывода данных типов, определенных пользователем. В программе на рис. 11.3 показан вывод строки, использующий одну операцию `поместить в поток`. Пример многократного использования операции `поместить в поток` приведен на рис. 11.4. Выполнение этой программы дает тот же результат, что и выполнение предыдущей программы.

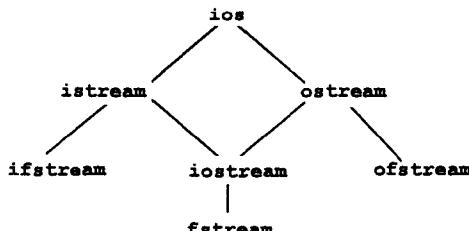


Рис. 11.2. Часть иерархии классов потоков ввода-вывода с ключевыми классами обработки файлов

```

// fig11_03.cpp
// Вывод строки операцией поместить в поток

# include <iostream.h>

main()
{
    cout << "Добро пожаловать в мир C++ !\n";
    return 0;
}
  
```

Добро пожаловать в мир C++ !

Рис. 11.3. Вывод строки с использованием операции `поместить в поток`

```
// fig11_04.cpp
// Вывод строки двумя операциями поместить в поток

#include <iostream.h>

main()
{
    cout << "Добро пожаловать в ";
    cout << "мир C++ !\n";

    return 0;
}
```

Добро пожаловать в мир C++ !

Рис. 11.4. Вывод строки с помощью двух операций поместить в поток

Переход на новую строку, осуществленный в этих программах с помощью управляющей последовательности `\n`, можно осуществить и с помощью *манипулятора потока endl* (end line — конец строки), как показано на рис. 11.5. Манипулятор потока `endl` вызывает переход на новую строку и кроме того приводит к сбросу буфера вывода (т.е. заставляет буфер немедленно вывести данные, даже если он полностью не заполнен). Сброс буфера вывода может быть также выполнен оператором:

```
cout << flush;
```

Манипуляторы потоков детально обсуждаются в разделе 11.6. Пример вывода значения выражения показан на рис. 11.6.

Хороший стиль программирования 11.2

При выводе выражений помещайте их в круглых скобках, чтобы предотвратить возникновение каких-либо проблем, связанных с последовательностью выполнения операций, вычисляющих значения в выражениях, и операции `<<`.

```
// fig11_05.cpp
// Использование манипулятора потока endl
# include <iostream.h>

main()
{
    cout << "Добро пожаловать в ";
    cout << "мир C++ !";
    cout << endl;           // манипулятор потока конец строки

    return 0;
}
```

Добро пожаловать в мир C++ !

Рис. 11.5. Использование манипулятора потока `endl`

```
// fig11_06.cpp
// Вывод значения выражения
#include <iostream.h>
main()
{
    cout << "47 плюс 53 равняется ";
    cout << (47 + 53);           // выражение
    cout << endl;

    return 0;
}
```

47 плюс 53 равняется 100

Рис. 11.6. Вывод значения выражения

11.3.2. Сцепление операций поместить в поток и взять из потока

Каждая из перегруженных операций `<<` и `>>` может быть использована в *сцепленной форме*, как показано на рис. 11. 7.

Многократные операции поместить в поток (см. рис. 11.7) выполняются в той последовательности, в которой они записаны :

```
((cout << "47 плюс 53 равняется ") << (47 + 53)) << endl);
```

т.е. операция поместить в поток `<<` имеет ассоциативность слева направо. Такой способ сцепления операций поместить в поток возможен, поскольку перегруженная операция `<<` возвращает ссылку на объект своего левого операнда, т.е. на объект `cout`. Таким образом, самое левое выражение в скобках

```
(cout << "47 плюс 53 равно")
```

выводит заданную строку символов и возвращает ссылку на `cout`. Это приводит к тому, что среднее выражение в скобках должно выполняться как

```
(cout << (47 + 53))
```

выводить целое значение 100 и возвращать ссылку на `cout`. Затем выполняется самое правое выражение в скобках как

```
cout << endl
```

которое переводит строку, сбрасывает `cout` и возвращает ссылку на `cout`. Эта ссылка в данном случае не используется.

```
// fig11_07.cpp
// Сцепление перегруженных операций поместить в поток <<.
#include <iostream.h>

main()
{
    cout << "47 плюс 53 равняется " << (47 + 53) << endl;

    return 0;
}
```

47 плюс 53 равняется 100

Рис. 11.7. Сцепление перегруженных операций поместить в поток `<<`

11.3.3. Вывод переменных типа `char *`

При написании программы в стиле языка С при вводе-выводе программист должен давать информацию о типе данных. В C++ типы данных определяются автоматически и это является замечательным достижением по сравнению с языком С. Но иногда это вызывает сложности. Например, мы знаем, что символьная строка имеет тип `char *`. Допустим, что нам необходимо напечатать значения этого указателя, т.е. адрес в памяти первого символа этой строки. Но операция `<<` перегружена для печати данных типа `char *` как строки с нулевым завершающим символом. Решение этой задачи заключается в том, чтобы привести тип этого указателя к типу `void *` (это следует делать и для любого указателя на переменную, если программист хочет вывести ее адрес). Программа на рис. 11.8 показывает печать переменной `char *` и как строки, и как адреса. Обратите внимание, что адрес печатается как шестнадцатеричное число (по основанию 16). Шестнадцатеричные числа в C++ начинаются с `0x` или `OX`. Более подробные сведения о представлении чисел в различных форматах вы найдете в разделах 11.6.1, 11.7.4, 11.7.5 и 11.7.7.

```
// fig.11_08.cpp
// Печать адреса, хранимого в переменной типа char *
#include <iostream.h>

main()
{
    char *string = "проверка";

    cout << "Значение строки равно " << string
        << "\nЗначение адреса строки (void *) string равно "
        << (void *) string << endl;
}
```

```
Значение строки равно проверка
Значение адреса строки (void *) string равно 0x00aa
```

Рис. 11.8. Печать адреса, который хранится в переменной типа `char *`

11.3.4. Вывод символов с помощью функции-элемента `put`; сцепленные выводы

Функция-элемент `put` выводит одиночный символ, как, например, в следующем операторе:

```
cout.put('A');
```

который отображает на экране А. Вызовы функции `put` могут быть сцепленные, как, например, в следующем операторе:

```
cout.put('A').put('\n');
```

Этот оператор выведет на экран букву А, а затем выведет символ новой строки. Как и в случае `<<` предыдущий оператор выполняется таким же способом, поскольку оператор точка (.) имеет ассоциативность слева направо, а функция-элемент `put` возвращает ссылку на объект, из которого функция

`put` была вызвана. Функция `put` может также вызываться с помощью выражения, имеющего значение кода ASCII символа; например, выражение `cout.put(65)` также выводит букву А.

11.4. Ввод потоков

Рассмотрим теперь операцию ввода потока. Она может быть выполнена с помощью операции взять из потока, т.е. перегруженной операции `>>`. Эта операция обычно игнорирует во входном потоке так называемые *символы разделители или пробельные символы* (пробелы, знаки табуляции, знак новой строки). Позже мы увидим, каким образом это можно изменить. Операция взять из потока возвращает нулевое значение (`false`), когда встречает в потоке признак конца файла; в противном случае операция взять из потока возвращает ссылку на объект, с помощью которого она вызывалась. Каждый поток содержит набор *битов состояния*, используемых для управления состоянием потока (форматированием, установкой ошибок потока и т.д.). Операция взять из потока приводит к установке бита `failbit` при вводе данных неправильного типа и приводит к установке бита `badbit` при неуспешном завершении операции. В дальнейшем мы увидим, каким образом надо проверять эти биты после операции ввода-вывода. В разделах 11.7 и 11.8 использование битов состояния рассматривается более детально.

11.4.1. Операция взять из потока

Для того, чтобы прочитать два целых числа, используем объект `cin` и перегруженную операцию взять из потока `>>`, как показано на рис. 11.9. Обратите внимание, что операции взять из потока можно сцеплять.

```
// fog11_09.cpp
// Вычисление суммы двух целых чисел, которые
// вводятся с клавиатуры с помощью cin
// и операции взять из потока
#include <iostream.h>

main()
{
    int x, y;

    cout << "Введите два целых числа: ";
    cin >> x >> y;
    cout << "Сумма чисел " << x << " и " << y << " равна "
        << (x + y) << endl;

    return 0;
}
```

Введите два целых числа: 30 92
Сумма чисел 30 и 92 равна 122

Рис. 11.9. Вычисление суммы двух целых чисел, вводимых с клавиатуры с помощью `cin` и операции взять из потока

Относительно высокий приоритет операций `>>` и `<<` может привести к возникновению проблем. Например, программа, приведенная на рис. 11.10, не может быть скомпилирована должным образом, если условное выражение не выделено с помощью круглых скобок. Читателю предоставляется возможность самому проверить это.

```
// fig.11_10.cpp
// Демонстрация проблемы приоритета операций
// Необходимо заключать условное выражение в круглые скобки
#include <iostream.h>

main()
{
    int x, y;
    cout << "Введите два целых числа ";
    cin >> x >> y;
    cout << x << (x == y ? " " : " не ")
        << "равно " << y << endl;

    return 0;
}

Введите два целых числа 7 5
7 не равно 5

Введите два целых числа 8 8
8 равно 8
```

Рис. 11.10. Устранение проблем, связанных с приоритетами операции поместить в поток и условной операции

Типичная ошибка программирования 11.1

Попытка прочитать данные из потока класса **ostream** (или из любого другого выходного потока).

Типичная ошибка программирования 1.2

Попытка записать данные в поток класса **istream** (или в любой другой входной поток).

Типичная ошибка программирования 1.3

Отсутствие круглых скобок, устанавливающих последовательность выполнения операций, при использовании операций с относительно высоким приоритетом вставить в поток `<<` и взять из потока `>>`.

Одним из наиболее распространенных способов последовательного ввода значений является использование операции `взять из потока` в условии продолжения цикла структуры `while`. Операция `взять из потока` возвращает `false (0)`, когда встречается признак конца файла. Рассмотрим программу на рис. 11.11, которая находит самую высокую оценку на экзамене. Пусть число оценок неизвестно и пользователь должен вводить признак конца файла для

того, чтобы показать, что все оценки уже введены. Условие продолжения цикла вида (`cin >> grade`) в структуре `while`, становится равным 0 (т.е. ложным), когда пользователь вводит признак конца файла.

```
// fig11_11.cpp
// Операция взять из потока, возвращающая
// ложь при вводе признака конца файла
#include <iostream.h>

main()
{
    int grade, highestGrade = -1;

    cout << "Введите оценку (в конце введите признак конца файла): ";
    while (cin >> grade) {
        if (grade > highestGrade)
            highestGrade = grade;

        cout << "Введите оценку (в конце введите признак
                 конца файла): ";
    }

    cout << endl << "Наивысшая оценка равна: " << highestGrade
        << endl;
    return 0;
}
```

```
Введите оценку (в конце введите признак конца файла): 67
Введите оценку (в конце введите признак конца файла): 87
Введите оценку (в конце введите признак конца файла): 73
Введите оценку (в конце введите признак конца файла): 95
Введите оценку (в конце введите признак конца файла): 34
Введите оценку (в конце введите признак конца файла): 99
Введите оценку (в конце введите признак конца файла): ^z
```

```
Наивысшая оценка равна: 99
```

Рис. 11.11. Операция взять из потока, возвращающая ложь при вводе признака конца файла

В программе на рис. 11.11 выражение `cin >> grade` может быть использовано в качестве условия, поскольку базовый класс `ios` (наследником которого является класс `istream`) обеспечивает перегруженную операцию приведения типа, которая преобразует поток в указатель типа `void *`. Значение указателя является нулевым, если произошла ошибка при попытке чтения значения или если был введен признак конца файла. Компилятор способен неявно использовать операцию приведения типа к `void *`.

11.4.2. Функции-элементы `get` и `getline`

Функция-элемент `get` без аргументов вводит одиночный символ из указанного потока (даже, если это символ разделитель) и возвращает этот символ в качестве значения вызова функции. Этот вариант функции `get` возвращает `EOF`, когда в потоке встречается признак конца файла.

Программа на рис. 11.12 демонстрирует использование функций-элементов `eof` и `get` для ввода из входного потока `cin` и использование функции-элемента `put` для вывода в выходной поток `cout`. Сначала программа печатает значение `cin.eof()`, т.е. 0 (ложь), чтобы показать, что конец файла в `cin` не достигнут. Пользователь вводит строку текста, завершающуюся признаком конца файла (`<ctrl>-z` с последующим `<return>` в IBM-совместимых операционных системах, `<ctrl>-d` — на компьютерах с ОС UNIX и Macintosh). Программа читает каждый символ и выводит его в `cout`, используя функцию-элемент `print`. Когда появляется признак конца файла, цикл `while` завершается и снова печатается значение `cin.eof()`, равное теперь 1 (истина), чтобы показать, что в `cin` достигнут конец файла. Заметим, что программа использует вариант функции-элемента `get` класса `istream`, который не принимает никаких аргументов и возвращает введенный символ.

```
// fig11_12.cpp
// Использование функций-элементов get, put и eof.
#include <iostream.h>

main()
{
    int c;

    cout << "До ввода cin.eof() равняется " << cin.eof() << endl
        << "Введите предложение, завершающееся признаком конца файла:" 
        << endl;

    while ( ( c = cin.get() ) != EOF)
        cout.put(c);

    cout << endl << "EOF в этой системе равняется " << c << endl;
    cout << "После ввода cin.eof() равняется " << cin.eof()
        << endl;
    return 0;
}
```

До ввода `cin.eof()` равняется 0
 Введите предложение, завершающееся признаком конца файла:
 Проверка функций-элементов `get` и `put`
 Проверка функций-элементов `get` и `put`
`EOF` в этой системе равняется -1
 После ввода `cin.eof()` равняется 1

Рис. 11.12. Использование функций-элементов `get`, `put` и `eof`

Другой вариант функции-элемента `get` с символьным аргументом вводит очередной символ из входного потока (даже, если это символ разделитель) и сохраняет его в символьном аргументе. Этот вариант функции `get` возвращает ложь, когда встречается признак конца файла; в остальных случаях этот вариант функции `get` возвращает ссылку на тот объект класса `istream`, для которого вызывалась функция-элемент `get`.

Третий вариант функции-элемента `get` принимает три параметра: символьный массив, максимальное число символов и ограничитель (по умолчанию значение '`\n`'). Этот вариант читает символы из входного потока до тех пор, пока не достигается число символов, на 1 меньше указанного макси-

малого числа, или пока не считывается ограничитель. Затем для завершения введенной строки в символьный массив, используемый в качестве буфера программы, помещается нулевой символ. Ограничитель в символьный массив не помещается, а остается во входном потоке (он будет следующим считываемым символом). Таким образом, результатом второго подряд использования функции `get` является пустая строка, если только ограничитель не удалить из входного потока. Программа на рис. 11.13 сравнивает ввод, использующий для `cin` операцию взять из потока (которая читает символы до тех пор, пока не встречается символ разделитель), и ввод с помощью `cin.get`. Обратите внимание, что в обращении к `cin.get` не задается символ ограничитель, так что по умолчанию используется '\n'.

```
// fig11_13. cpp
// Сопоставление ввода строки с помощью cin и cin.get.
#include <iostream.h>

const int SIZE = 80;

main()
{
    char buffer1[SIZE], buffer2[SIZE];

    cout << "Введите предложение:" << endl;
    cin >> buffer1;
    cout << endl << "Из cin прочитана строка:" << endl
        << buffer1 << endl << endl;

    cin.get(buffer2, SIZE);
    cout << "Строка, прочитанная с помощью cin.get:" << endl
        << buffer2 << endl;

    return 0;
}
```

Введите предложение:

Сравнение ввода строки с помощью cin и cin.get

Из cin прочитана строка:

Сравнение

Строка, прочитанная с помощью cin.get:

ввод строки с помощью cin и cin.get

Рис. 11.13. Сравнение ввода строки из `cin` с помощью операции взять из потока и с помощью `cin.get`

Функция-элемент `getline` действует подобно третьему варианту функции-элемента `get` и помещает нулевой символ после строки в символьном массиве. Но в отличие от `get` функция `getline` удаляет символ ограничитель из потока (т.е. читает этот символ и отбрасывает его); этот символ не сохраняется в символьном массиве. Программа, приведенная на рис. 11.14, демонстрирует использование функции-элемента `getline` для ввода строки текста.

```

// fig11_14.cpp
// Ввод символа с помощью функции-элемента getline.

#include <iostream.h>

const int SIZE = 80;

main()
{
    char buffer[SIZE];

    cout << "Введите предложение:" << endl;
    cin.getline(buffer, SIZE);

    cout << endl << "Введенное предложение:" << endl
        << buffer << endl;

    return (0);
}

```

Введите предложение:

Использование функции-элемента getline

Введенное предложение:

Использование функции-элемента getline

Рис. 11.14. Ввод символов с помощью функции-элемента **getline**

11.4.3. Другие функции-элементы класса istream (peek, putback, ignore)

Функция-элемент **ignore** пропускает заданное число символов (по умолчанию один символ) или завершает свою работу при обнаружении заданного ограничителя (по умолчанию символом ограничителем является **EOF**, который заставляет функцию **ignore** пропускать до конца файла при чтении из файла).

Функция-элемент **putback** возвращает обратно в этот поток предыдущий символ, полученный из входного потока с помощью функции **get**. Функция полезна для приложений, которые просматривают входной поток с целью поиска записи, начинаящейся с заданного символа. Когда этот символ введен, приложение возвращает его в поток, так что он может быть включен в те данные, которые будут вводиться.

Функция-элемент **peek** возвращает очередной символ из входного потока, но не удаляет его из потока.

11.4.4. Сохранение типов данных при вводе-выводе

Язык C++ обеспечивает *сохранение типов данных* при вводе-выводе потоков. Операции **<<** и **>>** перегружены так, чтобы принимать элементы данных заданных типов. Если обрабатываются непредусмотренные данные, то устанавливаются различные флаги ошибок, с помощью которых пользователь

может определить, были ли операции ввода-вывода успешными, или нет. Таким способом программа контролирует типы. Мы обсудим флаги ошибок в разделе 11.8.

11.5. Неформатированный ввод-вывод с использованием `read`, `gcount` и `write`

Неформатированный ввод-вывод выполняется с помощью функций-элементов `read` и `write`. Каждая из них вводит или выводит некоторое число байтов в символьный массив в памяти или из него. Эти байты не подвергаются какому-либо форматированию. Они просто вводятся или выводятся в качестве сырых байтов данных. Например, вызов

```
char buffer[ ] = "ПОЗДРАВЛЯЕМ С ДНЕМ РОЖДЕНИЯ";
cout.write(buffer, 12);
```

выводит первые 12 байтов символьного массива `buffer` (включая нулевые символы, которые могут быть выведены в `cout` и завершить операцию `<<`). Поскольку символьная строка указывает на адрес своего первого символа, то вызов

```
cout.write("ABCDEFGHIJKLMNPQRSTUVWXYZ", 10);
```

отобразит на экране первые 10 символов алфавита.

Функция-элемент `read` вводит в символьный массив указанное число символов. Если считывается меньшее количество символов, то устанавливается флаг `failbit`. Позже мы увидим, каким образом определять, установлен ли флаг `failbit` (см. раздел 11.8).

Функция-элемент `gcount` сообщает о количестве символов, прочитанных последней операцией ввода.

Программа на рис. 11.15 показывает работу функций-элементов `read` и `gcount` класса `istream` и функции-элемента `write` класса `ostream`. Программа вводит 20 символов (из более длинной входной последовательности) в массив символов `buffer` с помощью функции-элемента `read`, определяет число введенных символов с помощью `gcount` и выводит символьный массив `buffer` с помощью `write`.

11.6. Манипуляторы потоков

В языке C++ имеется возможность использовать *манипуляторы потоков*, которые решают задачи форматирования. Манипуляторы потоков позволяют выполнять следующие операции: задание ширины полей, задание точности, установку и сброс флагов формата, задание заполняющего символа полей, сброс потоков, вставку в выходной поток символа новой строки и сброс потока, вставку нулевого символа в выходной поток и пропуск символов разделителей во входном потоке. Более подробные сведения о манипуляторах приведены ниже.

```

// fig11_15.cpp
// Неформатированный ввод-вывод
// с помощью функций-элементов read, gcount и write.
#include <iostream.h>

const int SIZE = 80;

main()
{
    char buffer[SIZE];

    cout << "Введите предложение:" << endl;
    cin.read(buffer, 20);
    cout << endl << "Введенное предложение: " << endl;
    cout.write(buffer, cin.gcount());
    cout << endl;

    return 0;
}

```

Введите предложение:

Использование функций-элементов read, write и gcount

Введенное предложение:

Использование функции

Рис. 11.15. Неформатированный ввод-вывод с помощью функций-элементов **read**, **gcount** и **write**

11.6.1. Манипуляторы потоков **dec**, **oct**, **hex** и **setbase**, задающие основание чисел

Целые числа обычно интерпретируются как десятичные (с основанием 10). Для изменения основания интерпретации целых чисел в потоке запишите манипулятор **hex**, чтобы установить шестнадцатеричный формат представления элементов данных (с основанием 16), или запишите манипулятор **oct**, чтобы установить восьмеричный формат представления данных (с основанием 8). Запишите манипулятор **dec** для возврата к основанию потока 10.

Основание потока может быть также изменено с помощью манипулятора потока **setbase**, который принимает один целый параметр со значениями 10, 8 или 16, задающими соответствующие основания системы счисления. Поскольку манипулятор **setbase** принимает параметр, он называется *параметризованным манипулятором потока*. Использование манипулятора **setbase** или любого другого параметризованного манипулятора требует включение заголовочного файла **<iomanip.h>**. Основание потока остается установленным до тех пор, пока оно не будет изменено явным образом. На рис. 11.16 показано использование манипуляторов потока **hex**, **oct**, **dec** и **setbase**.

```
// fig11_16.cpp
// Использование манипуляторов потока hex, oct, dec и setbase.
#include <iostream.h>
#include <iomanip.h>
main()
{
    int n;

    cout << "Введите десятичное число: ";
    cin >> n;

    cout << n << " в шестнадцатеричном формате равно "
        << hex << n << endl
        << dec << n << " в восьмеричном формате равно "
        << oct << n << endl
        << setbase(10) << n << " в десятичном формате равно "
        << n << endl;

    return 0;
}
```

```
Введите десятичное число: 20
20 в шестнадцатеричном формате равно 14
20 в восьмеричном формате равно 24
20 в десятичном формате равно 20
```

Рис. 11.16. Использование манипуляторов потока **hex**, **oct**, **dec** и **setbase**

11.6.2. Точность чисел с плавающей запятой (**precision**, **setprecision**)

Мы можем управлять точностью печатаемых чисел с плавающей запятой, т.е. числом разрядов справа от десятичной точки, используя манипулятор потока **precision** или функцию-элемент **setprecision**. Вызов любой из этих установок точности действует для всех последующих операций вывода до тех пор, пока не будет произведена следующая установка точности. Функция-элемент **precision** не имеет никаких аргументов и возвращает текущее значение точности. Программа, приведенная на рис. 11.17, использует как функцию-элемент **precision**, так и манипулятор **setprecision** для печати таблицы корня квадратного из числа 2 с точностью, варьирующейся от 0 до 9. Отметим, что точность 0 имеет особое значение. Она восстанавливает точность по умолчанию, равную 6.

11.6.3. Ширина поля (**setw**, **width**)

Функция-элемент **width** класса **ios** устанавливает ширину поля (т.е. число символьных позиций, в которые значение будет выведено, или число символов, которые будут введены) и возвращает предыдущую ширину поля. Если обрабатываемые значения имеют меньше символов, чем заданная ширина поля, то для заполнения лишних позиций используются *заполняющие символы*. Если число символов в обрабатываемом значении больше, чем заданная ширина поля, то лишние символы не отсекаются и число будет напечатано полностью. Установка ширины поля влияет только на следующую операцию

поместить или взять; затем ширина поля устанавливается неявным образом на 0, т.е. поле для представления выходных значений будут просто такой ширины, которая необходима. Функция `width`, не имеющая аргументов, возвращает текущую установку.

```
// fig11_17.cpp
// Управление точностью печати значений с плавающей запятой
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main()
{
    double root2 = sqrt(2.0);

    cout << "Корень квадратный из 2 с точностью 0 - 9." << endl
        << "Точность, установлена с помощью "
        << "функции-элемента precision:" << endl;
    for (int places = 0; places <= 9; places++) {
        cout.precision(places);
        cout << root2 << endl;
    }

    cout << endl <<"Точность установлена с помощью "
        << "манипулятора setprecision:" << endl;

    for (places = 0; places <= 9; places++)
        cout << setprecision(places) << root2 << endl;

    return 0;
}
```

Корень квадратный из 2 с точностью 0 - 9.

Точность, установлена с помощью функции-элемента precision:

```
1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Точность, установлена с помощью манипулятора setprecision:

```
1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Рис. 11.17. Управление точностью печати значений с плавающей запятой

Типичная ошибка программирования 11.4

Если не обеспечивается достаточно широкое поле для обработки выходных данных, то каждое выходное значение печатается с такой шириной поля, которая необходима для его полного представления; это может вызывать сложности при чтении выходных данных.

Программа на рис. 11.18 демонстрирует использование функции-элемента `width` при вводе и при выводе. Обратите внимание, что читается по крайней мере на один символ меньше, чем установленная ширина поля, чтобы обеспечить размещение во входной строке нулевого символа. Помните, что операция взять из потока завершается, когда встречаются не лидирующий символ разделитель (т.е. не предшествующий значащим символам).

Для установки ширины поля может быть также использован манипулятор потока `setw`.

```
// fig11_18.cpp
// Демонстрация функции-элемента width

#include <iostream.h>

main ( )
{
    int w = 4;
    char string[10];

    cout << "Введите предложение:" << endl;
    cin.width(5);

    while (cin >> string) {
        cout.width(w++);
        cout << string << endl;
        cin.width(5);
    }

    return 0;
}
```

Введите предложение:

Это проверка функции-элемента width^Z

Это
проверка
функции-
элемента
width

Рис. 11.18. Демонстрация функции-элемента `width`

11.6.4. Манипуляторы, определяемые пользователем

Пользователи могут создавать собственные манипуляторы потоков. Рис. 11.19 показывает создание и использование новых манипуляторов потоков `bell`, `ret` (возврат каретки), `tab` и `endLine`. Пользователи могут также создавать собственные параметризованные манипуляторы потоков; обратитесь за справкой, как это сделать, к нашему руководству по системе.

11.7. Состояния формата потоков

Различные *флаги формата* задают виды форматирования, которые выполняются во время операций ввода-вывода. Управляют установками флагов функции-элементы `setf`, `unsetf` и `flags`.

11.7.1. Флаги состояний формата

Каждый из флагов состояний формата, показанных на рис. 11.20, (а также тех флагов, которые не показаны) определяется как перечислимый тип в классе `ios` и разъясняется в нескольких следующих разделах.

Эти флаги могут управляться функциями-элементами `flags`, `setf` и `unsetf`, но многие программисты предпочитают использовать манипуляторы потоков (см. раздел 11.7.8). Программист может использовать операцию побитовое ИЛИ | для объединения разных опций в одно значение типа `long` (см. рис. 11.23). Вызов функции-элемента `flags` для потока с заданием этих соединенных операцией | опций устанавливает опции этого потока и возвращает значение типа `long`, содержащее предыдущие значения опций. Это значение часто сохраняется с тем, чтобы можно было впоследствии вызвать функцию `flags` с этим сохраненным значением и восстановить предыдущие значения опций.

Функция `flags` должна задавать значение, выполняющее установку всех флагов. С другой стороны, единственный аргумент функции `setf` задает один или более флагов, соединенных операцией |, и может использовать текущие установки флагов для создания нового состояния формата.

Параметризованный манипулятор потока `setiosflags` выполняет те же функции, что и функция-элемент `setf`. Манипулятор потока `resetiosflags` выполняет те же функции, что и функция-элемент `unsetf`. Чтобы использовать любой из этих манипуляторов потока, убедитесь, что включена директива `#include <iomanip.h>`.

Флаг `skipws` показывает, что операция взять из потока >> должна пропускать символы разделители во входном потоке. По умолчанию операция >> пропускается символы разделители. Чтобы изменить это, т.е. сбросить флаг `skipws`, используйте вызов функции `unsetf (ios::skipws)`. Манипулятор потока `ws` также может быть использован для указания, надо ли пропускать символы разделители.

```
// fig11_19.cpp
// Создание и проверка непараметризованных манипуляторов
// потока, определенных пользователем.

#include <iostream.h>

//Манипулятор bell (использующий управляющую
//последовательность \a)
ostream& bell(ostream& output)
{
    return output << '\a';
}

//Манипулятор ret (использующий управляющую последовательность \r)
ostream& ret(ostream& output)
{
    return output << '\r';
}

//Манипулятор tab (использующий управляющую последовательность \t)
ostream& tab(ostream& output)
{
    return output << '\t';
}

//Манипулятор endLine (использующий управляющую
//последовательность \n)
//и функция-элемент flush
ostream& endLine(ostream& output)
{
    return output << '\n' << flush;
}

main()
{
    cout << "Проверка манипулятора tab:" << endl
        << 'a' << tab << 'b' << tab << 'c' << endl
        << "Проверка манипуляторов ret и bell:" 
        << endl << ".....";
    
    for (int i = 1; i <= 100; i++)
        cout << bell;

    cout << ret << "----" << endl;

    return 0;
}
```

Проверка манипулятора tab:
a b c
Проверка манипуляторов ret и bell:

Рис. 11.19. Создание и проверка непараметризованных манипуляторов потока, определенных пользователем

```

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

```

Рис. 11.20. Флаги состояния формата

11.7.2. Нулевые младшие разряды и десятичные точки (ios::showpoint)

Флаг **showpoint** устанавливается для вывода числа с обязательной печатью десятичной точки и нулевых младших разрядов (нулей в конце числа). Значение с плавающей точкой 79.0 без установки **showpoint** будет напечатано как 79, а с установкой **showpoint** — как 79.000000 (количество нулей определяется заданной точностью). Программа, приведенная на рис. 11.21, демонстрирует использование функции-элемента **setf** для установки флага **showpoint**, управляющего печатью нулевых младших разрядов и десятичной точки для значений с плавающей запятой.

```

// fig11_21.cpp
// Управление процессом печати нулевых младших разрядов
// и десятичной точки для значений с плавающей запятой

#include <iostream.h>
#include <iomanip.h>
#include <math.h>

main ( )
{
    cout << "cout печатает 9.9900 как: " << 9.9900 << endl
        << "cout печатает 9.9000 как: " << 9.9000 << endl
        << "cout печатает 9.0000 как: " << 9.0000 << endl << endl
        << "После установки флага ios::showpoint" << endl;
    cout.setf(ios::showpoint);

    cout << "cout печатает 9.9900 как: " << 9.9900 << endl
        << "cout печатает 9.9000 как: " << 9.9000 << endl
        << "cout печатает 9.0000 как: " << 9.0000 << endl;
    return 0;
}

```

```

cout печатает 9.9900 как: 9.99
cout печатает 9.9000 как: 9.9
cout печатает 9.0000 как: 9

После установки флага ios::showpoint
cout печатает 9.9900 как: 9.990000
cout печатает 9.9000 как: 9.900000
cout печатает 9.0000 как: 9.000000

```

Рис. 11.21. Управление печатью нулевых младших разрядов и десятичных точек для значений с плавающей запятой

11.7.3. Выравнивание (ios::left, ios::right, ios::internal)

Флаги **left** и **right** позволяют выравнивать печать соответственно по левой или правой границам поля с помощью заполняющих символов, печатаемых соответственно в правых или левых пустых позициях. Используемый для заполнения символ задается с помощью функции-элемента **fill** или параметризованного манипулятора потока **setfill** (см. раздел 11.7.4). Рис. 11.22 показывает использование манипуляторов **setw**, **setiosflags** и **resetiosflags** и функций-элементов **setf** и **unsetf** для управления выравниванием по левой и правой границам полей при печати целых чисел.

```

// fig11_22.cpp
// Выравнивание по левой и правой границам поля

#include <iostream.h>
#include <iomanip.h>

main()
{
    int x = 12345;

    cout << "По умолчанию проводится выравнивание"
        << " по правой границе поля:"
        << endl << setw(10) << x << endl << endl
        << "ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ-ЭЛЕМЕНТОВ" << endl
        << "Использование setf для установки ios::left:"
        << endl << setw(10);

    cout.setf(ios::left, ios::adjustfield);
    cout << x << endl
        << "Использование функции unsetf для восстановления умолчания:"
        << endl;
    cout.unsetf(ios::left);
    cout << setw(10) << x << endl << endl
        << "ИСПОЛЬЗОВАНИЕ ПАРАМЕТРИЗИРОВАННЫХ"
        << " МАНИПУЛЯТОРОВ ПОТОКА" << endl
        << "Использование setiosflags для установки ios::left:"
        << endl
        << setw(10) << setiosflags(ios::left) << x << endl

```

```

    << "Использование resetiosflags для восстановления умолчания:"
    << endl << setw(10) << resetiosflags(ios::left) << x
    << endl;
return 0;
}

```

По умолчанию проводится выравнивание по правой границе поля:
12345

ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ-ЭЛЕМЕНТОВ

Использование `setf` для установки `ios::left`:
12345

Использование функции `unsetf` для восстановления умолчания:
12345

ИСПОЛЬЗОВАНИЕ ПАРАМЕТРИЗИРОВАННЫХ МАНИПУЛЯТОРОВ ПОТОКА

Использование `setiosflags` для установки `ios::left`:
12345

Использование `resetiosflags` для восстановления умолчания:
12345

Рис. 11.22. Выравнивание по левой и правой границам поля

Флаг `internal` показывает, что знак числа (или основание, если установлен флаг `ios::showbase` — см. раздел 11.7.5) следует выравнивать по левой границе поля, значение числа следует выравнивать по правой границе, а в промежуточных пустых позициях следует разместить заполняющие символы. Флаги `left`, `right` и `internal` содержатся в статическом элементе данных `ios::adjustfield`. Аргумент `ios::adjustfield` необходимо задавать в качестве второго параметра функции-элемента `setf`, если устанавливаются флаги выравнивания `left`, `right` или `internal`. Это гарантирует, что функция `setf` установит только один из трех флагов (они взаимно исключают друг друга). На рис. 11.23 показан пример использования манипуляторов потока `setiosflags` и `setw` для задания внутренних пробелов. Обратите внимание, что использование флага `ios::showpos` приводит к обязательной печати знака плюс.

```

// fig11_23.cpp
// Печать целого числа с внутренними пробелами
// и принудительной печатью знака плюс.

#include <iostream.h>
#include <iomanip.h>

main()
{
    cout << setiosflags (ios::internal | ios::showpos)
        << setw(10) << 123 << endl;
    return 0;
}
+ 123

```

Рис. 11.23. Печать целого числа с внутренними пробелами и принудительной печатью знака плюс

11.7.4. Заполнение (fill, setfill)

Функция-элемент *fill* задает заполняющий символ, который используется при выравнивании в полях; если никакое значение заполняющего символа не задано, то для заполнения используются пробелы. Функция *fill* возвращает предыдущий заполняющий символ. Манипулятор *setfill* также устанавливает значение заполняющего символа. На рис. 11.24 показано использование функции-элемента *fill* и манипулятора *setfill* для управления установкой и восстановлением заполняющего символа.

```
// fig11_24.cpp
// Использование функции-элемента fill и
// манипулятора setfill для изменения заполняющего символа полей,
// ширина которых превышает ширину, необходимую для печати

#include <iostream.h>
#include <iomanip.h>

main()
{
    int x = 10000;

    cout << x
        << " напечатанное как целое с выравниванием right и left"
        << endl
        << "и как шестнадцатеричное с выравниванием internal"
        << endl
        << "Использование заполняющего символа по умолчанию (пробела)"
        << endl;
    cout.setf(ios::showbase);
    cout << setw(10) << x << endl;
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(10) << x << endl;
    cout.setf(ios::internal, ios::adjustfield);
    cout << setw(10) << hex << x << endl;

    cout << "Использование различных заполняющих символов:"
        << endl;
    cout.setf(ios::right, ios::adjustfield);
    cout.fill('*');
    cout << setw(10) << dec << x << endl;
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(10) << setfill('%') << x << endl;
    cout.setf(ios::internal, ios::adjustfield);
    cout << setw(10) << setfill('*') << hex
        << x << endl;

    return 0;
}
```

```
10000 напечатанное как целое с выравниванием right и left
и как шестнадцатеричное с выравниванием internal
Использование заполняющего символа по умолчанию (пробела)
10000
10000
```

```
0x 2710
```

Использование различных заполняющих символов:

```
*****10000
1000088888
0x****2710
```

Рис. 11.24. Использование функции-элемента `fill` и манипулятора `setfill` для изменения заполняющего символа полей, ширина которых превышает ширину, необходимую для печати

11.7.5. Основание системы счисления (`ios::dec`, `ios::oct`, `ios::hex`, `ios::showbase`)

Статический элемент `ios::basefield` (используемый аналогично тому, как с помощью `setf` используется `ios::adjustfield`) включает биты флагов `ios::oct`, `ios::hex` и `ios::dec`, которые трактуют целые числа соответственно как восьмеричные, шестнадцатеричные или десятичные значения. Если ни один из этих битов не установлен, то по умолчанию целые числа при операции поместить в поток трактуются как десятичные. При операции взять из потока данные по умолчанию обрабатываются в той форме, в которой они поступают: целые, начинающиеся с 0, обрабатываются как восьмеричные значения, целые, начинающиеся с 0х или 0X, обрабатываются как шестнадцатеричные значения, а все другие целые числа обрабатываются как десятичные значения. Если же для потока указывается определенное основание, то все целые значения данных в этом потоке обрабатываются с помощью этого основания до тех пор, пока не задано новое основание или пока не завершится программа.

```
// fig11_25.cpp
// Использование флага ios::showbase
#include <iostream.h>
#include <iomanip.h>

main()
{
    int x = 100;

    cout << setiosflags(ios::showbase)
        << "Печать целых чисел с разными основаниями:" << endl
        << x << endl
        << oct << x << endl
        << hex << x << endl;
    return 0;
}

Печать целых чисел с разными основаниями:
100
0144
0x64
```

Рис. 11.25. Использование флага `ios::showbase`

Устанавливайте флаг **showbase**, чтобы задавать основание целочисленных значений для вывода. Десятичные числа выводятся обычным способом, восьмеричные числа выводятся с нулем в старшем разряде (ведущим нулем), шестнадцатеричные числа выводятся с индикатором **0x** или **0X** в старших разрядах (флаг **uppercase** определяет, какой из этих двух вариантов будет выбран, см. раздел 11.7.7). На рис. 11.25 показано использование флага **showbase** для печати целых чисел в десятичном, восьмеричном и шестнадцатеричном форматах.

11.7.6. Числа с плавающей запятой; экспоненциальный формат (**ios::scientific**, **ios::fixed**)

Флаги **ios::scientific** и **ios::fixed** содержатся в статическом элементе данных **ios::floatfield**, используемом аналогично тому, как с помощью **setf** используется **ios::adjustfield**. Эти флаги управляют форматом вывода для чисел с плавающей запятой. Флаг **scientific** устанавливается для вывода чисел с плавающей запятой в экспоненциальном формате (в C++ он назван научным, но это не тот формат, который называется научным в других языках). Флаг **fixed** устанавливается для вывода чисел с плавающей запятой в формате с фиксированной точкой с заданным количеством разрядов (указанным с помощью функции-элемента **precision**) справа от десятичной точки. Без установки этих флагов значения чисел с десятичной запятой сами определяют форматы выводимых данных.

Вызов **cout.setf(0, ios::floatfield)** восстанавливает в системе формат вывода чисел с плавающей запятой по умолчанию. Программа на рис. 11.26 показывает вывод на экран чисел с плавающей запятой в фиксированном и экспоненциальном форматах с помощью двух аргументов функции **setf** и флага **ios::floatfield**.

```
// fig11_26.cpp
// Отображение на экране значений с плавающей запятой
// в форматах по умолчанию, экспоненциальном
// и с фиксированной точкой

#include <iostream.h>

main()
{
    double x = .001234567, y = 1.946e9;

    cout << "Отображение в формате по умолчанию:" << endl
        << x << '\t' << y << endl;
    cout.setf(ios::scientific, ios::floatfield);
    cout << "Отображение в экспоненциальном формате:" << endl
        << x << '\t' << y << endl;
    cout.unsetf(ios::scientific);
    cout << "Отображение в формате по умолчанию после unsetf:"
        << endl
        << x << '\t' << y << endl;
    cout.setf(ios::fixed, ios::floatfield);
    cout << "Отображение в формате с фиксированной точкой:" << endl
        << x << '\t' << y << endl;
```

```

    return 0;
}



---


Отображение в формате по умолчанию:
0.001235 1.946e+09
Отображение в экспоненциальном формате:
1.234567e-03 1.946e+09
Отображение в формате по умолчанию после unsetf:
0.001235 1.946e+09
Отображение в формате с фиксированной точкой:
0.001235 1946000000

```

Рис. 11.26. Отображение значений с плавающей запятой в форматах по умолчанию, экспоненциальном и с фиксированной точкой

11.7.7. Управление выводом в нижнем и верхнем регистрах (ios::uppercase)

Флаг `ios::uppercase` устанавливает вывод в верхнем регистре символов **X** и **E** в шестнадцатеричном и экспоненциальном форматах соответственно (см. рис.11.27). Когда флаг `ios::uppercase` установлен, все буквы шестнадцатеричного формата печатаются в верхнем регистре.

```

// fig11_27.cpp
// Использование флага ios::uppercase
#include <iostream.h>
#include <iomanip.h>

main()
{
    cout << setiosflags(ios::uppercase)
        << "Печать в верхнем регистре чисел в экспоненциальном фор-
мате"
        << endl << "и шестнадцатеричных значений:" << endl
        << hex << 123456789 << endl;
    return 0;
}

```

```

Печать в верхнем регистре чисел в экспоненциальном формате
и шестнадцатеричных значений:
4.345E+10
75BCD15

```

Рис. 11.27. Использование флага `ios::uppercase`

11.7.8. Установка и сброс флагов формата (flags, setiosflags и resetiosflags)

Если у функции-элемента `flags` нет аргумента, то она просто возвращает текущие установки флагов формата в виде значения типа `long`. Функция-элемент `flags` с аргументом типа `long` устанавливает флаги формата, заданные

с помощью аргумента, и возвращает прежние установки флагов. Любые флаги форматов, которые не определены в аргументе для функции-элемента `flags`, остаются в исходном состоянии. Заметим, что исходные установки флагов каждой системы могут различаться. Программа, приведенная на рис. 11.28, демонстрирует использование функции-элемента `flags` для установки нового состояния формата и сохранения прежнего его состояния с последующим восстановлением исходных установок форматов.

```
// fig11_28.cpp
// Демонстрация функции-элемента flags
#include <iostream.h>

main()
{
    int i = 1000;
    double d = 0.0947628;

    cout << "Значение переменной flags равно: "
        << cout.flags() << endl
        << "Печать данных типа int и double в исходном формате:"
        << endl
        << i << '\t' << d << endl << endl;
    long originalFormat = cout.flags(ios::oct | ios::scientific);
    cout << "Значение переменной flags равно: "
        << cout.flags() << endl
        << "Печать данных типа int и double в новом формате,"
        << endl
        << "заданном с помощью функции-элемента flags:" << endl
        << i << '\t' << d << endl << endl;
    cout.flags(originalFormat);
    cout << "Значение переменной flags равно: "
        << cout.flags() << endl
        << "Повторная печать данных в исходном формате:" << endl
        << i << '\t' << d << endl;

    return 0;
}
```

```
Значение переменной flags равно: 1
Печать данных типа int и double в исходном формате:
1000 0.094763
```

```
Значение переменной flags равно: 4040
Печать данных типа int и double в новом формате,
заданном с помощью функции-элемента flags:
1750 9.47628e-02
```

```
Значение переменной flags равно: 1
Повторная печать данных в исходном формате:
1000 0.094763
```

Рис. 11.28. Применение функции-элемента `flags`

Функция-элемент `setf` устанавливает флаги формата, заданные в качестве аргумента, и возвращает прежние установки этих флагов в виде значения типа `long`, которое можно запомнить как показано ниже:

```
long previousFlagSettings =
    cout.setf(ios::showpoint | ios::showpos);
```

Функция-элемент `setf` имеет два аргумента типа `long`:

```
cout.setf(ios::left, ios::adjustfield);
```

Этот оператор сначала очищает бит `ios::adjustfield`, а затем устанавливает флаг `ios::left`. Этот вариант флага `setf` влияет на битовые поля, связанные с `ios::basefield` (содержит `ios::dec`, `ios::oct` и `ios::hex`), с `ios::floatfield` (содержит `ios::scientific` и `ios::fixed`) и с `ios::adjustfield` (содержит `ios::left`, `ios::right` и `ios::internal`).

Функция-элемент `unsetf` сбрасывает указанные флаги и возвращает значения этих флагов, которые существовали ранее.

11.8. Состояния ошибок потока

Состояние потока может быть проверено с помощью битов класса `ios` — базового для классов `istream`, `ostream` и `iostream`, которые мы использовали для ввода-вывода.

Бит `eofbit` для входного потока автоматически устанавливается, когда встречается признак конца файла. Программа может использовать функцию-элемент `eof`, чтобы определить, встретился ли в потоке признак конца файла.

Вызов

```
cin.eof()
```

возвращает `true`, если в `cin` встретился признак конца файла, и `false` в противном случае.

Бит `failbit` устанавливается для потока, если в потоке происходит ошибка форматирования, но символы не утеряны. Функция-элемент `fail` определяет, не было ли отказа в операции с потоком; обычно после такой неудачной операции данные можно восстановить.

Бит `badbit` устанавливается для потока при возникновении ошибки, которая приводит к потере данных. Функция-элемент `bad` определяет, имела ли операция с потоками такую ошибку. При столь серьезных ошибках данные обычно не восстанавливаются.

Бит `goodbit` устанавливается для потока, если ни один из битов `eofbit`, `failbit` и `badbit` не установлен.

Функция-элемент `good` возвращает `true`, если для данного потока все функции `bad`, `fail` и `eof` должны вернуть `false`. Операции ввода-вывода можно нормально выполнять только с такими «хорошими» (`good`) потоками.

Функция-элемент `rdstate` возвращает состояние ошибки потока. Вызов `cout.rdstate`, например, вернул бы состояние потока, которое затем могло бы использоваться в операторе `switch`, который проверял бы `ios::eofbit`, `ios::badbit`, `ios::failbit` и `ios::goodbit`. Но более удобными средствами проверки состояния является использование функций-элементов `eof`, `bad`, `fail` и `good`. Использование этих функций не требует, чтобы программист оперировал с отдельными битами состояния.

Функция-элемент `clear` обычно используется для восстановления потока в нормальное состояние (когда функция-элемент `good` возвращает истину), при котором можно продолжать операции ввода-вывода данного потока. По

умолчанию параметр функции `clear` принимает значение `ios::goodbit`, так что оператор

```
cin.clear();
```

очистит входной поток `cin` и установит `goodbit` для этого потока. Оператор

```
cin.clear(ios::failbit)
```

устанавливает `failbit`. Пользователь может захотеть сделать это, если столкнется с проблемами при обработке `cin` с типом, определенным пользователем. Имя функции `clear` не кажется подходящим в этом контексте, но подобное применение `clear` вполне корректно.

Программа, приведенная на рис. 11.29, демонстрирует применение функций-элементов `rdstate`, `eof`, `fail`, `bad`, `good` и `clear`.

Функция-элемент `operator!` возвращает истину в том случае, если установлен либо `badbit`, либо `failbit`, либо оба вместе. Функция-элемент `operator void *` возвращает `false`, если установлен либо `badbit`, либо `failbit`, либо оба вместе. Эти функции полезны при обработке файлов и проверке истинности или ложности условия в структуре выбора или в структуре повторения.

11.9. Ввод-вывод определенных пользователем типов данных

В языке C++ предусмотрена возможность вводить и выводить данные стандартных типов, используя операции взять из потока `>>` и операции поместить в поток `<<`. Эти операции перегружены для обработки любых данных стандартного типа, включая строки и адреса памяти. Программист может перегрузить операции поместить в поток и взять из потока для выполнения ввода-вывода данных, тип которых определен пользователем. Программа, приведенная на рис. 11.30, демонстрирует перегрузку операций взять из потока и поместить в поток для обработки определенного пользователем класса, названного `PhoneNumber` и содержащего номера телефонов. Заметим, что эта программа предполагает правильный ввод телефонных номеров. Мы оставим на упражнения поиск ошибок этого ввода.

Операция взять из потока принимает в качестве параметров ссылку на `istream`, ссылку на определенный пользователем тип данных (в нашем случае — `PhoneNumber`) и возвращает ссылку на `istream`. На рис. 11.30 демонстрируется перегруженная операция взять из потока, которая используется для ввода номеров телефонов в виде

(800) 555-1212

в объекты типа `PhoneNumber`. Функция-операция считывает три части телефонного номера в элементы `areaCode`, `exchange` и `line` объекта типа `PhoneNumber` (объект `num` в функции-операции и объект `phone` в функции `main`). Круглые скобки, пробел и символ дефиса отбрасываются с помощью вызова функции-элемента `ignore` класса `istream`. Функция-операция возвращает ссылку на `input` класса `istream`. С помощью возвращения ссылки на поток операция ввода объектов типа `PhoneNumber` может быть сцеплена с операцией ввода другого объекта типа `PhoneNumber` или какого-то другого типа данных. Например, два объекта типа `PhoneNumber` могли быть введены следующим образом:

```
cin >> phone1 >> phone2;
```

```

// fig11_29.cpp
// Проверка состояний ошибок.
#include <iostream.h>

main()
{
    int x;
    cout << "До ошибочной операции ввода:" << endl
        << "cin.rdstate(): " << cin.rdstate() << endl
        << " cin.eof (): " << cin.eof () << endl
        << " cin. fail (): " << cin. fail () << endl
        << " cin.bad(): " << cin.bad() << endl
        << " cin.good(): " << cin.good() << endl << endl
        << "Ожидается целое, но вводится символ: ";
    cin>>x;

    cout << endl << "После ошибочной операции ввода:" << endl
        << "cin.rdstate(): " << cin.rdstate() << endl
        << " cin.eof (): " << cin.eof () << endl
        << " cin. fail (): " << cin. fail () << endl
        << " cin.bad(): " << cin.bad() << endl
        << " cin.good(): " << cin.good() << endl << endl;

    cin.clear();

    cout << "После cin.clear():" << endl
        << " cin. fail (): " << cin. fail () << endl
        << " cin.good(): " << cin.good() << endl << endl;

    return 0;
}

```

До ошибочной операции ввода:

```

cin.rdstate(): 0
  cin.eof (): 0
  cin. fail (): 0
  cin.bad(): 0
  cin.good(): 1

```

Ожидается целое, но вводится символ: А

После ошибочной операции ввода:

```

cin.rdstate(): 2
  cin.eof (): 0
  cin. fail (): 2
  cin.bad(): 0
  cin.good(): 0

```

После cin.clear():

```

  cin. fail (): 0
  cin.good(): 1

```

Рис. 11.29. Проверка состояний ошибок

Операция поместить в поток принимает в качестве параметров ссылку на `ostream`, ссылку на тип, определенный пользователем (в нашем случае — `PhoneNumber`), и возвращает ссылку на `ostream`. В программе, приведенной на рис. 11.30, показано, как перегруженная операция поместить в поток отображает объекты типа `PhoneNumber` в том же виде, в котором они вводились.

```
// fig11_30.cpp
// Операции поместить в поток и взять из потока определенные
// пользователем
#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 << "Введите номер телефона "
        << "в формате (123) 456-7890;" << endl;
    cin >> phone;
    cout << "Введенный номер телефона:" << endl
        << phone << endl;

    return 0;
}

Введите номер телефона в формате (123) 456-7890:
(800) 555-1212
Введенный номер телефона:
(800) 555-1212
```

Рис. 11.30. Определенные пользователем операции поместить в поток и взять из потока

Функция `operator` отображает части телефонного номера в виде строк, поскольку эти части хранятся в формате строк (вспомните, что функция-элемент `getline` класса `istream` сохраняет в памяти нулевой символ после завершения ввода).

Перегруженные функции `operator` объявляются в классе `PhoneNumber` как дружественные функции. Перегруженные операции ввода-вывода должны быть объявлены как дружественные, если им необходим доступ к каким-то элементам, помимо открытых. Друзья класса имеют доступ к закрытым элементам.

Замечание по технике программирования 11.3

Новые возможности ввода-вывода данных, тип которых определен пользователем, добавляются в C++ без модификации объявлений или элементов-данных закрытого типа в классе `ostream` и в классе `istream`. Это способствует расширяемости – наиболее привлекательной черте языка C++.

11.10. Связывание выходного потока с входным

Интерактивные приложения обычно включают класс `istream` для ввода и класс `ostream` для вывода. Когда на экране появляется приглашение на ввод, пользователь вводит соответствующие данные. Очевидно, что приглашение на ввод должно появляться до осуществления операции ввода. При буферизации выходных данных их вывод на экран осуществляется только когда буфер окажется полным, когда вывод явным образом сбрасываются программой или автоматически в конце программы. В языке C++ имеется функция-элемент `tie` для синхронизации (связывания) выполнения операций над потоками `istream` и `ostream`, которая гарантирует, что вывод появится раньше последующего ввода. Вызов

```
cin.tie(&cout);
```

связывает `cout` (класса `ostream`) и `cin` (класса `istream`). В действительности этот вызов является лишним, поскольку C++ автоматически выполняет эту операцию при создании стандартной пользовательской среды ввода-вывода. Пользователю следует, однако, явным образом связывать другие пары потоков классов `ostream` и `istream`. Для того, чтобы развязать входной поток `inputStream` от выходного, используйте следующий вызов :

```
inputStream.tie(0);
```

Резюме

- Каждая операция ввода-вывода осуществляется способом, чувствительным к типу данных.
- В языке C++ производится ввод-вывод *потоков* байтов. Поток — это просто последовательность байтов.

- Механизм ввода-вывода заключается в пересылке байтов данных от устройств в оперативную память и обратно эффективным и надежным способом.
- Язык C++ предоставляет возможности для ввода-вывода как на «низком», так и на «высоком» уровнях. Ввод-вывод на низком уровне сводится к тому, что некоторое число байтов данных просто следует переслать от устройства в память или из памяти в устройство. Ввод-вывод на высоком уровне заключается в том, что байты группируются в такие значащие элементы данных, как, например, целые числа, числа с плавающей запятой, символы, строки, а также данные типов, определенных пользователем.
- Язык C++ предоставляет возможности для неформатированного и форматированного ввода-вывода. Неформатированный ввод-вывод позволяет осуществлять пересылку файлов с высокой скоростью, но он обрабатывает только сырье данные, которыми сложно пользоваться. Форматированный ввод-вывод обрабатывает структурированные данные, но требует дополнительного времени на их обработку, что может быть недостатком при передаче больших объемов данных.
- Большинство программ на C++ включает заголовочный файл `<iostream.h>`, который содержит основную информацию, необходимую для всех операций ввода-вывода.
- Заголовочный файл `<iomanip.h>` содержит информацию, полезную для обработки форматированного ввода-вывода при помощи параметризованных манипуляторов потока.
- Заголовочный файл `<fstream.h>` содержит информацию, необходимую для проведения операций с файлами.
- Заголовочный файл `<strstream.h>` содержит необходимую информацию для выполнения форматированного ввода-вывода в память.
- Заголовочный файл `<stdiostream.h>` включает важные сведения для программ, использующих для выполнения операций ввода-вывода сочетание стилей языков С и C++.
- Класс `istream` поддерживает операции ввода потоков.
- Класс `ostream` поддерживает операции вывода потоков.
- Класс `iostream` поддерживает как операции ввода, так и операции вывода потоков.
- Классы `istream` и `ostream` являются производными классами прямого наследования базового класса `ios`.
- Класс `iostream` является производным классом множественного наследования классов `istream` и `ostream`.
- Операция сдвига влево (`<<`) перегружена для обозначения вывода в поток и называется операцией поместить в поток.
- Операция сдвига вправо (`>>`) перегружена для обозначения ввода потока и называется операцией взять из потока.
- Объект стандартного потока ввода `cin` класса `istream` привязан к стандартному устройству ввода, обычно к клавиатуре.

- Объект стандартного потока вывода `cout` класса `ostream` привязан к стандартному устройству вывода, обычно к экрану дисплея.
- Объект `cerr` класса `ostream` привязан к стандартному устройству вывода сообщений об ошибках. Выводимые потоки данных для объекта `cerr` являются небуферизованными. Это означает, что каждая операция поместить в `cerr` приводит к мгновенному появлению выводимых сообщений об ошибках.
- Манипулятор потока `endl` вызывает переход на новую строку и кроме того приводит к сбросу буфера вывода.
- Компилятор C++ автоматически определяет типы данных при вводе и выводе.
- По умолчанию адреса отображаются в шестнадцатеричном формате.
- Для печати адреса следует привести тип указателя к типу `void *`.
- Функция-элемент `put` выводит одиночный символ. Вызовы функции `put` могут быть сцепленными.
- Ввод потока осуществляется операцией взять из потока `>>`. Эта операция автоматически игнорирует символы разделители во входном потоке.
- Операция `>>` возвращает `false`, если в потоке встретился признак конца файла.
- Операция взять из потока приводит к установке бита `failbit` при вводе данных неправильного типа и к установке бита `badbit` при неуспешном завершении операции.
- Можно последовательно вводить данные, используя операцию взять из потока в условном операторе заголовка цикла `while`. Операция взять из потока возвращает `false`, когда встречается признак конца файла.
- Функция-элемент `get` без аргументов вводит одиночный символ из указанного потока и возвращает этот символ. Этот вариант функции `get` возвращает `EOF`, когда в потоке встречается признак конца файла.
- Функции-элемента `get` с символьным аргументом типа `char` вводят один символ. Функция возвращает `EOF`, когда встречается признак конца файла; в остальных случаях возвращается ссылка на тот объект класса `istream`, для которого вызывалась функция-элемент `get`.
- Функции-элемента `get` с тремя аргументами: символьным массивом, максимальное число символов и ограничителем (по умолчанию значение `'\n'`), читает символы из входного потока до тех пор, пока не достигается число символов, на 1 меньше указанного максимального числа, или пока не прочтется ограничитель. Введенная строка заканчивается нулевым символом. Ограничитель в символьный массив не помещается, а остается во входном потоке.
- Функция-элемент `getline` действует подобно варианту функции-элемента `get` с тремя аргументами. Функция `getline` удаляет символ ограничитель из входного потока и не сохраняет его в строке.
- Функция-элемент `ignore` пропускает заданное число символов (по умолчанию один символ) во входном потоке; она завершает свою работу

при обнаружении заданного ограничителя (по умолчанию символом ограничителем является EOF).

- Функция-элемент `putback` возвращает обратно в этот поток предыдущий символ, полученный из входного потока с помощью функции `get`.
- Функция-элемент `peek` возвращает очередной символ из входного потока, но не удаляет его из потока.
- Язык C++ обеспечивает сохранение типов данных при вводе-выводе. Если операциями << и >> обрабатываются непредусмотренные данные, то устанавливаются различные флаги ошибок, с помощью которых пользователь может определить, были ли операции ввода-вывода успешными, или нет.
- Неформатированный ввод-вывод выполняется с помощью функций-элементов `read` и `write`. Каждая из них вводит некоторое число байтов в символьный массив в памяти или выводит их из него. Эти байты не подвергаются какому-либо форматированию и вводятся или выводятся как сырье байты.
- Функция-элемент `gcount` сообщает о количестве символов, прочитанных последней операцией `read`.
- Функция-элемент `read` вводит в символьный массив указанное число символов. Если считывается меньшее количество символов, то устанавливается бит `failbit`.
- Для изменения основания интерпретации целых чисел в потоке используйте манипулятор `hex`, чтобы установить шестнадцатеричный формат представления элементов данных (с основанием 16), или манипулятор `oct`, чтобы установить восьмеричный формат представления данных (с основанием 8). Используйте манипулятор `dec` для возврата к основанию потока 10. Основание потока остается установленным до тех пор, пока оно не будет изменено явным образом новой установкой.
- Основание потока может быть также изменено с помощью манипулятора потока `setbase`, который принимает один целый параметр со значениями 10, 8 или 16, задающими соответствующие основания системы счисления.
- Точностью печатаемых чисел с плавающей запятой можно управлять, используя манипулятор потока `precision` или функцию-элемент `setprecision`. Вызов любой из этих установок точности действует для всех последующих операций вывода до тех пор, пока не будет произведена следующая установка точности. Функция-элемент `precision` не имеет никаких аргументов и возвращает текущее значение точности. Точность 0 устанавливает точность по умолчанию, равную 6.
- При использовании параметризованных манипуляторов потоков необходимо в программу включать заголовочный файл `<iomanip.h>`.
- Функция-элемент `width` устанавливает ширину поля и возвращает предыдущую ширину поля. Если обрабатываемые значения имеют меньше символов, чем заданная ширина поля, то для заполнения лишних позиций используются заполняющие символы. Установка ширины поля влияет только на следующую операцию поместить или взять; затем ширина поля устанавливается неявным образом на 0, т.е. поле

для представления выходных значений будут просто такой ширины, которая необходима. Если число символов в обрабатываемом значении больше, чем заданная ширина поля, то число будет напечатано полностью. Функция `width`, не имеющая аргументов, возвращает текущую установку. Для установки ширины поля может быть также использован манипулятор потока `setw`.

- При вводе манипулятор потока `setw` устанавливает максимальный размер строки; если вводится строка, превышающая заданный размер, то вводимая строка разбивается на фрагменты, не превышающие заданного размера.
- Пользователи могут создавать собственные манипуляторы потоков.
- Функции-элементы `setf`, `unsetf` и `flags` управляют установками флагов.
- Флаг `skipws` показывает, что операция взять из потока `>>` должна пропускать символы разделители во входном потоке. Манипулятор потока `ws` также может быть использован для указания, надо ли пропускать начальные символы разделители.
- Флаги форматов определяются в классе `ios` как переменные перечислимого типа.
- Флаги форматов могут управляться с помощью функций-элементов `flags` и `setf`, но многие программисты предпочитают использовать манипуляторы потоков. Можно использовать операцию побитовое ИЛИ `|` для объединения разных опций в одно значение типа `long`. Вызов функции-элемента `flags` для потока с заданием этих соединенных операций `|` опций устанавливает опции этого потока и возвращает значение типа `long`, содержащее предыдущие значения опций. Это значение часто сохраняется с тем, чтобы можно было впоследствии вызвать функцию `flags` с этим сохраненным значением и восстановить предыдущие значения опций.
- Функция `flags` должна задавать значение, выполняющее установку всех флагов. С другой стороны, единственный аргумент функции `setf` задает один или более флагов, соединенных операцией `|`, и может использовать текущие установки флагов для создания нового состояния формата.
- Флаг `showpoint` устанавливается для вывода числа с обязательной печатью десятичной точки и нулевых младших разрядов, число которых определяется заданной точностью.
- Флаги `left` и `right` позволяют выравнивать печать соответственно по левой или правой границам поля с помощью заполняющих символов, печатаемых соответственно в правых или левых пустых позициях.
- Флаг `internal` показывает, что знак числа (или основание, если установлен флаг `ios::showbase`) следует выравнивать по левой границе поля, значение числа следует выравнивать по правой границе, а в промежуточных пустых позициях следует разместить заполняющие символы.
- Флаги `left`, `right` и `internal` содержатся в статическом элементе данных `ios::adjustfield`.
- Функция-элемент `fill` задает заполняющий символ, который используется при выравнивании в полях с помощью `left`, `right` и `internal` (по

умолчанию — пробел). Функция `fill` возвращает предыдущий заполняющий символ. Манипулятор `setfill` также устанавливает значение заполняющего символа.

- Статический элемент `ios::basefield` включает биты флагов `oct`, `hex` и `dec`, которые трактуют целые числа соответственно как восьмеричные, шестнадцатеричные или десятичные значения. Если ни один из этих битов не установлен, то по умолчанию целые числа при операции поместить в поток трактуются как десятичные. При операции `взять из` потока данные по умолчанию обрабатываются в той форме, в которой они поступают.
- Флаг `showbase` устанавливается, чтобы задавать основание целочисленных значений для вывода.
- В статическом элементе данных `ios::floatfield` содержатся флаги `scientific` и `fixed`. Устанавливайте флаг `scientific` для вывода чисел с плавающей запятой в экспоненциальном формате. Устанавливайте флаг `fixed` для вывода чисел с плавающей запятой в формате с фиксированной точкой с заданным количеством разрядов, указанным с помощью функции-элемента `precision`.
- Вызов `cout.setf (0, ios::floatfield)` восстанавливает формат вывода чисел с плавающей запятой по умолчанию.
- Устанавливайте флаг `uppercase` для вывода в верхнем регистре символов `X` и `E` в шестнадцатеричном и экспоненциальном форматах соответственно. Когда флаг `ios::uppercase` установлен, все буквы шестнадцатеричного формата печатаются в верхнем регистре.
- Функция-элемент `flags` без аргумента возвращает текущие установки флагов формата в виде значения типа `long`. Функция-элемент `flags` с аргументом типа `long` устанавливает флаги формата, заданные с помощью аргумента, и возвращает прежние установки флагов.
- Функция-элемент `setf` устанавливает флаги формата, заданные в качестве аргумента, и возвращает прежние установки флагов в виде значения типа `long`.
- Функция-элемент `setf (long setBits, long resetBits)` очищает биты `resetBits`, а затем устанавливает биты `setBits`.
- Функция-элемент `unsetf` сбрасывает указанные флаги и возвращает значения этих флагов, которые существовали ранее.
- Параметризованный манипулятор потока `setiosflags` выполняет те же функции, что и функция-элемент `flags`.
- Манипулятор потока `resetiosflags` выполняет те же функции, что и функция-элемент `unsetf`.
- Состояние потока может быть проверено с помощью битов класса `ios`.
- Бит `eofbit` для входного потока автоматически устанавливается, когда встречается признак конца файла при операции ввода. Программа может использовать функцию-элемент `eof`, чтобы определить, установлен ли `eofbit`.
- Бит `failbit` устанавливается для потока, если в потоке происходит ошибка форматирования, но символы не потеряны. Функция-элемент `fail`

определяет, не было ли отказа в операции с потоком; обычно после такой неудачной операции данные можно восстановить.

- Бит `badbit` устанавливается для потока при возникновении ошибки, которая приводит к потере данных. Функция-элемент `bad` определяет, имела ли операция с потоками такую ошибку. При столь серьезных ошибках данные обычно не восстанавливаются.
- Функция-элемент `good` возвращает `true`, если для данного потока все функции `bad`, `fail` и `eof` должны вернуть `false`. Операции ввода-вывода можно нормально выполнять только с такими «хорошими» потоками.
- Функция-элемент `rdstate` возвращает состояние ошибки потока.
- Функция-элемент `clear` обычно используется для восстановления потока в нормальное состояние (когда функция-элемент `good` возвращает истину), при котором можно продолжать операции ввода-вывода данного потока.
- Программист может перегрузить операции поместить в поток и взять из потока для выполнения ввода-вывода данных, тип которых определен пользователем.
- Перегруженная операция взять из потока принимает в качестве параметров ссылку на `istream`, ссылку на определенный пользователем тип данных и возвращает ссылку на `istream`.
- Перегруженная операция поместить в поток принимает в качестве параметров ссылку на `ostream`, ссылку на тип, определенный пользователем, и возвращает ссылку на `ostream`.
- Часто перегруженные функции `operator` объявляются как дружественные функции класса. Это позволяет им иметь доступ к закрытым элементам класса.
- В языке C++ имеется функция-элемент `tie` для синхронизации (связывания) выполнения операций над потоками `istream` и `ostream`, которая гарантирует, что вывод появится раньше последующего ввода.

Терминология

<code>badbit</code>	<code>ios::showpos</code>
<code>cerr</code>	<code>left</code>
<code>cin</code>	<code>skipws</code>
<code>clog</code>	<code>uppercase</code>
<code>cout</code>	<code>width</code>
<code>endl</code>	библиотека классов потоков
<code>eofbit</code>	ввод потока
<code>failbit</code>	ведущие (начальные) 0x или 0X
<code>ios::adjustfield</code>	(в шестнадцатеричном формате)
<code>ios::basefield</code>	ведущий (начальный) нуль
<code>ios::fixed</code>	(в восьмеричном формате)
<code>ios::floatfield</code>	вывод потока
<code>ios::internal</code>	выравнивание по левой границе поля
<code>ios::scientific</code>	выравнивание по правой границе
<code>ios::showbase</code>	поля
<code>ios::showpoint</code>	заполнение

заполняющий символ	стандартный заголовочный файл <iomanip.h>
заполняющий символ по умолчанию (пробел)	точность по умолчанию
класс fstream	флаги формата
класс ifstream	форматированный ввод-вывод
класс ios	в память
класс iostream	функция-элемент bad
класс istream	функция-элемент clear
класс ofstream	функция-элемент eof
класс ostream	функция-элемент fail
манипулятор stream	функция-элемент fill
манипулятор потока dec	функция-элемент flags
манипулятор потока flush	функция-элемент flush
манипулятор потока hex	функция-элемент gcount
манипулятор потока oct	функция-элемент geline
манипулятор потока resetiosflags	функция-элемент get
манипулятор потока setbase	функция-элемент good
манипулятор потока setfill	функция-элемент ignore
манипулятор потока setiosflags	функция-элемент operator void *
манипулятор потока setprecision	функция-элемент operator!
манипулятор потока setw	функция-элемент peek
неформатированный ввод-вывод	функция-элемент precision
операция взять из потока (>>)	функция-элемент put
операция поместить в поток (<<)	функция-элемент putback
определенные пользователем потоки	функция-элемент rdstate
параметризованный манипулятор потока	функция-элемент read
предопределенные потоки	функция-элемент setf
признак конца файла	функция-элемент tie
расширяемость	функция-элемент unsetf
символы разделители	функция-элемент write
состояния формата	функция-элемент ws
сохранение типов данных при вводе-выводе	ширина поля

Типичные ошибки программирования

- 11.1. Попытка прочитать данные из потока класса **ostream** (или из любого другого выходного потока).
- 11.2. Попытка записать данные в поток класса **istream** (или в любой другой входной поток).
- 11.3. Отсутствие круглых скобок, устанавливающих последовательность выполнения операций, при использовании операций с относительно высоким приоритетом вставить в поток **<<** и **взять из потока >>**.
- 11.4. Если не обеспечивается достаточно широкое поле для обработки выходных данных, то каждое выходное значение печатается с такой шириной поля, которая необходима для его полного представления; это может вызывать сложности при чтении выходных данных.

Хороший стиль программирования

- 11.1. Используйте исключительно возможности языка C++ для организации ввода-вывода в программах, написанных на C++, хотя в этих программах доступен и стиль программирования языка С.
- 11.2. При выводе выражений помещайте их в круглых скобках, чтобы предотвратить возникновение каких-либо проблем, связанных с последовательностью выполнения операций, вычисляющих значения в выражениях, и операции <<.

Советы по повышению эффективности

- 11.1. Используйте неформатированный ввод-вывод для достижения максимальной эффективности при обработке файлов большого объема.

Замечания по технике программирования

- 11.1. Стиль программирования на языке C++ — это стиль, в котором предусмотрены средства сохранения типов данных
- 11.2. Язык C++ предоставляет возможность для стандартной обработки ввода-вывода встроенных типов данных и типов данных, определенных пользователем. Это свойство облегчает разработку программного обеспечения вообще и повторное его использование в частности.
- 11.3. Новые возможности ввода-вывода данных, тип которых определен пользователем, добавляются в C++ без модификации объявлений или элементов-данных закрытого типа в классе `ostream` и в классе `istream`. Это способствует расширяемости — наиболее привлекательной черте языка C++.

Упражнения для самопроверки

- 11.1. Заполнить пробелы в следующих утверждениях:

- a) Перегруженные операции потока часто определяются как _____ функции класса.
- b) Набор битов для выравнивания формата включает _____, _____ и _____.
- c) Ввод-вывод в C++ представляет собой обработку _____ битов.
- d) Параметризованные манипуляторы _____ и _____ могут использоваться для установки и сброса флагов состояния формата.
- e) Большая часть программ на C++ должна включать заголовочный файл _____, содержащий основную информацию, необходимую для всех операций ввода-вывода.
- f) Функции-элементы _____ и _____ используются для установки и сброса флагов состояния.
- g) Заголовочный файл _____ содержит информацию для выполнения форматированного ввода-вывода в память.

- h) При использовании параметризованных манипуляторов должен быть включен заголовочный файл _____.
- i) Заголовочный файл _____ содержит информацию для управления обработкой файлов.
- j) Манипулятор потока _____ осуществляет переход на новую строку в выходном потоке и сброс выходного потока.
- k) Заголовочный файл _____ позволяет использовать смешанный стиль программирования ввода-вывода языков С и С++.
- l) Функция-элемент _____ класса ostream используется для выполнения неформатированного вывода.
- m) Операции ввода поддерживаются классом _____.
- n) Вывод в стандартный поток ошибок направляется в объекты потоков _____ или _____.
- o) Операции вывода поддерживаются классом _____.
- p) Для операции поместить в поток используется символ _____.
- q) Четыре объекта, которые соответствуют стандартным устройствам системы, включают _____, _____, _____ и _____.
- r) Для операции взять из потока используется символ _____.
- s) Манипуляторы потока _____, _____ и _____ используются, чтобы задать восьмеричный, шестнадцатеричный и десятичный форматы представления целых чисел.
- t) По умолчанию точность для представления чисел с плавающей точкой равна _____.
- u) Установка флага _____ вызывает печать знака плюс для положительных чисел.

11.2. Укажите, что из нижеследующего верно, а что неверно. Если неверно, то объясните, почему.

- a) Функция-элемент потока flags() с аргументом типа long присваивает переменной состояния flags значение своего аргумента и возвращает ее прежнее значение.
- b) Операция поместить в поток << и операция взять из потока >> перегружены для обработки всех стандартных типов данных, включая строки, адреса памяти (только для операции поместить в поток) и все данные, тип которых определен пользователем.
- c) Функция-элемент потока flags() без аргументов производит сброс всех битов флагов в переменной состояния flags.
- d) Операция взять из потока >> может быть перегружена с помощью функции-операции, которая принимает в качестве параметров ссылку на istream, ссылку на определенный пользователем тип и возвращает ссылку на istream.
- e) Манипулятор потока ws обеспечивает пропуск ведущих (начальных) символов разделителей во входном потоке.
- f) Операция поместить в поток << может быть перегружена с помощью функции-операции, которая принимает в качестве параметра

ров ссылку на `istream`, ссылку на определенный пользователем тип и возвращает ссылку на `istream`.

g) При вводе с помощью операции взять из потока `>>` всегда происходит пропуск ведущих (начальных) символов разделителей во входном потоке.

h) Средства ввода-вывода — это составная часть C++.

i) Функция-элемент потока `rdstate()` возвращает состояние текущего потока.

j) Поток класса `cout` обычно связан с дисплеем.

k) Функция-элемент потока `good()` возвращает `true`, если все функции-элементы `bad()`, `fail()` и `eof()` возвращают `false`.

l) Поток класса `cin` обычно связан с экраном дисплея.

m) Если при операциях с потоком возникают неисправимые ошибки, функция-элемент `bad` возвращает `true`.

n) Вывод в `cerr` является небуферизованным, а вывод в `clog` является буферизованным.

o) Когда установлен флаг `ios::showpoint`, числа с плавающей запятой печатаются по умолчанию с точностью в шесть разрядов или печатаются с заданной точностью.

p) Функция-элемент `put` класса `ostream` выводит заданное число символов.

q) Манипуляторы потока `dec`, `oct` и `hex` оказывают воздействие только на следующую операцию вывода целого числа.

r) Адреса памяти при выводе отображаются по умолчанию как целые типа `long`.

11.3. Напишите по одному оператору, решающему следующие задачи:

a) Выведите строку "Введите ваше имя: ".

b) Установите флаг для вывода в верхнем регистре чисел в экспоненциальном формате и букв в шестнадцатеричном формате.

c) Выведите адрес переменной `string` типа `char *`.

d) Установите флаг печати чисел с плавающей запятой в экспоненциальном формате.

e) Выведите адрес переменной `integerPrt` типа `int *`.

f) Установите такой флаг, чтобы при выводе целых чисел на дисплее отображалось их основание при представлении в восьмеричном и шестнадцатеричном форматах.

g) Выведите значение типа `float`, на которое указывает `floatPtr`.

h) Используйте функцию-элемент потока, чтобы установить символ '*' в качестве заполняющего символа для печати с шириной поля, превышающей требуемую для печатаемого значения. Напишите отдельный оператор чтобы сделать то же самое с помощью манипулятора потока.

i) Выведите символы 'O' и 'K' одним оператором с помощью функции `put` класса `ostream`.

- j) Получите следующий символ из входного потока не удаляя его из потока.
- k) Введите один символ в переменную с типа `char` с помощью функции-элемента `get` класса `istream` двумя различными способами.
- l) Введите и отбросьте очередные шесть символов из входного потока.
- m) Используйте функцию-элемент `read` класса `istream` для ввода 50 символов в массив `line` типа `char`.
- n) Прочтите 10 символов в массив `name`. Прекратите чтение, если в потоке появится ограничитель '.'. Не удаляйте ограничитель из входного потока. Напишите другой оператор, который выполняет ту же задачу, но удаляет ограничитель из входного потока.
- o) Используйте функцию-элемент `gcount` класса `istream` для определения количества символов, введенных в символьный массив `line` последним вызовом функции-элемента `read` класса `istream`, и выведите это число символов, используя функцию-элемент `write` класса `ostream`.
- p) Напишите отдельные операторы для сброса выходного потока, использующие функцию-элемент `clear` и манипулятор потока `cout`.
- q) Выведите следующие значения: 124, 18.376, 'Z', 1000000 и "Строка".
- r) Напечатайте текущую установку точности с помощью функции-элемента `precision`.
- s) Введите целое число в переменную `months` типа `int` и число с плавающей запятой в переменную `percentageRate` типа `float`.
- t) Напечатайте 1.92, 1.925 и 1.9258 с точностью в три разряда, используя манипулятор.
- u) Напечатайте целое число 100 в восьмеричном, шестнадцатеричном и десятичном форматах с помощью манипуляторов потока.
- v) Напечатайте целое число 100 в десятичном, восьмеричном и шестнадцатеричном форматах, используя единственный манипулятор потока для изменения основания.
- w) Напечатайте 1234 с выравниванием по правой границе поля шириной 10 разрядов.
- x) Читайте символы в массив `line` до появления символа 'z' но не более 20 символов (включая и завершающий нулевой символ). Не удаляйте символ ограничитель из потока.
- y) Используйте целые переменные `x` и `y`, чтобы задать ширину поля и точность используемые для отображения значения 87.4573 типа `double` и выведите это значение на экран.
- 11.4. Найдите ошибку в каждом из приведенных ниже операторов и объясните, как ее можно исправить.
- a) `cout << "Значение x <= y равно:" << x <= y;`
- b) Следующий оператор должен печатать целое значение 'c'.
`cout << 'c';`

c) cout << "Строка в кавычках";

11.5. Для каждого из перечисленных ниже операторов, покажите, что будет выведено.

- a) cout << "12345" << endl;
 cout.width(5);
 cout.fill('*');
 cout << 123 << endl << 123;
- b) cout << setw(10) << setfill('\$') << 10000;
- c) cout << setw(8) << setprecision(3) << 1024.987654;
- d) cout << setioflags(ios::showbase) << oct << 99
 << endl << hex << 99;
- e) cout << 100000 << endl
 << setiosflags(ios::showpos) << 100000;
- f) cout << setw(10) << setprecision(2) <<
 << setiosflags(ios::scientific) << 444.93738;

Ответы на упражнения для самопроверки

11.1. a) дружественные. b) ios::left, ios::right, ios::internal. c) потоков.
 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, clog. o) ostream. p) <<. q) cin, cout,
 cerr, clog. r) >>. s) oct, hex, dec. t) 6 разрядам. u) ios::showpos.

11.2. a) Верно.

b) Неверно. Операции поместить в поток и взять из потока не перегружены для типов, определенных пользователем. Программист при создании собственного класса обязан специально создать перегруженные функции-операции, чтобы перегрузить операции с потоками для определенных им типов.

c) Неверно. Функция-элемент потока flag() без аргумента просто возвращает текущее значение переменной состояния flags.

d) Верно.

e) Верно.

f) Неверно. Для того, чтобы перегрузить операцию поместить в поток <<, перегруженная функция-операция должна получить в качестве параметров ссылку на класс ostream, ссылку на тип, определенный пользователем, и вернуть ссылку на класс ostream.

g) Верно, но до тех пор, пока флаг ios::skipws сброшен.

h) Неверно. Все средства ввода-вывода в C++ предоставляются стандартной библиотекой C++. В самом языке C++ не предусмотрены средства ввода, вывода и обработки файлов.

i) Верно.

j) Верно.

k) Верно.

- l) Неверно. Поток `cin` связан со стандартным устройством ввода компьютера, обычно с клавиатурой.
- m) Верно.
- n) Верно.
- o) Верно.
- p) Неверно. Функция-член `put` класса `ostream` выводит один символьный аргумент.
- q) Неверно. Манипуляторы потока `dec`, `oct` и `hex` устанавливают основание вывода целых чисел, которое действует до тех пор, пока основание не будет изменено или не завершится программа.
- r) Неверно. Адреса памяти отображаются по умолчанию в шестнадцатеричном формате. Для того, чтобы отобразить адреса как целые типов `long`, необходимо привести тип адресов к типу `long`.

11.3. a) `cout << "Введите ваше имя: "`

- b) `cout.setf(ios::uppercase);`
- c) `cout << (void *) string;`
- d) `cout.setf(ios::scientific, ios::floatfield);`
- e) `cout << integerPtr;`
- f) `cout << setiosflags(ios::showbase);`
- g) `cout << *floatPtr;`
- h) `cout.fill('*');`
`cout << setfill('*');`
- i) `cout.put('O').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 << "Строка";`
- r) `cout << 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) << 234;`
- x) `cin.get(line, 20, 'z');`

y) cout << setw(x) << setprecision(y) << 87.4573;

- 11.4. a)** Ошибка: приоритет операции << выше приоритета операции отношения <=, поэтому оператор вычисляется неправильно и это приводит к ошибке компилятора.

Исправление: для устранения этой ошибки следует заключить условное выражение ($x \leq y$) в круглые скобки. Подобные ошибки будут возникать с любыми выражениями, в которых используются операции с более низким приоритетом, чем приоритет операции <<, и которые не заключены в круглые скобки.

- b)** Ошибка: в отличие от языка С символы в языке C++ не обрабатываются как целые.

Исправление: чтобы напечатать численное значение символа из набора символов компьютера, символ должен быть приведен к целому значению, например, следующим образом:

cout << int('c');

- c)** Ошибка. Символы в виде кавычек не могут быть напечатаны в строке, пока не будет использована управляющая последовательность.

Поправка: Напечатайте строку одним из следующих способов:

```
cout << "" << "Строка в кавычках" << '';
cout << "\" Страна в кавычках\"";
```

- 11.5. a)** 12345
**123
123
- b)** \$\$\$\$\$\$100000
- c)** 1024.988
- d)** 0143
0x63
- e)** 100000
100000
- f)** 4.45e+02

Упражнения

- 11.6.** Напишите по одному оператору, выполняющему следующее:

- a) Напечатайте целое число 40000 с выравниванием по левой границе поля шириной 15 разрядов.
- b) Прочтите строку в переменную символьного массива state.
- c) Напечатайте число 200 со знаком и без него.
- d) Напечатайте десятичное значение 100 в шестнадцатеричном формате с предшествующими символами 0x.
- e) Читайте символы в массив s пока не встретится символ 'p', но не более 10 символов (включая завершающий нулевой символ). Извлеките указанный ограничитель из входного потока и отбросьте его.

- f) Напечатайте число 1.234 в виде значения с плавающей точкой с точностьюю 9 разрядов.
- g) Прочитайте строку "символы" из стандартного входного потока. Сохраните строку в символьном массиве *s*. Удаляйте кавычки из входного потока. Читайте максимум 50 символов (включая заключительный нулевой символ).
- 11.7. Напишите программу для проверки вводимых целых значений в десятичном, восьмеричном и шестнадцатеричном форматах. Выводите каждое прочитанное целое число во всех трех форматах. Проверьте программу со следующими входными данными: 10, 010, 0x10.
- 11.8. Напишите программу, которая печатает значения указателей, используя приведение их типа к типу целых чисел. Почему печатаются странные значения? Почему получаются ошибки?
- 11.9. Напишите программу для проверки результатов вывода на печать целого значения 12345 и значения с плавающей запятой 1.2345 в поля разной ширины. Что происходит, когда значения печатаются в полях, ширина которых меньше указанных значений?
- 11.10. Напишите программу, которая печатает значение 100.453627, округленное до ближайшего целого, до одной десятой, сотой, тысячной и десятитысячной.
- 11.11. Напишите программу, которая вводит строку с клавиатуры и определяет длину строки. Напечатайте строку, используя ее удвоенную длину в качестве ширины поля.
- 11.12. Напишите программу, которая преобразует температуру в целых числах по Фаренгейту от 0 до 212 градусов к значениям с плавающей запятой температуры по Цельсию с точностьюю до 3 знаков. Используйте для вычислений формулу
- ```
celsius = 5.0/9.0 * (fahrenheit - 32);
```
- Выходные данные должны быть отпечатаны в две колонки с выравниванием по правой границе поля, причем значения температуры по Цельсию должны содержать знак и перед положительными, и перед отрицательными температурами.
- 11.13. В некоторых языках программирования вводимые строки заключаются либо в одиночные кавычки (апострофы), либо в двойные кавычки. Напишите программу, которая читает три следующие строки suzy, "suzy" и 'suzy'. Игнорируются одиночные и двойные кавычки, или они читаются как часть строки?
- 11.14. На рис. 11.30 операции взять из потока и поместить в поток были перегружены для ввода и вывода объектов класса *PhoneNumber*. Перепишите операцию взять из потока так, чтобы она контролировала вводимые данные. Функция *operator>>* должна быть полностью переписана в соответствии со следующим алгоритмом:
- Ведите телефонный номер целиком в массив. Проверьте, что введено соответствующее число символов. Всего для телефонного номера должно быть прочитано 14 символов вида (800) 555-1212.

Используйте функцию-элемент потока `clear` для установки флага `ios::failbit` в случае неправильного ввода.

b) Код местности и коммутатор не должны начинаться с 0 или 1. Проверьте первую цифру в коде местности и коммутаторе, чтобы быть уверенными, что они не начинаются ни с 0, ни с 1. Используйте функцию-элемент потока `clear` для установки флага `ios::failbit` в случае неправильного ввода.

c) Средняя цифра кода местности всегда 0 или 1. Проверьте среднюю цифру на 0 и 1. Используйте функцию-элемент потока `clear` для установки флага `ios::failbit` в случае неправильного ввода. Если ни одна из приведенных выше операций не привела к установке флага `ios::failbit`, скопируйте три части телефонного номера в элементы `areaCode`, `exchange` и `line` объекта класса `PhoneNumber`. В главной программе прежде чем печатать телефонный номер надо проверить, не установлен ли `ios::failbit`, свидетельствующий о неправильном вводе. Если установлен, то программа должна напечатать сообщение об ошибке и не печатать номер телефона.

#### 11.15. Напишите следующую программу:

a) Создайте определенный пользователем класс `Point`, который содержит закрытые данные-элементы `xCoordinate` и `yCoordinate` и объявляет перегруженные функции-операции `взять из потока` и `поместить в поток` как дружественные функции класса.

b) Опишите функции-операции `взять из потока` и `поместить в поток`. Функция-операция `взять из потока` должна определять, являются ли вводимые данные правильными, и если нет, то она должна устанавливать индикатор неправильного ввода `ios::failbit`. Операция `поместить в поток` не должна выводить точку, если произошла ошибка ввода.

c) Напишите функцию `main`, которая проверяет ввод и вывод определенного пользователем класса `Point` с помощью перегруженных операций `взять из потока` и `поместить в поток`.

#### 11.16. Напишите следующую программу:

a) Создайте определенный пользователем класс `Complex`, который содержит закрытые данные-элементы `real` и `imaginary` и объявляет перегруженные функции-операции `взять из потока` и `поместить в поток` как дружественные функции класса.

b) Опишите функции-операции `взять из потока` и `поместить в поток`. Функция-операция `взять из потока` должна определять, являются ли вводимые данные правильными, и если нет, то она должна устанавливать индикатор неправильного ввода `ios::failbit`. Входные данные должны иметь форму:

$$3 + 8i$$

Значения могут быть как положительными, так и отрицательными, и одна из двух составляющих (действительная или мнимая часть) может отсутствовать. Если часть отсутствует, то соответствующий элемент данных должен быть задан равным 0. Операция `поместить в поток` не должна выводить, если произошла ошибка ввода. Фор-

мат вывода должен быть идентичен показанному выше формату ввода. Для отрицательных мнимых частей должен быть напечатан знак минус вместо знака плюс.

с) Напишите функцию `main`, которая проверяет ввод и вывод определенного пользователем класса `Complex` с помощью перегруженных операций взять из потока и поместить в поток

- 11.17. Напишите программу, которая использует структуру `for` для печати таблицы значений ASCII для набора символов ASCII с 33 по 126. Программа должна печатать десятичное, восьмеричное, шестнадцатеричное и символьное значения каждого символа. Используйте манипуляторы потока `dec`, `oct` и `hex` для печати целых значений.
- 11.18. Напишите программу, которая показывает, что каждая из функций-элементов `getline` и `get` с тремя аргументами заканчивает ввод строки конечным нулевым символом. Покажите также, что `get` оставляет символ ограничитель во входном потоке, а `getline` извлекает его из потока и отбрасывает. Что происходит с непрочитанными символами в потоке?
- 11.19. Напишите программу, которая создает определенный пользователем манипулятор `skipwhite` для пропуска лидирующих (начальных) символов разделителей во входном потоке. Манипулятор должен использовать функцию `isspace` из библиотеки `ctype.h` для проверки, не является ли символ символом разделителем. Каждый символ должен вводиться функцией-элементом `get` класса `istream`. Когда очередной символ оказывается не разделителем, манипулятор `skipwhite` должен заканчивать свою работу возвратом этого символа назад во входной поток и возвращением ссылки на `istream`.
- Проверьте этот манипулятор, написав функцию `main`, в которой флаг `ios::skipws` не установлен, так что операция взять из потока автоматически не пропускает символы разделители. Затем проверьте манипулятор на входном потоке, вводя сначала символы разделители, а затем значение символов. Печатайте вводимый символ, чтобы подтвердить, что символы разделители не вводятся.

г л а в а

---

# 12

## Шаблоны



### Ц е л и

- Научиться использовать шаблоны функций для создания группы однотипных (перегруженных) функций.
- Научиться различать шаблоны функций и шаблонные функции.
- Научиться использовать шаблоны классов для создания группы связанных типов классов.
- Научиться различать шаблоны класса и шаблонные классы.
- Понять, как перегружать шаблонные функции.
- Понять, как связаны между собой шаблоны, друзья, наследование и статические члены.

## План

- 12.1. Введение
- 12.2. Шаблоны функций
- 12.3. Перегрузка шаблонных функций
- 12.4. Шаблоны классов
- 12.5. Шаблоны классов и нетиповые параметры
- 12.6. Шаблоны и наследование
- 12.7. Шаблоны и друзья
- 12.8. Шаблоны и статические элементы

*Резюме • Терминология • Типичные ошибки программирования • Советы по повышению эффективности • Замечания по технике программирования  
• Упражнения для самопроверки • Ответы на упражнения для самопроверки  
• Упражнения*

### 12.1. Введение

В этой главе мы обсудим одно из последних нововведений стандарта языка C++ — *шаблоны*. Шаблоны дают нам возможность определять при помощи одного фрагмента кода целый набор взаимосвязанных функций (перегруженных), называемых *шаблонными функциями*, или набор связанных классов, называемых *шаблонными классами*.

Мы можем написать один *шаблон функции* сортировки массива, на основе которого C++ будет автоматически генерировать отдельные *шаблонные функции*, сортирующие массивы типов *int*, *float*, указателей на *char \** и т. д.

Мы обсуждали шаблоны функций в главе 3. Для удобства тех читателей, которые пропустили этот момент, мы обсудим их еще раз и рассмотрим пример.

Достаточно описать один *шаблон класса* стеков, а затем C++ будет автоматически создавать отдельные шаблонные классы типа стек целых, стек вещественных чисел, стек указателей типа `char *` и т. д.

Обратите внимание на различие между шаблонами функций и шаблонными функциями: шаблоны функций и шаблоны классов являются своего рода трафаретами, при помощи которых вычерчивают кривые; шаблонные функции и шаблонные классы можно сравнить с такими кривыми, имеющими одну и ту же форму, но отличающимися по цвету.

### **Замечание по технике программирования 12.1**

Шаблоны являются еще одной из многочисленных возможностей C++ по созданию более универсального программного обеспечения, которое можно использовать повторно.

В этой главе будут представлены примеры шаблонов функций и шаблонов классов. Мы также рассмотрим связь между шаблонами и такими свойствами C++, как перегрузка, наследование, дружественность и статические элементы классов.

Идея и детали механизма шаблонов обсуждаются здесь на основе работы Бьерна Страуструпта «Параметризованные типы в C++», изданной в трудах конференции USENIX C++, которая проходила в Денвере, штат Колорадо в октябре 1988 г.

## **12.2. Шаблоны функций**

Перегруженные функции обычно используются для выполнения *похожих* операций над различными типами данных. Если же для каждого типа данных должны выполняться *идентичные* операции, то более компактным и удобным решением является использование *шаблонов функций* — возможности, появившейся в C++ сравнительно недавно. При этом программист должен написать всего одно описание шаблона функции. Основываясь на типах аргументов, использованных при вызове этой функции, компилятор будет автоматически генерировать объектные коды функций, обрабатывающих каждый тип данных. В языке С эта задача выполнялась при помощи макросов, определяемых директивой препроцессора `#define`. Однако, при использовании макросов компилятор не выполняет проверку соответствия типов, из-за чего нередко возникают серьезные побочные эффекты. Шаблоны функций, являясь таким же компактным решением, как и макросы, позволяют однако компилятору полностью контролировать соответствие типов.

### **Замечание по технике программирования 12.2**

Шаблоны функций, как и макросы, позволяют создавать программное обеспечение, которое можно использовать повторно. Но в отличие от макросов шаблоны функций в C++ позволяют полностью контролировать соответствие типов.

Все описания шаблонов функций начинаются с ключевого слова `template`, за которым следует список формальных параметров шаблона, заключаемый в угловые скобки (`<` и `>`); каждому формальному параметру должно предшествовать ключевое слово `class`, как в следующих примерах:

```
template <class T>
```

или

```
template <class ElementType>
```

или

```
template <class BorderType, class FillType>
```

Формальные параметры в описании шаблона используются (наряду с параметрами встроенных типов или типов, определяемых пользователем) для определения *типов* параметров функции, типа возвращаемого функцией значения и типов переменных, объявляемых внутри функции. Далее, за этим заголовком, следует обычное описание функции. Заметьте, что ключевое слово `class`, используемое в шаблоне функции при задании типов параметров, фактически означает «любой встроенный тип или тип, определяемый пользователем».

### Типичная ошибка программирования 12.1

Ошибкаю является отсутствие в шаблоне ключевого слова `class` перед каждым формальным параметром типа.

Давайте рассмотрим в качестве примера шаблон функции `printArray`, приведенный на рис. 12.1. Этот шаблон используется в программе, приведенной на рис. 12.2.

В шаблоне функции `printArray` объявляется один формальный параметр `T` (вместо идентификатора `T` может быть использован любой другой допустимый идентификатор) для типа массива, который будет выводиться функцией `printArray`;

```
template<class T>
void printArray(T *array, const int count)
{
 for (int i = 0; i < count; i++)
 cout << array[i] << " ";
 cout << endl;
}
```

Рис. 12.1. Шаблон функции

Т называется *параметром типа*. Когда компилятор обнаруживает в тексте программы вызов функции `printArray`, он заменяет `T` во всей области определения шаблона на тип первого параметра функции `printArray` и C++ создает шаблонную функцию вывода массива указанного типа данных. После этого вновь созданная функция компилируется. В программе, приведенной на рис. 12.2, производится вызов трех функций `printArray`: одна выводит

массив типа `int`, другая выводит массив чисел с плавающей запятой и третья обрабатывает массив символов. Например, реализация функции для типа `int` будет выглядеть следующим образом:

```
void printArray(int *array, const int count)
{
 for (int i = 0; i < count; i++)
 cout << array[i] << " ";
 cout << endl;
}
```

Каждый формальный параметр из описания шаблона функции должен появиться в списке параметров функции по крайней мере один раз. Имя формального параметра может использоваться в списке параметров заголовка шаблона только один раз. Одно и то же имя формального параметра шаблона функции может использоваться несколькими шаблонами.

### Типичная ошибка программирования 12.2

Ошибка возникает, когда в списке параметров функции используются не все формальные параметры шаблона функции.

На рис. 12.2 приведена программа, использующая шаблон функции `printArray`. Программа начинается с определения массива `a` типа `int`, массива `b` типа `float` и массива `c` типа `char` с размерами соответственно 5, 7 и 6. Затем каждый из массивов выводится на экран при помощи вызова функции `printArray`: один раз с первым аргументом равным аргументу `a` типа `int *`, затем с аргументом `b` типа `float *` и в третьем вызове в качестве первого аргумента используется массив с типом `char *`. В результате вызова

```
printArray (a, aCount);
```

компилятор создает шаблонную функцию `printArray`, в которой вместо параметра типа `T` используется тип `int`. Вызов функции

```
printArray (b, bCount);
```

приводит к созданию второй шаблонной функции `printArray`, в которой параметр типа `T` заменяется на тип `float`. При обработке вызова функции

```
printArray (c, cCount);
```

компилятор создает третью шаблонную функцию `printArray`, для которой параметр типа `T` заменяется на тип `char`.

### Совет по повышению эффективности 12.1

Шаблоны несомненно расширяют возможности многократного использования программного кода. Но имейте в виду, что программа может создавать слишком много копий шаблонных функций и шаблонных классов. Для этих копий могут потребоваться значительные ресурсы памяти.

```
// Использование шаблонных 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 << endl;
}

main()
{
 const int aCount = 5, bCount = 7, cCount = 6;
 int a[aCount] = {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"; // шестая позиция
 // для нулевого указателя

 cout << "Массив а содержит:" << endl;
 printArray(a, aCount); // шаблонная функция
 // для целого типа

 cout << "Массив б содержит:" << endl;
 printArray(b, bCount); // шаблонная функция для типа float

 cout << "Массив с содержит:" << endl;
 printArray(c, cCount); // шаблонная функция для типа char

 return 0;
}
```

---

```
Массив а содержит:
1 2 3 4 5
Массив б содержит:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Массив с содержит:
H E L L O
```

**Рис. 12.2.** Пример использования шаблонных функций

## 12.3. Перегрузка шаблонных функций

Шаблонные функции и перегрузка функций тесно связаны друг с другом. Все родственные функции, полученные из шаблона, имеют одно и то же имя; поэтому компилятор использует механизм перегрузки для того, чтобы обеспечить вызов соответствующей функции.

Сам шаблон функции может быть перегружен несколькими способами. Мы можем определить другие шаблоны, имеющие то же самое имя функции, но различные наборы параметров. Например, шаблон функции `printArray` на рис. 12.2 может быть перегружен другим шаблоном функции `printArray`, который имеет дополнительные параметры `lowSubscript` и `highSubscript` для

определения той части массива, которая будет выводится (см. упражнение 12.4).

Шаблон функции может также быть перегружен, если мы введем другую не шаблонную функцию с тем же самым именем, но другим набором параметров функции. Например, шаблон функции `printArray` на рис. 12.2 мог бы быть перегружен версией не шаблонной функции, которая выводит элементы массива символьных строк в удобном табулированном виде, по столбцам (см. упражнение 12.5).

#### Типичная ошибка программирования 12.3

Если шаблон вызывается с определяемым пользователем типом в качестве параметра и если этот шаблон использует операции (например, `==`, `+`, `<=` и др.) с объектами этого типа, то такие операции должны быть перегружены! Если эти операции остались не перегруженными, то при редактировании связей будет выдано сообщение об ошибке, потому что компилятор, конечно, генерирует вызов соответствующих перегруженных функций-операций, не обращая внимание на то, что эти функции не определены.

Компилятор выполняет некий процесс согласования, чтобы определить, какой экземпляр функции соответствует данному вызову. Сначала компилятор пытается найти и использовать функцию, которая точно соответствует по своему имени и типам параметров вызываемой функции. Если на этом этапе компилятор терпит неудачу, то он ищет шаблон функции, с помощью которого он может генерировать шаблонную функцию с точным соответствием типов параметров и имени функции. Если такой шаблон обнаруживается, то компилятор генерирует и использует соответствующую шаблонную функцию. Обратите внимание, что компилятор ищет шаблон, полностью соответствующий вызываемой функции по типу всех параметров; автоматическое преобразование типов не производится. И в качестве последней попытки, компилятор последовательно выполняет процесс подбора перегруженной функции, как это описано в главе 3.

#### Типичная ошибка программирования 12.4

Компилятор подбирает вариант функции, соответствующий данному вызову. Если соответствующая функция не может быть найдена или если обнаружено несколько таких функций, то компилятор генерирует сообщение об ошибке.

## **12.4. Шаблоны классов**

Что такое стек (структура, в которую данные помещаются по одному и извлекаются в последовательности: последним вошел — первым вышел), можно понять, независимо от типа данных, помещаемых в стек. Тип данных должен быть задан только тогда, когда приходит время создать стек «фактически». Здесь мы опять имеем замечательную возможность создавать универсальное программное обеспечение, которое можно использовать повторно. Нам достаточно создать некое общее описание понятия стека и на основе этого родового класса создавать классы, являющиеся специфическими версиями для конкретного типа данных. Эта возможность обеспечивается в C++ *шаблонами классов*.

### Замечание по технике программирования 12.3

Применение шаблонов классов увеличивает возможности повторного использования программного обеспечения, когда классы для конкретного типа данных могут создаваться на основе родовой версии класса.

Шаблоны классов часто называют параметризованными типами, так как они имеют один или большее количество параметров типа, определяющих настройку родового шаблона класса на специфический тип данных при создании объекта класса.

Для того, чтобы использовать шаблонные классы, программисту достаточно один раз описать шаблон класса. Каждый раз, когда требуется реализация класса для нового типа данных, программист, используя простую краткую запись, сообщает об этом компилятору, который и создает исходный код для требуемого шаблонного класса. Шаблон класса **Stack**, например, может служить основой для создания многочисленных классов **Stack** необходимых программе типов (таких, например, как «**Stack** для данных типа **float**», «**Stack** для данных типа **int**», «**Stack** для данных типа **char**», «**Stack** для данных типа **Employee**» и т. д.).

Обратите внимание на описание шаблона класса **Stack** на рис. 12.3. Оно выглядит как традиционное описание класса, за исключением заголовка

```
template<class T>
```

указывающего на то, что это описание является шаблоном класса с параметром типа **T**, обозначающим тип классов **Stack**, которые будут создаваться на основе этого шаблона. Использовать именно идентификатор **T** не обязательно, можно использовать любой другой идентификатор. Идентификатор **T** определяет тип данных-элементов, хранящихся в стеке, и может использоваться в заголовке класса **Stack** и в функциях-элементах.

```
// TSTACK1.H
// Простой шаблон класса Stack
#ifndef TSTACK1_H
#define TSTACK1_H

#include <iostream.h>

template<class T>
class Stack {
public:
 Stack(int = 10); // конструктор с умолчанием
 // (размер стека равен 10)
 ~Stack() { delete [] stackPtr; } // деструктор
 int push(const T&); // помещение элемента в стек
 int pop(T&); // выталкивание элемента из стека
 int isEmpty() const { return top == -1; } // 1, если
 // стек пустой
 int isFull() const { return top == size - 1; } // 1, если стек
 // заполнен

private:
 int size; // размер стека
 int top; // положение вершины стека
 T *stackPtr; // указатель на стек
};
```

Рис. 12.3. Описание шаблона класса **Stack** (часть 1 из 4)

```

// Конструктор с умолчанием создает объект из 10 элементов
template<class T>
Stack<T>::Stack(int s)
{
 size = s > 0 && s < 1000 ? s : 10; // задается разумный
 // размер объекта
 top = -1; // Стек, изначально, пуст
 stackPtr = new T[size]; // выделение памяти
 // под элементы объекта Stack
}

// Помещение элемента в стек.
// В случае успеха возвращается 1, в противном случае - 0
template<class T>
int Stack<T>::push(const T &item)
{
 if (!isFull()) {
 stackPtr[++top] = item; // элемент помещается
 // в Stack
 return 1; // успешное завершение
 }
 return 0; // операция закончилась неудачей
}

// Выталкивание элемента из стека
template<class T>
int Stack<T>::pop(T &popValue)
{
 if (!isEmpty()) {
 popValue = stackPtr[top--]; // элемент выталкивается
 // из объекта Stack
 return 1; // успешное завершение
 // операции выталкивания
 }
 return 0; // операция закончилась неудачей
}
#endif

```

Рис. 12.3. Описание шаблона класса **Stack** (часть 2 из 4)

Теперь давайте рассмотрим программу драйвера (функцию **main**), в которой используется класс **Stack**. Программа начинается с объявления объекта **floatStack** размером 5 элементов. Этот объект объявляется как экземпляр класса **Stack<float>** (произносится как «**Stack** типа **float**»). При генерации исходного кода для класса **Stack** типа **float** компилятор заменит параметр типа **T** на **float**.

Затем программа последовательно размещает значения типа **float** 1.1, 2.2, 3.3, 4.4 и 5.5 в созданном объекте **floatStack**. Цикл размещения объектов функцией **push** завершится после того, как программа попытается поместить в стек шестое значение (стек уже заполнен, поскольку может вмещать максимум 5 элементов).

После этого программа поочередно выталкивает функцией **pop** все пять чисел из стека (в последовательности последним вошел — первым вышел). Программа пытается вытолкнуть и шестое значение, но так как **floatStack** уже пуст, то цикл выталкивания на этом завершается.

```
// FIG12_3.CPP
// Тестовая программа, использующая шаблон класса Stack

#include <iostream.h>
#include "tstack1.h"

main()
{
 Stack<float> floatStack(5);
 float f = 1.1;
 cout << "Размещение элементов в floatStack" << endl;

 while (floatStack.push(f)) { // в случае успеха возвращается 1

 cout << f << ' ';
 f += 1.1;
 }

 cout << endl << "Стек заполнен. Невозможно разместить "
 << f << endl << endl
 << "Выталкивание элементов из floatStack" << endl;

 while (floatStack.pop(f)) // в случае успеха возвращается 1
 cout << f << ' ';

 cout << endl
 << "Стек пуст. Больше ничего вытолкнуть невозможно"
 << endl;

 Stack<int> intStack;
 int i = 1;
 cout << endl << "Размещение элементов в intStack" << endl;

 while (intStack.push(i)) { // в случае успеха возвращается 1
 cout << i << ' ';
 i += 1;
 }

 cout << endl << "Стек заполнен. Невозможно разместить "
 << i << endl
 << endl << "Выталкивание элементов из intStack" << endl;

 while (intStack.pop(i)) // в случае успеха возвращается 1
 cout << i << ' ';

 cout << endl
 << "Стек пуст. Больше ничего вытолкнуть невозможно"
 << endl;
 return 0;
}
```

Рис. 12.3. Программа драйвер шаблона класса **Stack** (часть 3 из 4)

```
Размещение элементов в floatStack
```

```
1.1 2.2 3.3 4.4 5.5
```

```
Стек заполнен. Невозможно разместить 6.6
```

```
Выталкивание элементов из floatStack
```

```
5.5 4.4 3.3 2.2 1.1
```

```
Стек пуст. Больше ничего вытолкнуть невозможно
```

```
Размещение элементов в intStack
```

```
1 2 3 4 5 6 7 8 9 10
```

```
Стек заполнен. Невозможно разместить 11
```

```
Выталкивание элементов из intStack
```

```
10 9 8 7 6 5 4 3 2 1
```

```
Стек пуст. Больше ничего вытолкнуть невозможно
```

Рис. 12.3. Программа драйвер шаблона класса **Stack** (часть 4 из 4)

Затем, программа создает стек целого типа **intStack** при помощи объявления

```
Stack<int> intStack;
```

(произносится это как «**intStack** является объектом **Stack** элементов типа **int**»). Поскольку размер объекта не определен, используется значение размера по умолчанию, равное 10, как это определено в конструкторе с умолчанием. И опять в программе выполняются циклы помещения значений в **intStack**, пока он не заполнится, и выталкивания значений из **intStack**, пока он не станет пуст.

Каждое описания функции-элемента вне заголовка шаблона класса начинается с заголовка

```
template< class T>
```

Описание каждой из этих функций походит на стандартное описание функции за исключением того, что тип элементов класса **Stack** всегда указывается имеющим тип **T**. Чтобы связать каждое описание функции-элемента с областью действия шаблона класса, используется бинарная операция разрешения области действия с именем шаблона класса **Stack<T>**. В этом случае в качестве имени класса выступает **Stack<T>**. Когда создается объект **floatStack** типа **Stack<float>**, конструктор класса **Stack** создает массив элементов типа **float**, представляющий элементы данных стека. Оператор

```
stackPtr = new T[size];
```

в описании шаблона класса **Stack** замещается компилятором в шаблонном классе **Stack<float>** на оператор

```
stackPtr = new float[size];
```

### Типичная ошибка программирования 12.5

В отличие от не шаблонных классов, которые могут быть вложенными, шаблоны классов не могут вкладываться друг в друга. Попытка вложить один шаблон класса внутрь другого приводит к синтаксической ошибке.

## 12.5. Шаблоны классов и нетиповые параметры

Шаблон класса **Stack**, рассмотренный в предыдущем разделе, использовал только параметр типа в заголовке шаблона. Но в шаблонах имеется возможность использования и так называемых нетиповых параметров. Например, заголовок нашего шаблона можно модифицировать следующим образом, указав в нем нетиповой параметр **int elements**:

```
template<class T, int elements> // нетиповой параметр
```

Тогда, объявление типа

```
Stack<float, 100> mostRecentSalesFigures;
```

приведет к созданию (во время компиляции) шаблонного класса **Stack** с именем **mostRecentSalesFigures**, состоящего из 100 элементов данных типа **float**; этот шаблонный класс будет иметь тип **Stack<float, 100>**. В описании класса в разделе закрытых данных-элементов можно поместить следующее объявление массива

```
T stackHolder[elements]; // массив для размещения данных стека
```

### Совет по повышению эффективности 12.2

Если размер класса контейнера, например, массива или стека, может быть определен во время компиляции (например, при помощи нетипового параметра шаблона, указывающего размер), это устранит расходы на динамическое выделение памяти во время выполнения программы.

### Замечание по технике программирования 12.4

Определение размера класса контейнера во время компиляции (например, через нетиповой параметр шаблона) исключает возможность возникновения потенциально неисправимой ошибки во время выполнения программы, если оператору **new** не удастся получить необходимое количество памяти.

В упражнениях вы встретите задание, в котором нужно будет использовать нетиповой параметр, облегчающий создание шаблона для класса **Array**, который мы разработали в главе 8, «Перегрузка». Объекты класса **Arrga**, создаваемые на основе такого шаблона, получают необходимую память под свои элементы во время компиляции, вместо динамического выделения памяти во время выполнения программы.

Если для специфического типа данных нужен класс, который не соответствует общему шаблону класса, вы можете явно определить его, отменив тем самым действие шаблона для этого типа. Например, шаблон класса **Array** может использоваться для создания массивов любого типа. Однако, программист может «взять управление на себя» при создании класса **Array** для специфического типа, например, типа **Martian**. Для этого нужно просто создать новый класс с именем **Array<Martian>**.

## 12.6. Шаблоны и наследование

Шаблоны и наследование связаны друг с другом следующим образом:

- Шаблон класса может быть производным от шаблонного класса.
- Шаблон класса может являться производным от не шаблонного класса.
- Шаблонный класс может быть производным от шаблона класса.
- Не шаблонный класс может быть производным от шаблона класса.

## 12.7. Шаблоны и друзья

Мы уже знаем, что функции и целые классы могут быть объявлены друзьями не шаблонных классов. Для шаблонов классов также могут быть установлены отношения дружественности. Дружественность может быть установлена между шаблоном класса и глобальной функцией, функцией-элементом другого класса (возможно, шаблонного класса) или даже целым классом (возможно, шаблонным классом). Оформление этих отношений дружественности — дело не простое.

Если внутри шаблона класса **X**, объявленного как

```
template<class T> class X
```

находится объявление дружественной функции

```
friend void f1();
```

то функция **f1** является дружественной для каждого шаблонного класса, полученного из данного шаблона.

Если внутри шаблона класса **X**, объявленного как

```
template<class T> class X
```

находится объявление дружественной функции в форме

```
friend void f2(X<T> &);
```

то для конкретного типа **T**, например, **float**, дружественной для класса **X<float>** будет только функция **f2(X<float> &)**.

Внутри шаблона класса вы можете объявить функцию-элемент другого класса дружественной для любого шаблонного класса, полученного из данного шаблона. Для этого нужно использовать имя функции-элемента другого класса, имя этого класса и бинарную операцию разрешения области действия. Например, если внутри шаблона класса **X**, который был объявлен как

```
template<class T> class X
```

объявляется дружественная функция в форме

```
friend void A::f4();
```

то функция-элемент **f4** класса **A** будет дружественной для каждого шаблонного класса, полученного из данного шаблона.

Внутри шаблона класса **X**, который был объявлен следующим образом

```
template<class T> class X
```

объявление дружественной функции в виде

```
friend void C<T>::f5 (X<T> &);
```

для конкретного типа Т, например, float, сделает функцию-элемент

```
C<float>::f5 (X<float> &)
```

другом только шаблонного класса X<float>.

Внутри шаблона класса X, объявленного как

```
template<class T> class X
```

можно объявить другой, дружественный класс Y

```
friend class Y;
```

в результате чего, каждая из функций-элементов класса Y будет дружественной для каждого шаблонного класса, произведенного из шаблона класса X.

Если внутри шаблона класса X, который был объявлен как

```
template<class T> class X
```

объявлен второй класс Z в виде

```
friend class Z<T>;
```

то при создании шаблонного класса с конкретным типом Т, например, типом float, все элементы класса Z<float> будут друзьями шаблонного класса X<float>.

## 12.8. Шаблоны и статические элементы

В заключение скажем несколько слов относительно статических данных-элементов. Давайте вспомним, что в не шаблонных классах одна копия статического элемента данных используется всеми представителями класса и что статический элемент данных должен быть инициализирован в области действия файла.

Каждый шаблонный класс, полученный из шаблона класса, имеет собственную копию каждого статического элемента данных шаблона; все экземпляры этого шаблонного класса используют свой статический элемент данных. Как и статические элементы данных не шаблонного класса, статические элементы данных шаблонных классов должны быть инициализированы в области действия файла. Каждый шаблонный класс получает собственную копию статической функции-элемента шаблона класса.

### Резюме

- Шаблоны дают нам возможность определять при помощи одного фрагмента кода целый набор взаимосвязанных функций (перегруженных), называемых шаблонными функциями, или набор связанных классов, называемых шаблонными классами.
- Чтобы использовать шаблоны функций, программист должен написать всего одно описание шаблона функции. Основываясь на типах аргументов, использованных при вызове этой функции, компилятор будет автоматически генерировать объектные коды функций, обрабатывающих каждый тип данных.

- Все описания шаблонов функций начинаются с ключевого слова `template`, за которым следует список формальных параметров шаблона, заключаемый в угловые скобки (< и >); каждому формальному параметру должно предшествовать ключевое слово `class`. Ключевое слово `class`, используемое при определении типов параметров шаблона функции, фактически означает: «любой встроенный тип или тип, определяемый пользователем».
- Формальные параметры в описании шаблона используются для определения типов параметров функции, типа возвращаемого функцией значения и типов переменных, объявляемых внутри функции.
- Сам шаблон функции может быть перегружен несколькими способами. Мы можем определить другие шаблоны, имеющие то же самое имя функции, но различные наборы параметров. Шаблон функции может также быть перегружен, если мы введем другую не шаблонную функцию с тем же самым именем, но другим набором параметров.
- Шаблон класса представляет собой некое общее описание родового класса, на основе которого создаются специфические версии для конкретных типов данных.
- Шаблоны классов часто называют параметризованными типами, так как они имеют один или большее количество параметров типа, определяющих настройку родового шаблона класса на специфический тип данных при создании объекта класса.
- Для того, чтобы использовать шаблонные классы, программисту достаточно один раз описать шаблон класса. Каждый раз, когда требуется реализация класса для нового типа данных, программист, используя простую краткую запись, сообщает об этом компилятору, который и создает исходный код для требуемого шаблонного класса.
- Описание шаблона класса выглядит как традиционное описание класса, за исключением заголовка `template<class T>`, указывающего на то, что это описание является шаблоном класса с параметром типа `T`, обозначающим тип классов, которые будут создаваться на основе этого шаблона. Идентификатор `T` определяет тип данных-элементов, хранящихся в стеке, и может использоваться в заголовке класса и в функциях-элементах.
- Каждое описания функции-элемента вне заголовка шаблона класса начинается с заголовка `template<class T>`, за которым следует описание функции, подобное стандартному описанию, но как тип элемента класса всегда указывается параметр типа `T`. Чтобы связать каждое описание функции-элемента с областью действия шаблона класса, используется бинарная операция разрешения области действия с именем шаблона класса `ClassName<T>`.
- В шаблонах имеется возможность использовать так называемые нетиповые параметры.
- Для специфического типа можно определить класс, отменяющий действие шаблона класса для данного типа.
- Шаблон класса может быть производным от шаблонного класса. Шаблон класса может являться производным от не шаблонного класса. Шаблонный класс может быть производным от шаблона класса. Не шаблонный класс может быть производным от шаблона класса.

- Функции и целые классы могут быть объявлены друзьями не шаблонных классов. Для шаблонов классов также могут быть установлены отношения дружественности. Дружественность может быть установлена между шаблоном класса и глобальной функцией, функцией-элементом другого класса (возможно, шаблонного класса) или даже целым классом (возможно, шаблонным классом).
- Каждый шаблонный класс, полученный из шаблона класса, имеет собственную копию каждого статического элемента данных шаблона; все экземпляры этого шаблонного класса используют свой статический элемент данных. Как и статические элементы данных не шаблонного класса, статические элементы данных шаблонных классов должны быть инициализированы в области действия файл.
- Каждый шаблонный класс получает собственную копию статической функции-элемента шаблона класса.

## Терминология

|                                      |                                                |
|--------------------------------------|------------------------------------------------|
| <code>template&lt;class T&gt;</code> | статическая функция-член                       |
| аргумент шаблона                     | шаблона класса                                 |
| друзья шаблона                       | статическая функция-член                       |
| имя шаблона                          | шаблонного класса                              |
| имя шаблона класса                   | угловые скобки (< и >)                         |
| ключевое слово <code>class</code>    | формальный параметр                            |
| в шаблонном параметре типа           | заголовка шаблона                              |
| ключевое слово <code>template</code> | функция-член шаблонного класса                 |
| нетиповой параметр заголовка         | член данных <code>static</code> шаблона класса |
| шаблона                              | член данных <code>static</code> шаблонного     |
| объявление шаблона функции           | класса                                         |
| описание шаблона функции             | шаблон класса                                  |
| параметр типа в заголовке шаблона    | шаблон функции                                 |
| параметр шаблона                     | шаблонная функция                              |
| параметризованный тип                | шаблонный класс                                |
| перегрузка шаблонной функции         |                                                |

## Типичные ошибки программирования

- 12.1.** Ошибкой является отсутствие в шаблоне ключевого слова *class* перед каждым формальным параметром типа.
- 12.2.** Ошибка возникает, когда в списке параметров функции используются не все формальные параметры шаблона функции.
- 12.3.** Если шаблон вызывается с определяемым пользователем типом в качестве параметра и если этот шаблон использует операции (например, ==, +, <= и др.) с объектами этого типа, то такие операции должны быть перегружены! Если эти операции остались не перегруженными, то при редактировании связей будет выдано сообщение об ошибке, потому что компилятор, конечно, генерирует вызов соответствующих перегруженных функций-операций, не обращая внимание на то, что эти функции не определены.
- 12.4.** Компилятор подбирает вариант функции, соответствующий данному вызову. Если соответствующая функция не может быть найдена или

если обнаружено несколько таких функций, то компилятор генерирует сообщение об ошибке.

**12.5.** В отличие от нешаблонных классов, которые могут быть вложенными, шаблоны классов не могут вкладываться друг в друга. Попытка вложить один шаблон класса внутрь другого приводит к синтаксической ошибке.

### Советы по повышению эффективности

**12.1.** Шаблоны несомненно расширяют возможности многократного использования программного кода. Но имейте в виду, что программа может создавать слишком много копий шаблонных функций и шаблонных классов. Для этих копий могут потребоваться значительные ресурсы памяти.

**12.2.** Если размер класса контейнера, например, массива или стека, может быть определен во время компиляции (например, при помощи нетипового параметра шаблона, указывающего размер), это устранит расходы на динамическое выделение памяти во время выполнения программы.

### Замечания по технике программирования

**12.1.** Шаблоны являются еще одной из многочисленных возможностей C++ по созданию более универсального программного обеспечения, которое можно использовать повторно.

**12.2.** Шаблоны функций, как и макросы, позволяют создавать программное обеспечение, которое можно использовать повторно. Но в отличие от макросов шаблоны функций в C++ позволяют полностью контролировать соответствие типов.

**12.3.** Применение шаблонов классов увеличивает возможности повторного использования программного обеспечения, когда классы для конкретного типа данных могут создаваться на основе родовой версии класса.

**12.4.** Определение размера класса контейнера во время компиляции (например, через нетиповой параметр шаблона) исключает возможность возникновения потенциально неисправимой ошибки во время выполнения программы, если оператору `new` не удастся получить необходимое количество памяти.

### Упражнения для самопроверки

**12.1.** Определите, являются ли следующие утверждения истинными или ложными. Если утверждение ложно, объясните почему.

а) Дружественная функция шаблона функции должна быть шаблонной функцией.

б) Если несколько шаблонных классов произведены от одного и того же шаблона класса с единственным статическим элементом данных, то каждый из шаблонных классов совместно использует одну копию этого статического элемента данных.

- с) Шаблонная функция может быть перегружена другой шаблонной функцией с тем же самым именем.
- д) Имя формального параметра может быть использовано только один раз в списке формальных параметров описания шаблона. Имена формальных параметров шаблона должны быть уникальны среди всех описаний шаблонов.
- е) Ключевое слово `class`, используемое для параметров типа шаблона, означает: «любой определяемый пользователем тип класса».
- ф) Шаблонные классы не могут быть вложенными.

### 12.2. Заполните пропуски в каждом из следующих утверждений:

- а) Шаблоны дают нам возможность определить при помощи одного фрагмента кода группу связанных функций, называемых \_\_\_\_\_, или группу связанных классов, называемых \_\_\_\_\_.
- б) Все описания шаблонов функций начинаются с ключевого слова \_\_\_\_\_, за которым следует список формальных параметров шаблона функции, заключаемый в \_\_\_\_\_.
- с) Все функции, образованные из одного шаблона функции, имеют одно и то же имя, поэтому компилятор использует механизм \_\_\_\_\_ для того, чтобы обеспечить вызов соответствующей функции.
- д) Шаблоны классов также называются \_\_\_\_\_ типами.
- е) \_\_\_\_\_ операция \_\_\_\_\_ используется с именем шаблона класса, чтобы связать описание функции-элемента с областью действия шаблона класса.
- ф) Как и статические данные-элементы не шаблонных классов, статические данные-элементы шаблонов классов должны быть инициализированы в области действия \_\_\_\_\_.

### Ответы на упражнения для самопроверки

- 12.1. (а) Неверно. Это может быть и не шаблонная функция. (б) Неверно. Каждый шаблонный класс имеет свою копию статических данных-элементов. (с) Верно. (д) Неверно. Имена формальных параметров не обязательно должны быть уникальными среди шаблонов функций. е) Неверно. Ключевое слово `class` в этом контексте может также означать: «любой встроенный тип». (ф) Верно.
- 12.2. (а) шаблонными функциями, шаблонными классами. (б) `template`, угловые скобки (< и >). (с) перегрузки. (д) параметризованными. е) бинарная, разрешения области действия. (ф) файл.

### Упражнения

- 12.3. Напишите шаблон функции `bubbleSort` для программы сортировки на рис. 5.15. Напишите программу драйвер ввода, сортировки и вывода массивов типа `int` и `float`.

- 12.4. Перегрузите шаблон функции `printArray`, приведенный на рис. 12.2, добавив два дополнительных параметра, а именно, `int lowSubscript` и `int highSubscript`. В результате вызова этой функции должна выводиться только часть массива с индексами, ограниченными этими параметрами. Введите проверки допустимых значений `highSubscript` и `lowSubscript`. Если какой-то из них не лежит в допустимых пределах или если `highSubscript` имеет значение, не большее, чем значение `lowSubscript`, то перегруженная функция `printArray` должна возвращать нулевое значение; в противном случае `printArray` должна возвращать число выведенных элементов. После этого измените функцию `main`, чтобы проверить обе версии функций `printArray` на массивах `a`, `b` и `c`. Постарайтесь протестировать все возможности обеих версий `printArray`.
- 12.5. Перегрузите шаблон функции `printArray` (см. рис. 12.2) версией не шаблонной функции, выводящей массивы символьных строк в удобном табличном формате, по столбцам.
- 12.6. Напишите простой шаблон предикатной функции `isEqualTo`, которая сравнивает два своих параметра при помощи операции проверки равенства (`==`) и возвращает 1, если они равны, и 0, если не равны. Используйте этот шаблон функции в программе, которая вызывает `isEqualTo` с различными встроенными типами аргументов. Затем напишите отдельную версию программы, которая вызывает `isEqualTo` с определяемым пользователем типом и не перегруженной операцией равенства. Что случится, когда вы попытаетесь выполнить эту программу? Теперь перегрузите операцию равенства (используйте функцию-операцию `operator==`). Что получится, если теперь вы попытаетесь выполнить эту программу?
- 12.7. Используя нетиповой параметр `numberOfElements` и параметр типа `elementType`, создайте шаблон класса `Array`, который мы сконструировали в главе 8, «Перегрузка». С помощью этого шаблона будут создаваться экземпляры класса `Array` с заданным числом элементов указанного типа, определенным во время компиляции.
- 12.8. Напишите программу, в которой используется шаблон класса `Array`. Из этого шаблона может быть получен представитель класса `Array` для любого типа элемента. Переопределите шаблон явным описанием класса `Array` для типа `float` (`class Array<float>`). В программе драйвере создайте экземпляр класса `Array` типа `int` при помощи шаблона и покажите, что при создании экземпляра класса `Array` типа `float` используется явное описание класса: `class Array<float>`.
- 12.9. Объясните различие между шаблоном функции и шаблонной функцией.
- 12.10. Что можно сравнить с трафаретом, шаблоном класса или шаблонный класс? Аргументируйте ваш ответ.
- 12.11. Какова связь между шаблонами функций и перегрузкой?
- 12.12. Почему предпочтительнее использовать шаблоны функций, а не макросы?

- 12.13.** Как может отразиться на эффективности программы использование шаблонов функций и шаблонов классов?
- 12.14.** При вызове функции компилятор подбирает шаблонную функцию, соответствующую данному вызову. При каких обстоятельствах этот процесс подбора заканчивается ошибкой компиляции?
- 12.15.** Почему нередко шаблон класса называют параметризованным типом?
- 12.16.** Объясните, почему вы можете использовать оператор  
`Array<Employee> workerList (100);`  
в программе на C++ с шаблонными классами.
- 12.17.** Проверьте ваш ответ на упражнение 12.16. Объясните теперь, почему вы могли бы использовать определение  
`Array<Employee> workerList;`  
в программе на C++ с шаблонными классами.
- 12.18.** Объяснить использование следующей нотации в программе C++, использующей шаблонные классы:  
`template<class T> Array<T>::Array(int s)`
- 12.19.** Объясните, почему шаблоны для классов контейнеров типа массив или стек часто используют нетиповые параметры?
- 12.20.** Опишите, как нужно определить класс для специфического типа, чтобы переопределить шаблон класса именно для этого типа.
- 12.21.** Опишите связь между шаблонами класса и наследованием.
- 12.22.** Предположим, что шаблон класса имеет заголовок  
`template<class T1> class C1`  
Опишите отношения дружественности, возникающие, если внутри шаблона класса поместить приведенные ниже операторы, объявляющие друзей. Идентификаторы, начинающиеся с символа «*f*» являются функциями, идентификаторы, начинающиеся с символа «*C*» — классы, а идентификаторы с начальным символом «*T*» обозначают любой тип (т. е. встроенный тип или тип класса).  
a) `friend void f1();`  
b) `friend void f2(C1<T1> &);`  
c) `friend void C2::f4();`  
d) `friend void C3<T1>::f5(C1<T1> &);`  
e) `friend class C5;`  
f) `friend class C6<T1>;`
- 12.23.** Предположим, что шаблон класса `Employee` имеет статический элемент данных `count`. Предположим далее, что из этого шаблона класса получены три шаблонных класса. Сколько копий статического элемента данных будут иметься в этом случае? Как будет использоваться каждая из них (если они вообще будут существовать)?

# 13

## Обработка исключений



### Ц е л и

- Понять, что такое исключения и как они обрабатываются.
- Научиться использовать блоки **try** для обработки, которая должна выполняться при возникновении исключения.
- Научиться генерировать исключения в точке их возникновения.
- Научиться использовать блоки **catch** для описания обработчиков исключений.
- Понять, как обрабатываются неперехваченные и не предусмотренные исключения.

## План

- 13.1. Введение
- 13.2. Когда должна использоваться обработка исключений
- 13.3 Другие методы обработки ошибок
- 13.4. Основы обработки исключений в C++
- 13.5. Простой пример обработки исключений: деление на нуль
- 13.6. Блоки *try*
- 13.7. Генерация исключений
- 13.8. Перехват исключений
- 13.9. Повторная генерация исключений
- 13.10. Создание условного выражения
- 13.11. Спецификация исключений
- 13.12. Обработка непредусмотренных исключений
- 13.13. Конструкторы, деструкторы и обработка исключений
- 13.14. Исключения и наследование

*Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения*

### 13.1. Введение

В этой главе мы представляем одно из последних добавлений в язык C++, а именно, *обработку исключительных ситуаций* или, короче, *исключений*. Расширяемость C++ может приводить к увеличению количества и видов возможных ошибок. Каждый новый класс добавляет свои собственные возможные ошибки. Возможности языка, описываемые ниже, позволяют программистам писать более ясные, более живущие (робастные), более отказоустойчивые программы. Современные системы (такие, как операционная система Microsoft Windows NT), разрабатываемые с учетом этих и им подобных методов, приносят положительные результаты. Мы также укажем случаи, когда обработка исключений не должна использоваться.

Стиль и детали обработки исключений, представленные в этой главе, основаны на работе Эндрю Кенига (Andrew Koenig) и Бьерна Страуструпа (Bjarne Stroustrup), изложенной в их статье «Exception Handling for C++ (revised)», изданной в трудах Конференции Proceedings of the USENIX C++ Conference, проведенной в Сан-Франциско в апреле 1990 года. Их работа формулирует основы, которые, вероятно, станут вариантом стандарта ANSI по обработке исключений.

Обработка ошибок изменяется качественно и количественно в зависимости от того, предназначена ли данная прикладная программа для распространения, или нет. Изделия, предназначенные для распространения, как правило, предусматривают более тщательную обработку ошибок, чем «временный» программный продукт.

Существует много популярных способов борьбы с ошибками. Часто обработка ошибок распределяется по всему тексту программы. С ошибками работают в тех местах программы, где они могут появляться. Преимущество этого подхода в том, что программист, читающий текст программы, может видеть обработку ошибки в непосредственной близости от соответствующих операторов и определить, была ли осуществлена соответствующая проверка ошибки.

Проблема такого подхода состоит в том, что основной код в этом случае «загрязняется» обработкой ошибок. Для программиста, работающего с прикладной программой, становится труднее читать этот код и определять, правильно ли он функционирует. Это делает код более трудным для восприятия и поддержки.

Вот некоторые распространенные примеры исключений: нехватка памяти, выход индекса за пределы массива, арифметическое переполнение, деление на нуль, недопустимые параметры функций.

Новые возможности C++ по обработке исключений позволяют программисту вынести операторы обработки ошибок из «основной линии» выполнения программы. Это облегчает чтение программы и внесение в нее изменений. Выделение кода обработки ошибок из основной линии программы не противоречит достоинствам блочного построения, которые мы обсудили в контексте как структурного, так и объектно-ориентированного программирования.

Другое преимущество нового стиля обработки исключений заключается в том, что становится возможным перехватывать или все виды исключений, или только исключения некоторого определенного типа, или исключения взаимосвязанных типов. Это делает программы более устойчивыми к ошибкам, уменьшая вероятность того, что какие-то ошибки не будут перехвачены программой.

Обработка исключений осуществляется для того, чтобы дать возможность программе перехватить и обработать ошибку прежде, чем ошибка произойдет и наступят ее неприятные последствия. Если программист не обеспечивает средства обработки неисправимой ошибки, то при ее возникновении программа прекращает свою работу.

Обработка исключений создана для работы с так называемыми *синхронными ошибками* типа попытки деления на нуль (ошибка возникает, когда программа выполняет команду деления). При обработке исключений программа прежде, чем выполнить деление, проверяет знаменатель и генерирует (возбуждает или «выбрасывает», как говорят иногда в описаниях C++) исключение, если знаменатель равен нулю.

Обработка исключений не предназначена для работы с асинхронными ситуациями типа завершения операции ввода-вывода с диска, поступления сообщений от локальной сети, щелчка мыши и т.п.; эти ситуации лучше обрабатываются другими средствами, такими, как обработка прерываний.

Обработка исключений используется в случаях, при которых система может быть восстановлена для нормальной работы после ошибки, вызвавшей исключение. Процедура восстановления называется *обработчиком исключения*.

Обработка исключений обычно используется в случаях, при которых ошибка обрабатывается в другой части программы (то есть в другой области действия), чем та, в которой эта ошибка обнаружена. Например, программа, которая ведет интерактивный диалог с пользователем, не должна сама использовать исключения для обработки ошибок ввода.

### **Хороший стиль программирования 13.1**

Используйте исключения для ошибок, которые должны быть обработаны в другой области действия, отличной от той, где они происходят. Используйте другие средства для обработки ошибок в той области действия, в которой они происходят.

### **Хороший стиль программирования 13.2**

Избегайте использовать обработку исключений для целей, отличных от обработки ошибок, потому что это может уменьшить ясность программы.

Существует еще одна причина избегать использования методики обработки исключений для обычного программного управления. Механизм исключений создан для обработки ошибок, которые проявляются редко, и используется чаще всего в тех случаях, когда программа должна завершиться. Поэтому совершенно не обязательно, чтобы создатели трансляторов C++ при реализации обработки исключений заботились о достижении оптимальной эффективности, которую можно было бы ожидать при обычном выполнении программы.

### **Совет по повышению эффективности 13.1**

Хотя можно использовать обработку исключений для целей, отличных от обработки ошибок, но это может уменьшать эффективность программы.

### **Совет по повышению эффективности 13.2**

Обработка исключений в общем случае выполнена в трансляторах таким образом, что когда исключение не возникает, присутствие кода обработки исключений не вносит никаких дополнительных издержек (или они очень невелики). Когда же исключения возникают, они влекут за собой накладные расходы во время выполнения.

### **Замечание по технике программирования 13.1**

Поток управления со стандартными управляющими структурами вообще более ясен и более эффективен, чем применение исключений.

### **Типичная ошибка программирования 13.1**

Причина, по которой исключения могут быть опасными как альтернатива нормальному потоку управления, заключается в том, что стек может не разматываться и ресурсы, распределенные до возникновения исключения, могут не освобождаться. Этой проблемы можно избежать тщательным программированием.

Обработка исключений улучшает отказоустойчивость программы. Становится «более приятным» писать обработку ошибок, так что программисты чаще делают это. Становится также возможным перехватывать исключения самыми различными путями, например, по типу, или даже определять, что должны быть перехвачены исключения любых типов. Обработка исключений обеспечивает программиста богатым дисциплинирующим набором возможностей обработки ошибок.

Важно обратить внимание на то, что огромное большинство программ, написанных сегодня, поддерживает только одну линию выполнения задачи. Однако в современных операционных системах, таких, как Windows NT, OS/2 и различные версии UNIX, уделяется все большее внимание многопоточному выполнению задач. Методы, обсуждаемые в этой главе, применяются также для многопоточных программ, хотя мы и не обсуждаем такие программы специально.

### **Замечание по мобильности 13.1**

Как уже было сказано, стандарт ANSI для C++ еще не завершен и возможно, что особенности работы с исключениями, описанные в этой главе, могут еще развиваться далее в течение следующих нескольких лет. Когда мы завершали написание этой книги, главные продавцы трансляторов C++ начали поставлять версии с обработкой исключений. Опыт, полученный при работе с ними в течение нескольких следующих лет, должен помочь в продвижении к стандарту.

Мы покажем, как обращаться с «неперехваченными» исключениями. Мы рассмотрим также, как обрабатываются непредусмотренные исключения. Мы покажем, как взаимосвязанные исключения могут быть представлены классами исключений, полученными из общего базового класса исключений.

Новые возможности обработки исключений в C++, вероятно, будут широко использоваться в результате усилий по стандартизации C++ со стороны ANSI. Такая стандартизация особенно важна в больших проектах по созданию программного обеспечения, в которых десятки или даже сотни людей работают над проектированием отдельных компонентов системы и эти компоненты должны быть эффективно объединены в единую полную систему и при этом правильно функционировать.

### **Замечание по технике программирования 13.2**

Обработка исключений особенно полезна для систем с отдельно разработанными компонентами. Это типично для больших программных комплексов. Обработка исключений позволяет проще объединять компоненты. Каждый компонент может сам осуществлять обнаружение исключительной ситуации отдельно от ее обработки.

Обработка исключений может рассматриваться как еще один способ возврата управления из функции или выхода из блока. Как правило, когда

генерируется исключение, оно будет обработано источником вызова данной функции, сгенерировавшей это исключение, источником вызова этого вызова и так далее, как бы далеко назад ни надо было пройти по цепочке обращений, чтобы найти обработчик этого исключения.

## 13.2. Когда должна использоваться обработка исключений

Обработка исключений должна использоваться:

- чтобы обрабатывать только исключительные ситуации, несмотря на то, что ничто не мешает программисту использовать исключения как альтернативу программного управления.
- чтобы обрабатывать исключения, возникающие в тех компонентах программы, которые сами не имеют механизма обработки этих исключений.
- чтобы обрабатывать исключения, возникающие в таких компонентах программы, как функции, библиотеки и классы, которые широко используются и в которых не имеет смысла вводить собственную обработку исключений.
- в больших проектах, чтобы обрабатывать одинаковым способом ошибки, возникающие в различных местах проекта.

### Хороший стиль программирования 13.3

Используйте обычные методы обработки ошибок (а не обработку исключений) для построения непосредственного, локального обработчика ошибок, в котором программе легче управляться со своими собственными ошибками.

### Замечание по технике программирования 13.3

Когда идет работа с библиотеками, источник вызова библиотечной функции будет, вероятно, использовать свой уникальный обработчик исключения, сгенерированного библиотечной функции. Маловероятно, чтобы библиотечная функция выполняла такую обработку ошибки, которая удовлетворила бы конкретные потребности всех пользователей. Следовательно, исключения – удобный способ работы с ошибками, возникающими в библиотечных функциях.

## 13.3. Другие методы обработки ошибок

Мы уже рассматривали ранее ряд вариантов работы в исключительных ситуациях:

- Использование утилиты `assert` для проверки ошибок разработки. Если проверяемое этой утилитой утверждение неправильно, программа завершается и данный фрагмент программы должен быть исправлен программистом. Это полезно во время отладки.
- Простое игнорирование исключений. Это недопустимо в программах, предназначенных для широкого распространения, а также в целевом

программном обеспечении, необходимом для работы в критических ситуациях. Но для вашего собственного программного обеспечения, создаваемого для каких-то ваших частных задач совершенно нормально игнорировать многие виды ошибок.

- Прерывание программы. Это не дает возможность программе закончиться нормально и при этом выдать неправильные результаты. Фактически, для многих типов ошибок прерывание — хорошая стратегия, особенно для исправимых ошибок, которые дают возможность программе доработать до конца и вводят в заблуждение программиста, считающего, что программа функционировала правильно. Однако, такая стратегия тоже не подходит для ответственных приложений. Здесь также важны проблемы ресурсов. Если программа получила некоторые ресурсы, она, как правило, должна возвратить их до своего завершения.

### Типичная ошибка программирования 13.2

Прерывание выполнения программы может оставить ресурс в таком состоянии, в котором другие программы не могут его использовать, и мы получаем так называемую «утечку ресурса».

- Установка некоторого индикатора ошибки типа `errno`. Однако проблема в том, что программа может не проверять эти индикаторы ошибки во всех тех точках, в которых эти ошибки могут возникать.
- Проверка условия ошибки, выдача сообщения об ошибке и вызов `exit` для передачи соответствующего кода ошибки среди выполнения программы.
- `setjmp` и `longjmp`. Эти возможности, доступные через `<setjmp.h>`, позволяют программисту, минуя глубоко вложенные вызовы функций, задать непосредственный переход к обработчику ошибки. Без `setjmp/longjmp` программа должна выполнить несколько возвратов, чтобы выйти из глубоко вложенных вызовов функций. `setjmp` и `longjmp` могут, конечно, использоваться для перехода к обработчику ошибки. Но они опасны в C++, поскольку они разматывают стек без вызовов деструкторов автоматических объектов. Это может вести к ряду серьезных проблем.
- Выделение некоторых специфических видов ошибок для их обработки. Например, когда операция `new` дает сбой при выделении памяти, она может заставить выполняться функцию `new_handler` для обработки этой ошибки. Эту функцию можно менять, передавая имя функции в качестве аргумента `set_new_handler`.

## 13.4. Основы обработки исключений в C++

В этом разделе мы кратко рассмотрим основы программирования обработки исключений. В последующих разделах мы исследуем обработку исключений более детально.

Обработка исключений в C++ подключается в ситуации, в которой функция обнаруживает ошибку, но не способна сама с ней управиться. Такая функция *генерирует исключение* (или, как иногда говорят, *возбуждает* или

выбрасывает исключение). При этом нет никаких гарантий, что где-то вне этой функции существует нечто (обработчик исключения), запускающее обработку именно этого вида исключения. Если существует, то исключение будет перехвачено и обработано. Если же для этого специфического вида исключения не имеется никакого обработчика, программа завершается.

Программист заключает в так называемый блок *try* (блок испытания) тот код, который может сгенерировать ошибку, возбуждающую исключение. Следом за блоком *try* записывается один или более блоков *catch* (блоки перехвата). Каждый блок *catch* определяет тип исключений, которые он может перехватывать и обрабатывать. Каждый блок *catch* содержит программу обработки — обработчик исключения.

Если исключительная ситуация соответствует типу параметра в одном из блоков *catch*, выполняется код этого блока. В противном случае вызывается функция *terminate* (завершение), которая по умолчанию вызывает функцию *abort* (аварийное завершение работы).

Программное управление при генерации исключения осуществляет выход из блока *try* и последовательный поиск в блоках *catch* соответствующего обработчика (мы скоро обсудим то, что делает обработчик «соответствующим»). Если никакие исключения в блоке *try* не генерируются, обработчики исключений для такого блока пропускаются и программа продолжает выполняться с операторов, следующих за последним блоком *catch*.

Мы можем указать те исключения, которые могут генерироваться конкретной функцией. В частном случае мы можем определить, что функция вообще не генерирует никаких исключений.

Исключение генерируется в функции блока *try* или в функции, вызываемой непосредственно или косвенно из блока *try*.

Точка, в которой выполняется генерация (*throw*), называется *точкой генерации*. Этот термин также используется для описания самого выражения *throw*.

После того, как исключение сгенерировано, управление не может возвратиться в точку генерации.

Когда возникает исключение, можно передать информацию из точки генерации обработчику этого исключения. Это информация о типе самого сгенерированного объекта или информация, помещенная в этот объект.

Генерируемый объект обычно является строкой символов (для сообщения об ошибке) или объектом классом. Генерируемый объект передает информацию обработчику исключения.

#### Замечание по технике программирования 13.4

Ключевым моментом в стиле обработки исключений в C++ является то, что часть программы или системы, которая будет обрабатывать исключение, может быть совершенно отделена и удалена от той части программы, которая обнаружила и возбудила исключение.

### **13.5. Простой пример обработки исключений: деление на нуль**

Давайте теперь рассмотрим простой пример обработки исключений. Программа на рис. 13.1 использует операторы *try*, *throw* и *catch*, чтобы обнаружить, указать и обработать исключение деления на нуль.

```

// Простой пример обработки исключения.
// Контроль ошибки деления на ноль.
#include <iostream.h>

//Определение класса DivideByZeroError, используемого при
//обработке исключения, генерируемого делением на нуль.
class DivideByZeroError {
public:
 DivideByZeroError (): message ("Деление на нуль") { }
 void printMessage () const {cout << message;}
private:
 const char *message;
};

// Описание функции quotient. Используется, чтобы показать
//генерацию исключения при ошибке деления на нуль.
float quotient (int num1, int num2)
{
 if (num2 == 0)
 throw DivideByZeroError ();

 return (float) num1 / num2;
}

```

**Рис. 13.1.** Простой пример обработки исключения деления на нуль (часть 1 из 2)

```

//Управляющая программа
main ()
{
 cout << "Введите два целых числа для расчета их частного: ";

 int number1, number2;
 cin >> number1 >> number2;

 try { // включает код, который может сгенерировать исключение
 float result = quotient (number1, number2);

 cout << "Частное равно " << result << endl;
 }

 catch (DivideByZeroError error) { // обработчик ошибки
 cout << "ОШИБКА: ";
 error.printMessage ();
 cout << endl;
 }

 return 1; // завершение при ошибке
}

return 0; // нормальное завершение
}

```

---

Введите два целых числа для расчета их частного: 7 3  
Частное равно 2.333333

Введите два целых числа для расчета их частного: 23 0  
ОШИБКА: Деление на нуль

**Рис. 13.1.** Простой пример обработки исключения деления на нуль (часть 2 из 2)

Рассмотрим два приведенных результата работы программы. Первый иллюстрирует успешное выполнение. Во втором введен знаменатель, равный нулю; программа обнаруживает эту ошибку и выдает о ней сообщение.

Теперь рассмотрите программу драйвер `main`. Программа предлагает ввести два целых числа. Обратите внимание на локализованное объявление переменных `number1` и `number2`.

Далее программа выполняет блок `try`, включающий код, который может возбудить исключение. Заметьте, что фактическое деление, которое может вызывать ошибку, не присутствует явно в блока `try`. Вызывается функция `quotient`, содержащая операцию деления. Как мы скоро увидим, функция `quotient` создает объект исключение деления на нуль. В общем случае ошибка может появляться при выполнении операторов, явно упомянутых в блоке `try`, или в вызовах функций или даже в глубоко вложенных вызовах функций, инициализированных оператором из блока `try`.

После блока `try` записан блок `catch`, содержащий обработчик исключения для ошибки деления на нуль. Вообще, когда исключение генерируется внутри блока `try`, оно перехватывается блоком `catch`, в котором определен тип, соответствующий сгенерированному исключению. В программе на рис. 13.1 определено, что блок `catch` будет перехватывать объекты исключения типа `DivideByZeroError`; этот тип соответствует типу объекта исключения, генерируемого в функции `quotient`. Тело данного обработчика исключения просто выдает сообщение об ошибке и возвращает 1, что указывает на прерывание выполнения из-за ошибки. Но обработчики исключений могут быть, конечно, намного сложнее, чем этот.

Если, в процессе выполнения код в блоке `try` не генерирует исключение, тогда все обработчики `catch`, следующие за блоком `try`, пропускаются и управление передается первой строке после блоков `catch`; в программе на рис. 13.1 выполняется оператор возврата `return`, который возвращает 0, указывая на нормальное завершение работы.

Теперь давайте исследуем описания класса `DivideByZeroError` и функции `quotient`. В функции `quotient`, когда условный оператор `if` определяет, что знаменатель равен нулю, выполняется оператор `throw`, который специфицирует имя конструктора объекта исключения. В результате создается объект класса `DivideByZeroError`. Этот объект будет перехвачен оператором `catch` (в котором специфицирован тип `DivideByZeroError`), расположенным после блока `try`. Конструктор класса `DivideByZeroError` просто копирует строку «Деление на нуль» в закрытый элемент данных `message`. Сгенерированный объект принимается как параметр в обработчике `catch` (в данном случае, как параметр `error`) и сообщение печатается путем обращения к открытой функции `printMessage`.

#### Хороший стиль программирования 13.4

Избегайте имени `Exception` для любого класса исключения. Весьма вероятно, что это имя используется библиотеками, и возможно даже будет включено в развивающийся стандарт ANSI для C++.

#### Хороший стиль программирования 13.5

Привязка каждого типа ошибки времени выполнения к соответственно названному объекту исключения улучшает ясность программы.

## 13.6. Блоки try

Исключения, которые генерируются в блоке `try`, обычно перехватываются обработчиком, описанным в блоке `catch`, следующем непосредственно за этим блоком `try`.

```
try {
 ...
}

catch (...) {
 ...
}
```

За блоком `try` может не следовать ни одного блока `catch`, или может следовать несколько таких блоков. Если при выполнении блока `try` не генерируется ни одно исключение, все обработчики исключений пропускаются и управление передается первому оператору после последнего обработчика.

## 13.7. Генерация исключений

Ключевое слово `throw` используется для того, чтобы указать, какое исключение генерируется. Это называется *генерацией исключения* или *возбуждением исключения*. Обычно `throw` имеет один операнд (специальный случай без операндов мы обсудим отдельно). Операнд `throw` может быть любого типа. Если операнд является объектом, мы называем его *объектом исключения*. Вместо объекта может быть сгенерировано условное выражение (см. раздел 13.10). Можно также генерировать объекты, не предназначенные для обработки ошибок.

Где перехватывается исключение? После генерации исключение будет перехвачено ближайшим обработчиком исключений (ближайшим к блоку `try`, в котором было сгенерировано исключение), содержащим спецификацию соответствующего типа. Обработчики исключений для блока `try` перечисляются сразу после него.

В процессе генерации исключения создается и инициализируется временная копия операнда `throw`. Этот временный объект затем инициализирует параметр в обработчике исключения. Временный объект уничтожается, когда завершается выполнение обработчика исключения и управление передается программе.

### Замечание по технике программирования 13.5

Если необходимо передать информацию относительно ошибки, которая вызвала исключение, то такая информация может быть помещена в сгенерированный объект. Обработчик `catch` должен содержать в этом случае имя параметра, через который эта информация может быть принята.

### Замечание по технике программирования 13.6

Может быть сгенерирован объект, не содержащий информацию для передачи; в этом случае обработчику достаточно простой информации о том, что сгенерировано исключение данного типа, чтобы правильно выполнить свою задачу.

Когда исключение сгенерировано, программное управление покидает текущий блок `try` и передается соответствующему обработчику `catch` (если он существует), расположенному после этого блока `try`. Возможно, что точка генерации расположена в глубоко вложенной области действия внутри блока `try`; управление все равно будет передано этому обработчику `catch`. Возможно также, что точка генерации расположена в глубоко вложенных вызовах функций; и в этом случае управление будет передано этому обработчику `catch`.

Может оказаться, что сам блок `try` не содержит никаких проверок ошибок и не включает никаких операторов `throw`, но код, вызываемый из блока `try`, может, конечно, содержать контроль ошибок, в частности, в конструкторах. Например, код в блоке `try` может обрабатывать индексирование массива в объекте класса `массив`, в котором функция-элемент `operator[]` может быть перегружена генерацией исключения, связанного с ошибкой выхода индекса за допустимые пределы. Любое обращение к этой функции может сгенерировать исключение или вызвать другую функцию, которая тоже может сгенерировать исключение.

Исключение может прервать выполнение программы, но это не обязательно. Однако, выполнение блока, в котором сгенерировано исключение, завершается.

#### Типичная ошибка программирования 13.3

Исключение должно генерироваться только внутри блока `try`. Исключение сгенерированное вне блока `try`, вызывает обращение к `terminate` – прерыванию программы.

## 13.8. Перехват исключений

Обработчики исключений содержатся в блоках `catch`. Каждый блок `catch` начинается с ключевого слова `catch`, за которым следуют круглые скобки, содержащие тип и необязательное имя параметра. Затем в фигурных скобках записываются операторы обработки исключения. Когда исключение перехвачено, начинает выполняться программа в блоке `catch`.

#### Типичная ошибка программирования 13.4

Предположение, что после обработки исключения управление вернется к первому оператору после того, который сгенерировал это исключение.

Обработчик `catch` определяет свою собственную область действия. Обработчик специфицирует в круглых скобках тип объекта, который должен быть перехвачен. Параметр в обработчике `catch` может быть поименован или нет. Если параметр поименован, на него можно ссылаться в обработчике. Если параметр не назван, а указан только тип соответствующего объекта исключения, то информация из точки генерации обработчику не передается; передается только управление из точки генерации в обработчик. Этого достаточно для большинства исключений.

#### Типичная ошибка программирования 13.5

Задание разделяемого запятыми списка аргументов `catch`.

Исключение, у которого тип сгенерированного объекта соответствует типу аргумента в заголовке **catch**, вызывает выполнение программы обработки этого блока **catch**.

Исключение перехватывается первым обработчиком **catch**, следующим за активным в настоящее время блоком **try** и соответствующим типу сгенерированного объекта. Правила соответствия уже были коротко рассмотрены.

Исключение, которое не перехвачено, вызывает **terminate**, что по умолчанию приводит к вызову **abort** для аварийного завершения программы. Можно изменить это поведение, определив другую функцию, которая должна будет выполняться; имя этой функции надо задать как аргумент в обращении к функции **set\_terminate**.

Если после **catch** в круглых скобках записано многоточие:

```
catch (...)
```

это означает, что будут перехватываться все исключения.

### Типичная ошибка программирования 13.6

Размещение **catch (...)** перед другими блоками **catch** препятствует выполнению всех других обработчиков; **catch (...)** всегда должен размещаться последним в списке обработчиков после блока **try**, иначе будет зафиксирована синтаксическая ошибка.

### Замечание по технике программирования 13.7

Недостаток перехвата исключений с помощью **catch (...)** заключается в том, что вы обычно не можете знать, каков тип исключения. Другой недостаток заключается в том, что без поименованного параметра не существует никакого способа обратиться в обработчике исключения к объекту исключения.

Возможно, что некоторому сгенерированному объекту не будет соответствовать ни один обработчик. Это вызывает продолжение поиска соответствия в следующем внешнем блоке **try**, включающем данный. При продолжении этого процесса в конечном счете может быть выяснено, что в программе не существует обработчика, который соответствовал бы типу сгенерированного объекта; в этом случае вызывается функция **terminate**, которая по умолчанию вызывает функцию **abort**.

### Замечание по технике программирования 13.8

Программист определяет последовательность, в которой перечисляются обработчики исключений. Эта последовательность может влиять на то, как обрабатываются исключения, возникающие в этом блоке **try**.

Обработчики исключений поочередно просматриваются в поисках соответствующего типа. Выполняется первый обработчик соответствующего типа. Когда он завершает свою работу, управление передается на первый оператор после последнего блока **catch**, то есть на первый оператор после последнего обработчика исключений этого блока **try**.

Возможно, что данному типу исключения будут соответствовать несколько обработчиков исключений. В этом случае выполняется первый обработчик, соответствующий типу исключения. Если данному типу соответствуют несколько обработчиков и если они отличаются друг от друга, то последова-

тельность записи обработчиков будет влиять на способ, которым эти исключения будут обрабатываться.

Может оказаться, что несколько обработчиков `catch` содержат тип класса, который соответствует типу конкретного сгенерированного объекта. Это может случиться по нескольким причинам. Во-первых, может существовать обработчик `catch (...)`, который перехватит любое исключение. Во-вторых, из-за иерархии наследования может оказаться, что объект производного класса может быть перехвачен как обработчиком данного класса, так и обработчиком любого базового класса, от которого порожден данный класс.

### Типичная ошибка программирования 13.7

Размещение `catch`, который перехватывает объект базового класса, перед `catch`, который перехватывает объект класса, производного от данного базового, является синтаксической ошибкой. Перехватчик `catch` базового класса перехватит все объекты производных классов, так что `catch` производного класса никогда не будет выполняться.

Иногда программа может обрабатывать многие близко связанные типы исключений. Вместо того, чтобы обеспечивать каждое исключение отдельным классом и обработчиком `catch`, программист может создать один класс исключения и один обработчик `catch` для группы исключений. При возникновении каждого из таких исключений может создаваться один объект исключения с различными закрытыми данными. Обработчик `catch` может просматривать эти закрытые данные, чтобы различить типы исключений.

Как фиксируется соответствие типа? Тип параметра в заголовке `catch` соответствует типу сгенерированного объекта, если:

- они имеют действительно одинаковый тип;
- тип параметра обработчика `catch` является открытым базовым классом класса сгенерированного объекта;
- параметр обработчика имеет тип указатель и сгенерированный объект тоже имеет тип указатель, преобразуемый в тип параметра путем допустимых преобразований указателей. Например, указатель производного класса преобразуется в указатель базового класса стандартными операциями преобразования;
- обработчик `catch` записан в форме `catch (...)`.

### Типичная ошибка программирования 13.8

Размещение обработчика исключения с типом аргумента `void *` перед обработчиками исключений с другими типами указателей вызывает синтаксическую ошибку. Обработчик `void *` будет перехватывать все исключения типа указатель, так что другие обработчики никогда не будут выполняться.

Возможно, что хотя имеется обработчик с точным соответствием типа, будет использовано соответствие, требующее стандартных преобразований, потому что этот обработчик встретится ранее того, который обеспечивает точное соответствие.

Можно сгенерировать объекты исключений `const` и `volatile`. В этом случае тип параметра обработчика `catch` также должен быть объявлен как `const` или `volatile` соответственно.

По умолчанию, если никакой обработчик для исключения не найден, программа завершается. Хотя это может показаться оправданным, программисты не обязаны этому следовать. Чаще при возникновении ошибки продолжается выполнение программы, возможно только несколько «прихрамывающее».

Обработчики исключений, следующие за блоком `try`, напоминают оператор `switch`. Интересно, что в данном случае нет необходимости использовать `break`, чтобы выйти из обработчика, пропуская остающиеся обработчики исключений. Каждый блок `catch` определяет отличную от других область действия, в то время как все случаи в операторе `switch` находятся внутри общей области действия этой структуры.

Исключение не имеет доступа к объектам, описанным внутри блока `try`, потому что они уже удалены в результате разматывания стека к тому моменту, когда обработчик начинает выполняться (см. раздел 13.13).

Что происходит, когда исключение возникает в обработчике исключения? Первоначальное исключение, которое было перехвачено, формально считается обработанным в тот момент, когда начинает выполняться обработчик исключения. Так что исключения, возникающие в обработчике, должны обрабатываться вне того блока `try`, в котором было сгенерировано первоначальное исключение.

Обработчики исключений могут быть написаны различными способами. Они могут рассмотреть ошибку и решить вызвать функцию `terminate`. Они могут просто повторно сгенерировать исключение (см. раздел 13.9). Они могут преобразовать один тип исключения в другой, генерируя это другое исключение. Они могут выполнить любые необходимые восстановления и продолжить выполнение с первого оператора после последнего обработчика исключения. Они могут рассмотреть ситуацию, вызвавшую ошибку, удалить причину ошибки и повторить вызов первоначальной функции, которая вызвала исключение (это не должно создавать бесконечную рекурсию). Они могут просто возвращать некоторое значение состояния в среду и т.д.

### Замечание по технике программирования 13.9

Самое лучшее – включить вашу стратегию обработки исключений в проектируемую систему до начала процесса проектирования. Трудно добавлять эффективную обработку исключений после того, как система реализована.

Если блок `try` не генерирует никаких исключений, то после завершения нормального выполнения блока `try` управление передается первому оператору после последнего обработчика `catch`, следующего за этим блоком `try`.

Невозможно возвратиться к точке генерации исключения, используя оператор `return` в обработчике `catch`. Такой оператор `return` просто вернет управление в ту функцию, которая вызывала функцию, содержащую данный блок `catch`.

### **Замечание по технике программирования 13.10**

Еще одна причина, по которой нецелесообразно использовать исключения для обычного потока управления, заключается в том, что эти «дополнительные» исключения могут попадаться на пути подлинных исключений, связанных с ошибками. Поэтому программисту становится труднее следить за большим числом исключений. Например, когда программа обрабатывает чрезмерное разнообразие исключений, можно ли быть действительно уверенным в том, какое из них перехватывается обработчиком **catch (...)**? Исключительные ситуации должны быть редкими, а не встречаться постоянно.

Важна последовательность обработчиков **catch**. Обработчики, которые перехватывают объекты производных классов, должны быть помещены перед обработчиками, которые перехватывают объекты базового класса; иначе, обработчик базового класса перехватит как объекты самого базового класса, так и объекты всех производных классов.

Когда исключение перехвачено, возможно, что ресурсы, которые были выделены, еще не освобождены в блоке **try**. Обработчик **catch**, если возможно, должен высвободить эти ресурсы. Например, обработчик **catch** должен освободить область памяти, выделенную операцией **new**, должен закрыть любые файлы, открытые в блоке **try**, сгенерировавшем исключение. Автоматические объекты разрушаются в результате разматывания стека прежде, чем обработчик начнет выполнятся.

Блок **catch** может обрабатывать ошибку способом, который дает возможность программе продолжать выполняться правильно. Или блок **catch** может завершить программу.

Обработчик **catch** непосредственно сам может обнаружить ошибку и сгенерировать исключение. Подобное исключение не будет обработано обработчиками **catch**, связанными с тем же самым блоком **try**, что и обработчик, сгенерировавший исключение. Это исключение будет перехвачено, если возможно, обработчиком, связанным со следующим внешним блоком **try**.

### **Типичная ошибка программирования 13.9**

Предположение, что исключение, сгенерированное обработчиком **catch**, будет обработано этим или любым другим обработчиком, связанным с тем же блоком **try**, который генерировал первоначальное исключение.

## **13.9. Повторная генерация исключений**

Возможно, что обработчик, который перехватил исключение, решит, что он не может обработать это исключение, или может просто потребоваться освободить ресурсы прежде, чем проводить дальнейшую обработку. В этом случае обработчик может просто повторно генерировать это исключение оператором

```
throw;
```

Такой оператор **throw** без аргументов повторно генерирует то же самое исключение. Если никакое исключение не было генерировано, то оператор повторной генерации вызывает обращение к функции завершения **terminate**. Поэтому оператор **throw** должен появляться только в обработчике **catch**; в противном случае, он вызовет обращение к **terminate**.

### Типичная ошибка программирования 13.10

Размещение пустого оператора **throw** вне обработчика **catch**; выполнение такого оператора **throw** вызовет обращение к **terminate**.

Даже если обработчик может обработать исключение и независимо от того, делает ли он какую-либо обработку этого исключения, он может повторно возбудить исключение для последующей обработки его вне этого обработчика.

Повторно сгенерированное исключение обнаруживается следующим внешним блоком **try** и перехватывается обработчиком из списка, следующего за этим внешним блоком **try**.

### Замечание по технике программирования 13.11

Используйте **catch (...)**, чтобы выполнить восстановления, которые не зависят от типа исключения, например, освобождение общих ресурсов. Исключение может быть повторно сгенерировано для применения к нему более специфических обработок во внешних блоках **catch**.

## 13.10. Создание условного выражения

Можно генерировать условное выражение. Но будьте внимательны, потому что правила приведения типов могут привести к тому, что значение, возвращаемое условным выражением, будет иметь тип, отличный от ожидаемого. Например, при генерации или **int**, или **double** из одного и того же условного выражения это условное выражение преобразует **int** в **double**. Следовательно результат будет всегда перехватываться обработчиком **catch** с параметром типа **double**, а не в зависимости от результата иногда перехватываться обработчиком **double** (если результат действительно **double**), а иногда — обработчиком **int**.

## 13.11. Спецификация исключений

В описании функции может быть указана *спецификация исключений* — список исключений, которые могут генерироваться этой функцией:

```
int g (float h) throw (a, b, c)
{
 // тело функции
}
```

Можно ограничить типы исключений, которые могут генерироваться данной функцией. Типы исключений специфицируются в объявлении функции как *спецификация исключений*. Эта спецификация перечисляет исключения, которые могут быть сгенерированы в функции. Функция может генерировать обозначенные исключения или типы, производные от них. Хотя при этом предполагается гарантия, что другие типы исключений генерироваться не будут, генерация их все-таки возможна. Если генерируется исключение, не перечисленное в спецификации, вызывается функция обработки непредусмотренных исключений **unexpected**.

Появление `throw ()` (то есть *пустой спецификации исключений*) после списка параметров функции объявляет, что функция не будет вырабатывать никаких исключений. На самом же деле такая функция могла бы **возбудить исключение**; это привело бы к вызову функции `unexpected`.

### Типичная ошибка программирования 13.11

Генерация исключения, не перечисленного в спецификации исключений функции, вызывает обращение к `unexpected`.

Функция без спецификации исключений может генерировать любое исключение:

```
void g (); // эта функция может генерировать любое исключение
```

Функции, содержащие пустую спецификацию исключений, не генерируют исключений:

```
void g () throw (); // эта функция не генерирует никаких исключений
```

Функция `unexpected` может быть переопределена с помощью функции `set_unexpected`.

Интересным аспектом обработки исключений является то, что компилятор не будет рассматривать как ошибку компиляции случай, когда функция содержит выражение `throw` для исключения, не перечисленного в спецификации исключений данной функции. Функция должна попытаться сгенерировать такое исключение во время выполнения прежде, чем эта ошибка будет перехвачена.

Если функция генерирует исключение некоторого класса, то эта функция может также генерировать исключения всех классов, которые являются производными от этого класса.

Никакая спецификация исключения не означает, что функция может генерировать любое исключение.

## 13.12. Обработка непредусмотренных исключений

Функция `unexpected` вызывает функцию, определенную с помощью функции `set_unexpected`. Если нет функции, определенной таким способом, то по умолчанию вызывается функция завершения программы `terminate`.

Функция `terminate` может быть вызвана различными путями:

- явно;
- если сгенерированное исключение не может быть перехвачено;
- если во время обработки исключения разрушен стек;
- как заданное по умолчанию действие при вызове `unexpected`;
- во время разматывания стека, вызванного исключением, и попытки со стороны деструктора сгенерировать исключение, что вызывает обращение к `terminate`.

Функция `set_terminate` может задавать функцию, которая будет вызываться при вызове `terminate`. Иначе, `terminate` вызывает `abort`.

Прототипы функций `set_terminate` и `set_unexpected` находятся в заголовочных файлах `<terminate.h>` и `<unexpected.h>` соответственно.

Функции `set_terminate` и `set_unexpected` возвращают указатели на предыдущие версии функции `terminate` и `unexpected` соответственно. Это дает возможность программисту сохранить указатели на эти функции, так что впоследствии они могут быть восстановлены.

Функции `set_terminate` и `set_unexpected` получают в качестве аргументов указатели на функции. Каждый аргумент должен указывать на функцию, возвращающую тип `void` и без аргументов.

Если последним действием определенной пользователем функции завершения не является завершение работы программы, то автоматически будет вызываться функция `abort` для прекращения работы программы после выполнения других операторов определенной пользователем функции завершения.

### 13.13. Конструкторы, деструкторы и обработка исключений

Сначала давайте разберемся с проблемой, которую мы упомянули, но которая еще не было удовлетворительно решена. Что случится, когда ошибка обнаружена в конструкторе? Например, как должен реагировать конструктор строки `String`, когда операция `new` возвращает нуль, указывающий, что в памяти нет достаточного места для хранения внутреннего представления строки `String`? Проблема состоит в том, что поскольку конструктор не может возвратить значение, то как мы информируем внешний мир, что объект не был нормально создан? Один вариант — просто возвратить неправильно созданный объект и надеяться, что фрагменты программы, использующие данный объект, проверят его и определят, что объект неверный. Другой вариант состоит в том, чтобы установить некоторую переменную вне конструктора. Генерируемое исключение передает внешнему миру информацию о безуспешной работе конструктора и дает возможность обработать этот отказ.

Чтобы перехватить исключение, обработчик должен иметь доступ к конструктору копии сгенерированного объекта (копия по умолчанию тоже подойдет).

Исключения, генерируемые в конструкторах, обусловливают вызов деструкторов всех объектов, являющихся частями того объекта, который создавался перед генерацией исключения.

Деструкторы вызываются для каждого автоматического объекта, созданного в блоке `try` до генерации исключения (это называется разматыванием стека). Исключение обрабатывается в момент, когда начинает выполняться обработчик; разматывание стека гарантированно завершается к этому времени. Если деструктор, вызываемый в результате разматывания стека, генерирует исключение, то вызывается функция завершения `terminate`.

Если объект имеет объекты-элементы и если исключение сгенерировано прежде, чем вмещающий объект полностью создан, тогда деструкторы будут выполняться для объектов-элементов, которые были созданы до генерации исключения.

Если к моменту генерации исключения был частично создан массив объектов, то будут выполняться деструкторы только созданных элементов массива.

При возникновении исключения может осложниться управление ресурсами. Исключение может препятствовать работе операторов, освобождающих ресурсы. Одним из возможных способов решения этой проблемы является инициализация локального объекта при выделении ресурса. Когда возникает исключение, будет вызываться его деструктор, который может освободить ресурс.

Можно перехватить исключения, генерируемые деструкторами. Просто включите функцию вызова деструктора в блок `try` и предусмотрите обработчик `catch` соответствующего типа.

Деструктор сгенерированного объекта выполняется после того, как завершается выполнение обработчика исключения.

## 13.14. Исключения и наследование

Различные классы исключений могут быть порождены из общего базового класса. Если написан `catch` для перехвата объектов исключений типа базового класса, он может также перехватывать все объекты классов, порожденных из этого базового класса. Это позволяет осуществлять полиморфную обработку родственных ошибок.

Использование наследования для исключений дает возможность обработчику исключений перехватывать родственные ошибки, используя очень компактную запись. Конечно, можно было бы перехватывать каждый тип объекта исключения порожденного класса индивидуально, но более удобно перехватить объект исключения базового класса. Кроме того, перехватывание объектов исключений порожденных классов индивидуально может быть источником ошибок, если программист забывает явно проверить какие-то из типов порожденных классов.

### Резюме

- Некоторые распространенные примеры исключений: нехватка памяти, выход индекса за пределы массива, арифметическое переполнение, деление на нуль, недопустимые параметры функций.
- Смысл обработки исключений заключается в том, чтобы дать возможность программам перехватывать и обрабатывать ошибки прежде, чем они произойдут и наступят их неприятные последствия. Если программист не обеспечивает средства обработки неисправимой ошибки, то при ее возникновении программа прекращает свою работу; исправимые ошибки обычно позволяют программе продолжать выполнение, но приводят к неправильным результатам.
- Обработка исключений создана для работы с синхронными ошибками, т.е. ошибками, которые появляются как результат выполнения программы.
- Обработка исключений не предназначена для работы с асинхронными ситуациями типа завершения операции ввода-вывода с диска, поступления сообщений от локальной сети, щелчка мыши, и т.п.; эти ситуации лучше обрабатываются другими средствами, такими, как обработка прерываний.

- Обработка исключений обычно используется в случаях, при которых ошибка обрабатывается в другой части программы (то есть в другой области действия), чем та, в которой эта ошибка обнаружена.
- Исключения не должны использоваться как альтернативный механизм задания потока управления. Поток управления со стандартными управляющими структурами вообще более ясен и более эффективен, чем применение исключений.
- Обработка исключений должна использоваться для тех компонентов программы, которые сами не предназначены непосредственно для обработки этих исключений.
- Обработка исключений должна использоваться, чтобы обрабатывать исключения, возникающие в таких компонентах программы, как функции, библиотеки и классы, которые широко используются и в которые не имеет смысла вводить собственную обработку исключений.
- Обработка исключений должна использоваться в больших проектах, чтобы обрабатывать одинаковым способом ошибки, возникающие в различных местах проекта
- Обработка исключений в C++ подключается в ситуации, в которой функция обнаруживает ошибку, но не способна сама с ней управиться. Такая функция генерирует исключение (или, как иногда говорят, возбуждает или выбрасывает исключение). Если исключение соответствует типу параметра в одном из блоков `catch`, выполняется код этого блока. В противном случае, вызывается функция завершения `terminate`, которая по умолчанию вызывает функцию `abort` для аварийного завершения работы.
- Программист включает в блок `try` код, который может генерировать ошибку, создающую исключение. Следом за блоком `try` записывается блоки перехвата `catch` (один или более). Каждый блок `catch` определяет тип исключений, которые он может перехватывать и обрабатывать. Каждый блок `catch` содержит программу — обработчик исключения.
- Программное управление при генерации исключения осуществляет выход из блока `try` и последовательный поиск в блоках `catch` соответствующего обработчика. Если никакие исключения в блоке `try` не генерируются, обработчики исключений для такого блока пропускаются и программа продолжает выполняться с операторов, следующих за последним блоком `catch`.
- Исключения генерируются в блоке `try` или в функции, вызываемой непосредственно или косвенно из блока `try`.
- После того, как исключение сгенерировано, управление не может возвратиться в точку его генерации.
- Из точки генерации исключения можно передать информацию обработчику этого исключения. Это информация о типе самого сгенерированного объекта или информация, помещенная в этот объект.
- Один из наиболее популярных создаваемых типов исключений — тип `char *`. Его просто включать в сообщение об ошибке как operand `throw`.

- Операнд `throw` может быть любого типа. Если операнд является объектом, мы называем его объектом исключения.
- Исключения, которые может генерировать некоторая функция, могут быть определены в спецификации исключений. Пустая спецификация исключений объявляет, что функция не будет генерировать никакие исключения.
- Исключение перехватывается ближайшим обработчиком исключений (ближайшим к блоку `try`, в котором было сгенерировано исключение), содержащим спецификацию соответствующего типа.
- В процессе генерации исключения создается и инициализируется временная копия операнда `throw`. Этот временный объект затем инициализирует параметр в обработчике исключения. Временный объект уничтожается, когда завершается выполнение обработчика исключения и управление передается программе.
- Ошибки не всегда проверяются явно. Например блок `try` может не содержать никаких проверок ошибок и не включать никаких операторов `throw`. Но код, вызываемый из блока `try`, может, конечно, содержать контроль ошибок, в частности, в конструкторах.
- Исключение завершает выполнение блока, в котором оно возникло.
- Обработчики исключений содержатся в блоках `catch`. Каждый блок `catch` начинается с ключевого слова `catch`, за которым следуют круглые скобки, содержащие тип и необязательное имя параметра. Затем в фигурных скобках записываются операторы обработки исключения. Когда исключение перехвачено, начинает выполняться программа в блоке `catch`.
- Обработчик `catch` определяет свою собственную область действия.
- Параметр в обработчике `catch` может быть поименован или нет. Если параметр поименован, на него можно ссылаться в обработчике. Если параметр не назван, то есть, если указан только тип соответствующего объекта исключения или записано многоточие, обозначающее перехват всех типов, то обработчик будет игнорировать сгенерированный объект. Обработчик может сгенерировать повторное исключение для использования его во внешнем блоке `try`.
- Можно изменить реакцию на неперехваченное исключение, заменив функцию завершения `terminate` другой и задав ее имя как аргумент в вызове функции `set_terminate`.
- `catch (...)` означает перехват всех исключений.
- Возможно, что никакой драйвер не будет соответствовать некоторому сгенерированному объекту. Это вызывает продолжение поиска соответствия в следующем внешнем блоке `try`, включающем данный.
- Обработчики исключений поочередно просматриваются в поисках соответствующего типа. Выполняется первый обработчик соответствующего типа. Когда этот обработчик завершает свою работу, управление передается на первый оператор после последнего блока `catch`.
- Последовательность записи обработчиков влияет на способ, которым исключение будет обрабатываться.

- Объект производного класса может быть перехвачен или обработчиком, в котором специфицирован этот тип производного класса, или обработчиками, в которых специфицированы типы любых базовых классов этого производного класса.
- Иногда программа может обрабатывать многие близко связанные типы исключений. Вместо того, чтобы обеспечивать каждое исключение отдельным классом и обработчиком `catch`, программист может создать один класс исключения и один обработчик `catch` для группы исключений. При возникновении каждого из таких исключений может создаваться один объект исключения с различными закрытыми данными. Обработчик `catch` может просматривать эти закрытые данные, чтобы различить типы исключений.
- Возможно, что, хотя имеется обработчик с точным соответствием типа, будет использовано соответствие, требующее стандартных преобразований, потому что этот обработчик встретится ранее того, который обеспечивает точное соответствие.
- По умолчанию, если для исключения не найден никакой обработчик, программа завершается.
- Обработчик исключения не может непосредственно обращаться к переменным в области действия блока `try`. Информация, необходимая обработчику, обычно передается в сгенерированный объект.
- Обработчики исключений могут рассмотреть ошибку и решить вызвать функцию `terminate`. Они могут просто повторно сгенерировать исключение. Они могут преобразовать один тип исключения в другой, генерируя это другое исключение. Они могут выполнить любые необходимые восстановления и продолжить выполнение с первого оператора после последнего обработчика исключения. Они могут рассмотреть ситуацию, вызвавшую ошибку, удалить причину ошибки и повторить вызов первоначальной функции, которая вызвала исключение (это не должно создавать бесконечную рекурсию). Они могут просто возвращать некоторое значение состояния в среду выполнения и т.д.
- Обработчик, который перехватывает объект производного класса, должен размещаться перед обработчиком, который перехватывает объект базового класса. Если обработчик базового класса был первым, он перехватит как объекты базового класса, так и объекты всех производных классов.
- Когда исключение перехвачено, возможно, что ресурсы, которые были выделены, еще не освобождены в блоке `try`. Обработчик `catch` должен освободить эти ресурсы.
- Возможно, что обработчик, который перехватил исключение, решит, что он не может сам обработать это исключение. В этом случае обработчик может просто повторно сгенерировать это исключение. Повторное исключение генерируется оператором `throw` без аргументов. Если никакое исключение не было сгенерировано, то оператор повторной генерации вызывает обращение к функции завершения `terminate`.

- Даже если обработчик может обработать исключение и независимо от того, делает ли он какую-либо обработку этого исключения, он может повторно возбудить исключение для последующей обработки его вне этого обработчика. Повторно сгенерированное исключение обнаруживается следующим внешним блоком `try` и перехватывается обработчиком из списка, следующего за этим внешним блоком `try`.
- Функция без спецификации исключений может генерировать любое исключение.
- Функция обработки непредусмотренных исключений `unexpected` вызывает функцию, указанную с помощью функции `set_unexpected`. Если нет функции, определенной таким способом, то по умолчанию вызывается функция завершения программы `terminate`.
- Функция `terminate` может быть вызвана различными путями: явно; если сгенерированное исключение не может быть перехвачено; если во время обработки исключения разрушен стек; как заданное по умолчанию действие при вызове `unexpected`; во время разматывания стека, вызванного исключением, и попытки со стороны деструктора сгенерировать исключение, что вызывает обращение к `terminate`.
- Прототипы функций `set_terminate` и `set_unexpected` находятся в заголовочных файлах `<terminate.h>` и `<unexpected.h>` соответственно.
- Функции `set_terminate` и `set_unexpected` возвращают указатели на предыдущие версии функций `terminate` и `unexpected` соответственно. Это дает возможность программисту сохранить указатели на эти функции, так что впоследствии они могут быть восстановлены.
- Функции `set_terminate` и `set_unexpected` получают в качестве аргументов указатели на функции. Каждый аргумент должен указать на функцию, возвращающую тип `void` и без аргументов.
- Если последним действием определенной пользователем функции завершения не является завершение работы программы, то автоматически будет вызываться функция `abort` для прекращения работы программы после выполнения других операторов определенной пользователем функции завершения.
- Исключение, сгенерированное вне блока `try`, вызывает завершение программы.
- Если после блока `try` обработчик не может быть найден, продолжается разматывание стека, пока соответствующий обработчик не будет найден. Если обработчик так и не найдется, вызывается функция завершения `terminate`, которая по умолчанию прерывает программу с помощью функции `abort`.
- Спецификации исключений перечисляют исключения, которые могут быть сгенерированы в функции. Функция может генерировать обозначенные исключения или типы, производные от них. Если генерируется исключение, не предусмотренное в спецификации, вызывается функция `unexpected`.
- Если функция генерирует исключение некоторого класса, то эта функция может также генерировать исключения всех классов, которые являются производными от этого класса.

- Чтобы перехватить исключение, обработчик должен иметь доступ к конструктору копии генерированного объекта.
- Исключения, генерируемые в конструкторах, обусловливают вызов деструкторов всех объектов-элементов того объекта, который создавался перед генерацией исключения.
- Если к моменту генерации исключения был частично создан массив объектов, то будут выполняться деструкторы только созданных элементов массива.
- Исключения, генерируемые деструкторами, можно перехватить, включив функцию вызова деструктора в блок `try` и предусмотрев обработчик `catch` соответствующего типа.
- Преимущества использования наследования для исключений заключается в том, что это дает возможность обработчику исключений перехватывать родственные ошибки, используя очень компактную запись. Конечно, можно было бы перехватывать каждый тип объекта исключения порожденного класса индивидуально, но более удобно перехватить объект исключения базового класса.

## Терминология

|                                         |                                        |
|-----------------------------------------|----------------------------------------|
| <code>abort ()</code>                   | обработка исключения                   |
| <code>catch</code>                      | обработчик для базового класса         |
| <code>catch (...)</code>                | обработчик для производного класса     |
| <code>exit ()</code>                    | обрабтчик исключения                   |
| <code>new_handler</code>                | обрабтчик исключения                   |
| <code>set_new_handler ()</code>         | по умолчанию                           |
| <code>set_terminate ()</code>           | обратный вызов                         |
| <code>set_unexpected ()</code>          | объект исключения                      |
| <code>terminate ()</code>               | объявление исключения                  |
| <code>throw</code> без аргументов       | оператор <code>try</code>              |
| <code>unexpected ()</code>              | ответственные приложения               |
| аргумент <code>catch</code>             | отказоустойчивость                     |
| асинхронная ошибка                      | перехват всех исключений               |
| блок <code>catch</code>                 | перехват группы исключений             |
| блок <code>try</code>                   | перехват исключения                    |
| включающий в себя блок <code>try</code> | повторная генерация исключения         |
| вложенные обработчики исключений        | предложение <code>catch</code>         |
| возбуждение исключения                  | пустая спецификация <code>throw</code> |
| выражение <code>throw</code>            | пустая спецификация исключений         |
| генерация исключения                    | разматывание стека                     |
| генерация непредусмотренного исключения | генерированное исключение              |
| генерация объекта                       | генерированный аргумент                |
| живучесть программы                     | синхронная ошибка                      |
| исключение                              | спецификация исключений                |
| исключения, не связанные с ошибками     | список <code>throw</code>              |
| макрос <code>assert</code>              | список исключений                      |
| многоточие (...) как тип перехвата      | список обработчиков                    |
| неперехваченное исключение              | тип генерированного объекта            |
| нехватка свободной памяти               | точка генерации исключений             |
| обработка исключений                    | условное исключение                    |
|                                         | функция без спецификации исключений    |

## Типичные ошибки программирования

- 13.1. Причина, по которой исключения могут быть опасными как альтернатива нормальному потоку управления, заключается в том, что стек может не разматываться и ресурсы, распределенные до возникновения исключения, могут не освобождаться. Этой проблемы можно избежать тщательным программированием.
- 13.2. Прерывание выполнения программы может оставить ресурс в таком состоянии, в котором другие программы не могут его использовать, и мы получаем так называемую «утечку ресурса».
- 13.3. Исключение должно генерироваться только внутри блока `try`. Исключение сгенерированное вне блока `try`, вызывает обращение к `terminate` — прерыванию программы.
- 13.4. Предположение, что после обработки исключения управление вернется к первому оператору после того, который сгенерировал это исключение.
- 13.5. Задание разделяемого запятыми списка аргументов `catch`.
- 13.6. Размещение `catch (...)` перед другими блоками `catch` препятствует выполнению всех других обработчиков; `catch (...)` всегда должен размещаться последним в списке обработчиков после блока `try`, иначе будет зафиксирована синтаксическая ошибка.
- 13.7. Размещение `catch`, который перехватывает объект базового класса, перед `catch`, который перехватывает объект класса, производного от данного базового, является синтаксической ошибкой. Перехватчик `catch` базового класса перехватит все объекты производных классов, так что `catch` производного класса никогда не будет выполняться.
- 13.8. Размещение обработчика исключения с типом аргумента `void *` перед обработчиками исключений с другими типами указателей вызывает синтаксическую ошибку. Обработчик `void *` будет перехватывать все исключения типа указатель, так что другие обработчики никогда не будут выполняться.
- 13.9. Предположение, что исключение, сгенерированное обработчиком `catch`, будет обработано этим или любым другим обработчиком, связанным с тем же блоком `try`, который сгенерировал первоначальное исключение.
- 13.10. Размещение пустого оператора `throw` вне обработчика `catch`; выполнение такого оператора `throw` вызовет обращение к `terminate`.
- 13.11. Генерация исключения, не перечисленного в спецификации исключений функции, вызывает обращение к `unexpected`.

## Хороший стиль программирования

- 13.1. Используйте исключения для ошибок, которые должны быть обработаны в другой области действия, отличной от той, где они происходят. Используйте другие средства для обработки ошибок в той области действия, в которой они происходят.

- 13.2. Избегайте использовать обработку исключений для целей, отличных от обработки ошибок, потому что это может уменьшить ясность программы.
- 13.3. Используйте обычные методы обработки ошибок (а не обработку исключений) для построения непосредственного, локального обработчика ошибок, в котором программе легче управляться со своими собственными ошибками.
- 13.4. Избегайте имени `Exception` для любого класса исключения. Весьма вероятно, что это имя используется библиотеками, и возможно даже будет включено в развивающийся стандарт ANSI для C++.
- 13.5. Привязка каждого типа ошибки времени выполнения к соответственно названному объекту исключения улучшает ясность программы.

### Советы по повышению эффективности

- 13.1. Хотя и можно использовать обработку исключений для целей, отличных от обработки ошибок, но это может уменьшать эффективность программы.
- 13.2. Обработка исключений в общем случае выполнена в трансляторах таким образом, что, когда исключение не возникает, присутствие кода обработки исключений не вносит никаких дополнительных издержек (или они очень невелики). Когда же исключения возникают, они влекут за собой накладные расходы во время выполнения.

### Замечания по мобильности

- 13.1. Как уже было сказано, стандарт ANSI для C++ еще не завершен и возможно, что особенности работы с исключениями, описанные в этой главе, могут еще развиваться далее в течение следующих нескольких лет. Когда мы завершили написание этой книги, главные продавцы трансляторов C++ начали поставлять версии с обработкой исключений. Опыт, полученный при работе с ними в течение нескольких следующих лет должен помочь в продвижении к стандарту.

### Замечания по технике программирования

- 13.1. Поток управления со стандартными управляющими структурами вообще более ясен и более эффективен, чем применение исключений.
- 13.2. Обработка исключений особенно полезна для систем с отдельно разработанными компонентами. Это типично для больших программных комплексов. Обработка исключений позволяет проще объединять компоненты. Каждый компонент может сам осуществлять обнаружение исключительной ситуации отдельно от ее обработки.
- 13.3. Когда идет работа с библиотеками, источник вызова библиотечной функции будет, вероятно, использовать свой уникальный обработ-

чик исключения, сгенерированного библиотечной функции. Маловероятно, чтобы библиотечная функция выполняла такую обработку ошибки, которая удовлетворила бы конкретные потребности всех пользователей. Следовательно, исключения — удобный способ работы с ошибками, возникающими в библиотечных функциях.

- 13.4. Ключевым моментом в стиле обработки исключений в C++ является то, что часть программы или системы, которая будет обрабатывать исключение, может быть совершенно отделена и удалена от части программы, которая обнаружила и возбудила исключение.
- 13.5. Если необходимо передать информацию относительно ошибки, которая вызвала исключение, то такая информация может быть помещена в сгенерированный объект. Обработчик `catch` должен содержать в этом случае имя параметра, через который эта информация может быть принята.
- 13.6. Может быть сгенерирован объект, не содержащий информацию для передачи; в этом случае обработчику достаточно простой информации о том, что сгенерировано исключение данного типа, чтобы правильно выполнить свою задачу.
- 13.7. Недостаток перехвата исключений с помощью `catch (...)` заключается в том, что вы обычно не можете знать, каков тип исключения. Другой недостаток заключается в том, что без поименованного параметра не существует никакого способа обратиться в обработчике исключения к объекту исключения.
- 13.8. Программист определяет последовательность, в которой перечисляются обработчики исключений. Эта последовательность может влиять на то, как обрабатываются исключения, возникающие в этом блоке `try`.
- 13.9. Самое лучшее — включить вашу стратегию обработки исключений в проектируемую систему до начала процесса проектирования. Трудно добавлять эффективную обработку исключений после того, как система реализована.
- 13.10. Еще одна причина, по которой нецелесообразно использовать исключения для обычного потока управления, заключается в том, что эти «дополнительные» исключения могут попадаться на пути подлинных исключений, связанных с ошибками. Поэтому программисту становится труднее следить за большим числом исключений. Например, когда программа обрабатывает чрезмерное разнообразие исключений, можно ли быть действительно уверенными в том, какое из них перехватывается обработчиком `catch (...)`? Исключительные ситуации должны быть редкими, а не встречаться постоянно.
- 13.11. Используйте `catch (...)`, чтобы выполнить восстановления, которые не зависят от типа исключения, например, освобождение общих ресурсов. Исключение может быть повторно сгенерировано для применения к нему более специфических обработок во внешних блоках `catch`.

## Упражнения для самопроверки

- 13.1. Перечислите пять обычных примеров исключений.
- 13.2. Приведите несколько причин, по которым методы обработки исключений не должны использоваться для обычного программного управления.
- 13.3. Почему целесообразно использовать исключения для обработки ошибок, вызванных библиотечными функциями?
- 13.4. Что такое — «утечка ресурса»?
- 13.5. Если в блоке `try` не генерируются никакие исключения, куда передается управление после того, как блок `try` завершил работу?
- 13.6. Что произойдет, если исключение будет сгенерировано вне блока `try`?
- 13.7. Укажите основное достоинство и основной недостаток использования `catch (...)`.
- 13.8. Что произойдет, если ни один из обработчиков не соответствует типу сгенерированного объекта?
- 13.9. Что случится, если несколько обработчиков соответствуют типу сгенерированного объекта?
- 13.10. Почему программисту может быть желательно определить базовый тип класса как тип обработчика `catch` и затем генерировать объекты типов производных классов?
- 13.11. Как можно написать обработчик `catch`, чтобы обработать родственные типы ошибок без использования наследования классов исключений?
- 13.12. Какой тип указателя надо использовать в обработчике `catch`, чтобы перехватывать любое исключение типа указатель?
- 13.13. Предположите, что доступен обработчик `catch` с точным соответствием типу объекта исключения. При каких обстоятельствах может выполняться не этот, а другой обработчик для объектов исключения этого типа?
- 13.14. Должна ли генерация исключения вызывать завершение программы?
- 13.15. Что происходит, когда обработчик `catch` генерирует исключение?
- 13.16. Что делает оператор `throw;?`
- 13.17. Как программист ограничивает типы исключений, которые могут генерироваться в функции?
- 13.18. Что происходит, если функция генерирует исключение типа, не допускаемого спецификацией исключений этой функции?
- 13.19. Что происходит с автоматическими объектами, которые были созданы в блоке `try`, когда этот блок генерирует исключение?

## Ответы на упражнения для самопроверки

- 13.1. Нехватка памяти, выход индекса за пределы массива, арифметическое переполнение, деление на нуль, недопустимые параметры функций.
- 13.2. (а) Обработка исключений создана, чтобы обрабатывать нечасто встречающиеся ситуации, но которые часто приводят к завершению программы, так что от авторов компиляторов не требуется, чтобы выполнение обработки исключений проводилось оптимально.  
(б) Поток управления со стандартными управляющими структурами вообще более ясен и более эффективен, чем применение исключений. (с) Могут появляться проблемы, связанные с тем, что стек может не разматываться и ресурсы, распределенные до возникновения исключения, могут не освобождаться. (д) «Дополнительные» исключения могут попадаться на пути подлинных исключений, связанных с ошибками. Поэтому программисту становится труднее следить за большим числом исключений. Например, можно ли быть действительно уверенным в том, какое из исключений перехватывается обработчиком `catch (...)`?
- 13.3. Маловероятно, чтобы библиотечная функция выполняла такую обработку ошибок, которая удовлетворяла бы потребности всех пользователей.
- 13.4. Прерывание выполнения программы может оставить ресурс в состоянии, в котором другие программы не будут способны его использовать.
- 13.5. Обработчики исключений (в блоках `catch`) для этого блока `try` пропускаются и программа продолжает выполнение, начиная с оператора после последнего блока `catch`.
- 13.6. Исключение, сгенерированное вне блока `try`, вызывает обращение к функции `terminate`.
- 13.7. Обработчик вида `catch (...)` перехватывает ошибки любого типа, генерирующиеся в блоке `try`. Достоинство заключается в том, что никакая ошибка не может ускользнуть от обработки. Недостаток заключается в том, что такой обработчик `catch` не имеет параметра `i`, значит, не может ссылаться на информацию в сгенерированном объекте; следовательно, он не может знать причину ошибки.
- 13.8. Это вызовет продолжение поиска соответствия в следующем внешнем блоке `try`. При продолжении этого процесса может оказаться, что в программе не имеется ни одного обработчика, соответствующего типу сгенерированного объекта; в этом случае вызывается функция завершения `terminate`, которая по умолчанию вызывает в свою очередь функцию `abort`. Альтернативная функция `terminate` может быть задана как аргумент функции `set_terminate`.
- 13.9. Выполняется первый из соответствующих обработчиков исключения после блока `try`.
- 13.10. Это прекрасный способ перехватывать родственные типы исключений.

- 13.11. Сделайте единственный класс исключения и обработчик `catch` для группы исключений. При возникновении каждого исключения можно создавать объект исключения с различными закрытыми данными. Обработчик `catch` может просматривать эти закрытые данные и определять тип исключения.
- 13.12. `Void *`.
- 13.13. Если обработчик, использующий стандартные преобразования, будет записан перед обработчиком с точным соответствием.
- 13.14. Нет, но завершается выполнение блока, в котором сгенерировано исключение.
- 13.15. Исключение будет обработано обработчиком `catch` (если таковой существует), связанным с блоком `try` (если он есть), включающим тот обработчик `catch`, который вызвал исключение.
- 13.16. Он генерирует повторное исключение.
- 13.17. Записывает спецификацию исключений, представляющую собой список типов исключений, которые могут генерироваться в функции.
- 13.18. Вызывается функция `unexpected`.
- 13.19. В процессе разматывания стека вызываются деструкторы каждого из этих объектов.

## Упражнения

- 13.20. Составьте список различных исключительных ситуаций, которые встречаются в программах данной книги. Внесите в список сколько сможете дополнительных исключительных ситуаций. Для каждой из них кратко опишите, как программа могла бы обработать данное исключение, используя методы обработки исключений, рассмотренные в этой главе. Некоторые типичные исключения: деление на нуль, арифметическое переполнение, выход индекса массива за допустимые пределы, нехватка свободной памяти и т.д.
- 13.21. При каких обстоятельствах программист мог бы не указывать имя параметра при описании типа объекта, который будет перехватываться обработчиком?
- 13.22. Программа содержит оператор
- ```
throw;
```
- Где обычно встречается этот оператор? Что будет, если этот оператор появляется в другой части программы?
- 13.23. При каких обстоятельствах вы бы использовали следующий оператор?
- ```
catch (...) { throw; }
```
- 13.24. Сравните обработку исключений с другими способами обработки ошибок, рассмотренными в этой главе.

- 13.25. Составьте список преимуществ обработки исключений перед обычными средствами обработки ошибок.
- 13.26. Приведите несколько аргументов в обоснование того, что исключения не должны использоваться как альтернативная форма программного управления.
- 13.27. Опишите методику обработки родственных исключений.
- 13.28. До этой главы мы выяснили, что работа с ошибками, обнаруженными конструкторами, несколько затруднительна. Исключения дают нам намного более удобные средства работы с такими ошибками. Рассмотрите конструктор для класса `String`. Конструктор использует операцию `new`, чтобы выделить область памяти. Предположите сбой операции `new`. Покажите, как бы вы поступали в этом случае без обработки исключения. Рассмотрите основные проблемы. Покажите, как бы вы обрабатывали исключение, связанное с подобной нехваткой памяти. Объясните, почему обработка исключения предпочтительнее.
- 13.29. Предположите, что программа генерирует исключение и начинает выполняться соответствующий обработчик. Теперь предположите, что обработчик исключения сам генерирует такое же исключение. Создает ли это бесконечную рекурсию? Напишите программу на C++, чтобы проверить ваш анализ этой ситуации.
- 13.30. Используйте наследование, чтобы создать базовый класс исключений и различные производные классы исключений. Затем покажите, что обработчик `catch`, определяющий базовый класс, перехватывает исключения производных классов.
- 13.31 Генерация исключений с условным выражением имеет свои сложности. Покажите условное выражение, которое возвращает или тип `double`, или `int`. Создайте обработчик `catch` для перехвата целых `int` и обработчик перехвата типа `double`. Покажите, что выполняется только обработчик `catch` типа `double` независимо от того, возвращается ли `int` или `double`.
- 13.32. Напишите программу на C++, предназначенную для генерации и обработки ошибки, связанной с нехваткой памяти. Ваша программа должна в цикле давать запрос на динамическое выделение памяти с помощью операции `new`.
- 13.33. Напишите программу на C++, которая показывает, что все деструкторы объектов, созданных в блоке, вызываются прежде, чем в этом блоке генерируется исключение.
- 13.34. Напишите программу на C++, которая показывает, что при возникновении исключения вызываются деструкторы только тех объектов-элементов, которые были созданы прежде, чем произошла генерация исключения.
- 13.35. Напишите программу на C++, которая показывает, что любое исключение перехватывается с помощью `catch (...)`.
- 13.36. Напишите программу на C++, которая показывает, что важна последовательность обработчиков исключений. Выполняется первый

соответствующий обработчик. Скомпилируйте и выполните вашу программу с двумя различными последовательностями обработчиков, чтобы показать, что при этом наблюдаются различные результаты.

- 13.37. Напишите программу на C++, которая демонстрирует конструктор, передающий информацию о своем отказе обработчику исключения после блока `try`.
- 13.38. Напишите программу на C++, которая использует многоуровневую иерархию наследования классов исключения для создания ситуации, в которой важна последовательность обработчиков исключений.
- 13.39. При использовании `setjmp` и `longjmp` программа может сразу передать управление подпрограмме ошибки из глубоко вложенной функции. К сожалению, в этом случае при разматывания стека не вызываются деструкторы автоматических объектов, которые были созданы в течение последовательных вложенных вызовов функций. Напишите программу на C++, которая показывает, что эти деструкторы действительно не вызываются.
- 13.40. Напишите программу на C++, которая иллюстрирует повторную генерацию исключений.
- 13.41. Напишите программу на C++, которая использует функцию `set_unexpected` для установки определенной пользователем функции обработки непредусмотренных исключений `unexpected`, затем снова использует `set_unexpected`, а затем возвращает `unexpected` обратно к предыдущей функции. Напишите подобную же программу для проверки `set_terminate` и `terminate`.
- 13.42. Напишите программу на C++, которая показывает, что функция со своим собственным блоком `try` не должна перехватывать каждую возможную ошибку, сгенерированную внутри `try`. Некоторые исключения могут быть пропущены и обработаны в других областях действия.
- 13.43. Напишите программу на C++, которая генерирует ошибку в глубоко вложенном обращении к функции и все такие исключение перехватывается обработчиком `catch`, следующим за блоком `try`, включающим эту цепочку вызовов.

г л а в а

---

# 14

## Обработка файлов и ввод-вывод потоков строк



### Ц е л и

- Научиться создавать, читать, записывать и обновлять файлы.
- Овладеть обработкой файлов последовательного доступа.
- Овладеть обработкой файлов произвольного доступа.
- Научиться определять операции неформатированного ввода-вывода высокого уровня.
- Понять различие между обработкой файлов с форматированными и «сырыми» данными.
- Построить программу обработки запросов для файлов произвольного доступа.
- Научиться осуществлять ввод-вывод символьных строк.

## План

- 14.1. Введение**
- 14.2. Иерархия данных**
- 14.3. Файлы и потоки**
- 14.4. Файлы последовательного доступа**
- 14.5. Чтение данных из файла последовательного доступа**
- 14.6. Обновление файлов последовательного доступа**
- 14.7. Файлы произвольного доступа**
- 14.8. Создание файла произвольного доступа**
- 14.9. Произвольная запись данных в файл произвольного доступа**
- 14.10. Последовательное считывание данных из файла произвольного доступа**
- 14.11. Пример: программа по обработке запросов**
- 14.12. Обработка потока строк**
- 14.13. Ввод-вывод объектов**

*Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения*

### 14.1. Введение

Хранение данных в переменных и массивах является временным. Файлы предназначены для постоянного хранения больших объемов данных. Компьютеры сохраняют файлы на *вспомогательных запоминающих устройствах*, таких, как магнитные диски, оптические диски и магнитные ленты. В этой главе объясняется, каким образом создаются и обновляются файлы данных и как они обрабатываются программами на C++. Также проводится сравнение обработки файлов с форматированными данными и с необработанными («сырыми») данными. Детально изучаются методы ввода данных в символьные массивы и вывода данных из символьных массивов.

## 14.2. Иерархия данных

В конечном счете, все элементы данных, обрабатываемые компьютером, сводятся к комбинациям нулей и единиц. Так делается потому, что это является наиболее простым и экономичным при создании электронных устройств, которые могут принимать два устойчивых состояния: одно состояние представляется 0, а другое — 1. Просто удивительно, что сложнейшие функции, выполняемые компьютерами, являются, по большей части, операциями с нулями и единицами.

Наименьшему элементу данных в компьютере можно присвоить значение либо 0, либо 1.

Такой элемент данных называется *битом* («binary digit» — бинарный разряд, который может принимать одно из двух значений). Компьютер выполняет простые битовые операции, такие, как проверка значения бита, присвоение биту значения и инверсия бита (из 1 в 0 или из 0 в 1).

Программистам неудобно работать с данными, представленными на низком уровне — битами. Они предпочитают работать с данными, представленными *десятичными цифрами* (т.е. 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9), *буквами* (т.е. прописными буквами A — Z и строчными a — z) и специальными символами (т.е. \$, @, %, &, \*, (, ), -, +, :, ?, / и многими другими). Цифры, буквы и специальные символы называются *символами*. Множество всех символов, используемых для написания программ и представления элементов данных на конкретном компьютере, называется *набором символов* компьютера. Поскольку компьютеры могут обрабатывать только единицы и нули, то каждый символ в алфавите компьютера представляется в виде комбинации единиц и нулей, которая называется *байтом*. Байты обычно состоят из восьми битов. Программисты создают программы и элементы данных с помощью символов, а компьютеры манипулируют и обрабатывают символы как комбинацию битов.

Как символы состоят из битов, так и поля состоят из символов (или байтов). Поле — это группа символов, имеющая некоторый смысл. Например, поле, состоящее только из прописных и строчных букв, может использоваться для представления имени человека.

Элементы данных, обрабатываемые компьютерами, образуют *иерархию данных*, в которой элементы данных укрупняются и становятся все более сложными по структуре, поскольку процесс обработки переходит от битов к символам (байтам), полям и т.д.

Запись (т.е. *struct* или *class* в C++) образуется из нескольких полей, (называемых в C++ элементами или членами). В платежной ведомости, например, запись для отдельного служащего может состоять из следующих полей:

1. Идентификационный номер служащего
2. Имя
3. Адрес
4. Почасовой оклад
5. Число заявленных льгот
6. Годовой доход
7. Объем удерживаемых федеральных налогов и т.д.

Таким образом, запись — это группа связных полей. В предыдущем примере каждое из этих полей относится к одному и тому же служащему. Конечно, компания, у которой большое число служащих, должна иметь платежную ведомость на каждого из них. Файл — это группа связных записей. Файл платежных ведомостей компании обычно содержит по одной записи для каждого служащего. Следовательно, файл платежных ведомостей мелкой компании может иметь, например, 22 записи, в то время как для крупной компании он может содержать 100000 записей. Весьма распространенным явлением для компаний является наличие многих файлов, каждый из которых содержит миллионы символов информации. На рис. 14.1 поясняется *иерархия данных*.

Для облегчения поиска в файле заданных записей по крайней мере одно их полей в каждой записи выбирается в качестве *ключа записи*. Ключ идентифицирует, что запись относится к конкретному человеку или сущности, то есть что она является уникальной среди других записей в файле. В платежной записи, описанной выше, идентификационный номер служащего обычно выбирается в качестве ключа записи.

Существует множество способов организации записей в файле. Наиболее распространенный тип организации записей в файле называется *последовательным файлом*, в котором записи обычно хранятся в последовательности, соответствующей ключевому полю. В файле платежных ведомостей записи обычно размещаются в последовательности, соответствующей идентификационному номеру служащего. Первая запись служащего в файле содержит наименьший идентификационный номер служащего, а последующих записей хранятся в порядке возрастания идентификационных номеров служащих.

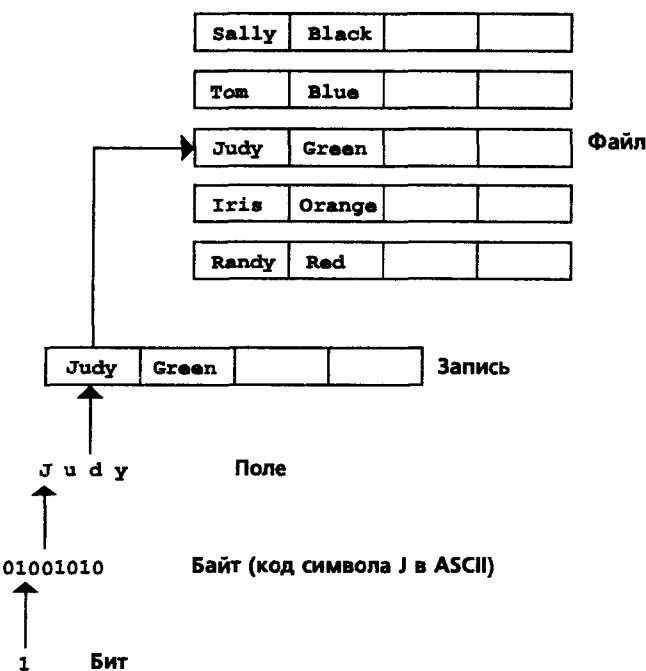


Рис. 14.1. Иерархия данных

Большинство фирм использует множество разнообразных файлов для хранения данных. Например, компании могут иметь файлы по заработной плате, файлы счетов по кредитной задолженности (со списками сумм по задолженности клиентов), итоговые файлы (со списком фактов по всем статьям) и многие другие типы файлов.

Группу связанных файлов, порой, называют *базой данных*. Совокупность программ, предназначенных для создания баз данных и управления ими, называется *системой управления базой данных* — СУБД.

### 14.3. Файлы и потоки

В C++ каждый файл рассматривается как последовательный поток байтов (рис. 14.2). Каждый файл завершается или *маркером конца файла* (EOF — end-of-file marker) или указанным числом байтов, записанным в служебную структуру данных поддерживающей системой. Когда файл *открывается*, то создается объект и с этим объектом связывается поток. В главе 11 показано, что автоматически создаются четыре объекта — `cin`, `cout`, `cerr` и `clog`. Потоки, связанные с этими объектами, обеспечивают каналы связи между программой и отдельными файлами или устройствами. Например, объект `cin` (объект стандартного потока ввода) дает возможность программе вводить данные с клавиатуры, объект `cout` (объект стандартного потока вывода) позволяет программе выводить данные на экран, объекты `cerr` и `clog` (объекты стандартного потока ошибок) позволяют программе выводить на экран сообщения об ошибках.

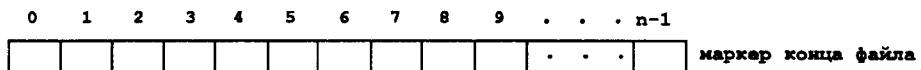


Рис. 14.2. Язык C++ рассматривает файл как набор из  $n$  байтов

Для обработки файлов в C++ должны быть включены заголовочные файлы `<iostream.h>` и `<fstream.h>`. Файл `<fstream.h>` включает определения классов потоков `ifstream` (для ввода из файла), `ofstream` (для вывода в файл) и `fstream` (для ввода-вывода файлов). Файлы открываются путем создания объектов этих классов потоков. Эти классы потоков являются производными (т.е. наследуют функциональные возможности) соответственно от классов `istream`, `ostream` и `iostream`. Таким образом, функции-элементы, операции и манипуляторы, описанные в главе 11, «Потоки ввода-вывода в C++», могут быть также применены и к потокам файлов. Иерархия классов ввода-вывода, рассмотренная до этого момента, показана на рис. 14.3.

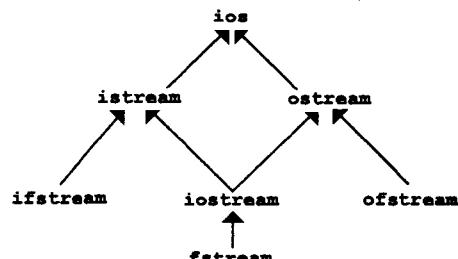


Рис. 14.3. Часть иерархии классов потоков ввода-вывода

## 14.1. Создание файла последовательного доступа

C++ не предписывает никакой структуры файлу. Таким образом, понятия, вроде «запись», не существуют в файлах языка C++. Следовательно, программист должен задавать структуру файлов в соответствии с требованиями прикладных программ. В следующем примере показано, каким образом программист может задавать простую структуру записей в файле. Сначала представим программу, а затем детально изучим ее.

Программа на рис. 14.4 создает простой файл последовательного доступа, который можно использовать в системе платежных счетов по кредиторской задолженности для помощи в управлении деньгами по этим счетам.

```
// fig14_4.cpp
// Создание последовательного файла
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main ()
{
 ofstream outClientFile("clients.dat", ios::out);

 if (! outClientFile) {
 cerr << " Файл не может быть открыт" << endl;
 exit(1); // прототип в stdlib.h
 }

 cout << "Введите счет, имя и баланс." << endl
 << "Введите EOF для окончания ввода." << endl << "? ";

 int account;
 char name[10];
 float balance;

 while (cin >> account >> name >> balance) {
 outClientFile << account << ' ' << name
 << ' ' << balance << endl;
 cout << "? ";
 }
}
```

```
Введите счет, имя и баланс.
Введите EOF для окончания ввода.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
?^Z
```

Рис. 14.4. Создание последовательного файла

Для каждого клиента программа получает номер счета, имя клиента, баланс клиента (сумма, которую клиент должен компании за товары и услуги, полученные в прошлом). Данные, полученные для каждого клиента, образуют запись. Номер счета в этом приложении используется в качестве ключа записи, то есть файл будет создаваться и обрабатываться в соответствии с порядком номеров счетов. Эта программа предполагает, что пользователь вводит записи в последовательности номеров счетов. В более полную систему платежных счетов может быть включена возможность сортировки, позволяющая пользователю вводить записи в произвольной последовательности — записи будут потом рассортированы и записаны в файл.

Теперь давайте рассмотрим эту программу. Как утверждалось выше, файлы открываются путем создания объектов классов потоков `ifstream`, `ofstream` или `fstream`. На рис. 14.4 файл должен быть открыт для вывода, так что создается объект `ofstream`. Конструктору объекта передаются два аргумента — *имя файла* и *режим открытия файла*. Для объекта `ofstream` режим открытия файла может быть или `ios::out` для вывода данных в файл, или `ios::app` — для добавления данных в конец файла (без модификации каких-либо данных, уже имеющихся в файле). Существующие файлы, открываемые режимом `ios::out`, усекаются — все данные в файле отбрасываются. Если какой-то файл еще не существует, тогда создается файл с тем же именем. Объявление

```
ofstream outClientFile("clients.dat", ios::out);
```

создает объект `outClientFile` класса `ofstream`, связанный с файлом `clients.dat`, который открывается для вывода. Аргументы `"clients.dat"` и `ios::out` передаются конструктору класса `ofstream`, который открывает файл. Это устанавливает «линию связи» с файлом. По умолчанию объекты класса `ofstream` открыты для вывода, поэтому для открытия файла `clients.dat` для вывода может быть использован оператор

```
ofstream outClientFile("clients.dat");
```

На рис. 14.5 приведен список режимов открытия файлов.

#### Типичная ошибка программирования 14.1

Открытие существующего файла для вывода (`ios::out`) в то время, как пользователь на самом деле желает сохранить этот файл; содержимое файла отбрасывается без предупреждения.

| Режим                       | Описание                                                                                                       |
|-----------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>ios::app</code>       | Записать все данных в конец файла.                                                                             |
| <code>ios::ate</code>       | Переместиться в конец исходного открытого файла. Данные могут быть записаны в любое место файла.               |
| <code>ios::in</code>        | Открыть файл для ввода.                                                                                        |
| <code>ios::out</code>       | Открыть файл для вывода.                                                                                       |
| <code>ios::trunc</code>     | Отбрасывать содержимое файла, если он существует (это также по умолчанию делается для <code>ios::out</code> ). |
| <code>ios::nocreate</code>  | Если файл не существует, то операция его открытия не выполняется.                                              |
| <code>ios::noreplace</code> | Если файл существует, то операция его открытия не выполняется.                                                 |

Рис. 14.5. Режимы открытия файла

### Типичная ошибка программирования 14.2

Использование неверного объекта класса **ofstream** при ссылке на файл.

Объект класса **ofstream** может быть создан без открытия какого-то файла — в этом случае файл может быть связан с объектом позднее. Например, объявление

```
ofstream outClientFile;
```

создает объект **outClientFile** класса **ofstream**. Функция-элемент **open** класса **ofstream** открывает файл и связывает его с существующим объектом класса **ofstream**, как показано ниже:

```
outClientFile.open("clients.dat", ios::out);
```

### Типичная ошибка программирования 14.3

Не открывается файл перед попыткой сослаться на него в программе.

После создания объекта класса **ofstream** и попытки открыть его программа проверяет, была ли операция открытия файла успешной. Фрагмент программы

```
if (!outClientFile) {
 cerr << "Файл не может быть открыт" << endl;
 exit(1);
}
```

использует перегруженную функцию-операцию **operator!** — элемент класса **ios**, для определения того, успешно ли открылся файл. Условие возвращает ненулевое значение (**true**), если при выполнении операции **open** для потока устанавливаются или **failbit** или **badbit**. Некоторые возможные ошибки являются следствием попытки открыть для чтения несуществующий файл, попытки открыть файл для чтения без разрешения или открытия файла для записи, когда на диске нет свободного места.

Когда условие указывает, что попытка открытия файла была безуспешной, выводится сообщение «Файл не может быть открыт» и вызывается функция **exit** для завершения программы. Аргумент функции **exit** возвращается среде окружения, из которой программа была вызвана. Аргумент 0 показывает, что программа завершается нормально, а любое другое значение, указывает среде окружения, что программа прекратила выполнение из-за ошибки. Значение, возвращаемое функцией **exit**, используется средой окружения (чаще всего операционной системой) для соответствующей реакции на ошибку.

Другая перегруженная функция-операция **operator void\*** — элемент класса **ios**, превращает поток в указатель, так что можно проверить, является ли он 0 (нулевым указателем), или имеет ненулевое значение (любое другое значение указателя). Если для потока устанавливаются **failbit** или **badbit** (см. главу 11), то возвращается 0 (**false**). Условие в заголовке оператора **while** автоматически вызывает функцию-элемент **operator void\***:

```
while (cin >> account >> name >> balance)
```

Это условие истинно, пока для `cin` не устанавливаются ни `failbit`, ни `badbit`. Ввод маркера конца файла устанавливает `failbit` для `cin`. Функция `operator void*` может быть использована для проверки конца файла в объекте ввода вместо явного вызова функции-элемента `eof`.

Если файл открылся успешно, то программа начинает обрабатывать данные. Следующий оператор запрашивает пользователя о вводе различных полей каждой записи или информацию о конце файла, если ввод данных завершен:

```
cout << "Введите счет, имя и баланс." << endl
 << "Введите EOF – для окончания ввода." << endl << " ? " ;
```

Рис. 14.6 дает список комбинаций клавиш для ввода признака конца файла в различных системах.

### Строка

```
while (cin >> account >> name >> balance)
```

вводит каждый набор данных и определяет, не введен ли признак конца файла. Когда достигнут конец файла или вводятся неверные данные, операция извлечь из потока `>>` возвращает 0 (обычно операция извлечь из потока возвращает `cin`) и оператор `while` завершает свою работу. Пользователь должен вводить признак конца файла, чтобы сообщить программе о том, что ввод необходимых данных завершен. Маркер конца файла устанавливается, когда пользователь вводит ключевую комбинацию конца файла. Оператор `while` продолжает выполнение цикла до тех пор, пока не будет введен маркер конца файла.

### Оператор

```
outClientFile << account << " " << name
 << " " << balance << endl;
```

записывает набор данных в файл `"clients.dat"`, используя операцию поместить в поток `<<` и объект `outClientFile`, связанный с файлом в начале программы. Может быть проведена выборка данных с помощью программы, предназначеннной для чтения файла (см. раздел 14.5). Заметим, что файл, приведенный на рис. 14.4, является текстовым файлом. Он может быть прочитан с помощью любого текстового редактора.

Как только вводится признак конца файла, функция `main` завершается. Это приводит к тому, что объект `outClientFile` уничтожается вызовом его деструктора, который закрывает файл `clients.dat`. Объект `ofstream` может быть явным образом закрыт программистом с помощью функции-элемента `close`:

```
outClientFile.close();
```

| Операционная система | Комбинация клавиш             |
|----------------------|-------------------------------|
| UNIX                 | <code>&lt;ctrl&gt; + d</code> |
| IBM PC и совместимые | <code>&lt;ctrl&gt; + z</code> |
| Macintosh            | <code>&lt;ctrl&gt; + d</code> |
| VAX (VMS)            | <code>&lt;ctrl&gt; + z</code> |

Рис. 14.6. Комбинации клавиш для различных популярных операционных систем

### Совет по повышению эффективности 14.1

Закрывайте явным образом каждый файл, как только станет понятным, что программа не будет обращаться к этому файлу снова. Это поможет сократить используемые ресурсы, которые программа продолжает потреблять длительное время после того, когда ей уже не надо обращаться к данному файлу. Этот прием делает программу также более ясной.

В примере выполнения программы на рис. 14.4 пользователь вводит сведения по пяти счетам и затем сообщает, что ввод данных завершен, с помощью ввода признака конца файла (^Z появляется на экранах компьютеров, совместимых с IBM PC). Для проверки того, что файл создан успешно, в следующем разделе мы создадим программу чтения файла и печати его содержимого.

## **14.5. Чтение данных из файла последовательного доступа**

Данные сохраняются в файлах так, чтобы их можно было найти и обработать, когда в этом возникнет необходимость. В предыдущем разделе было показано, как создать файл последовательного доступа. В этом разделе обсудим, как последовательно считывать данные из файла.

Программа, приведенная на рис. 14.7, считывает записи из файла "clients.dat", созданного программой на рис. 14.4, и печатает содержимое записей. Файлы открываются для ввода путем создания объекта класса **ifstream**. Объекту передаются два аргумента — имя файла и режим открытия файла. Объявление

```
ifstream inClientFile("clients.dat", ios::in);
```

создает объект **inClientFile** класса **ifstream** и связывает с ним файл **clients.dat**, то есть открывает его для ввода. Аргументы в круглых скобках передаются конструктору класса **ifstream**, который открывает файл и устанавливает с ним «линию связи». Объекты класса **ifstream** открываются для ввода по умолчанию, так что для открытия файла **clients.dat** на ввод может быть использован оператор

```
ifstream inClientFile("clients.dat");
```

Так же, как и в случае объекта класса **ofstream**, объект класса **istream** может быть создан без открытия какого-то файла, а файл может быть связан с ним позднее.

### Хороший стиль программирования 14.1

Открывайте файл для только для ввода (используя **ios::in**), если содержимое файла не должно быть модифицировано. Это способствует предотвращению непреднамеренной модификации содержимого файла. Это пример принципа наименьших привилегий.

Программа использует условие **! inClientFile** для определения того, успешно ли открыт файл, до попытки осуществить выборку из этого файла. Стока

```
while (inClientFile >> account >> name >> balance)
```

читает из файла набор данных (т.е. запись). После первого выполнения этого оператора переменная `account` имеет значение 100, переменная `name` имеет значение "JONES", а переменная `balance` — значение 24.98. Всякий раз, когда выполняется приведенный оператор, в переменные `account`, `name` и `balance` считывается следующая запись из файла. Записи выводятся на экран с помощью функции `outputLine`, которая использует параметризованные манипуляторы потока для форматирования данных, изображаемых на экране. Когда достигается конец файла, входная последовательность в операторе `while` возвращает 0 (а обычно возвращается поток `inClientFile`), файл закрывается с помощью деструктора класса `ifstream` и программа завершается.

```
// fig14_7.cpp
// Чтение и печать последовательного файла
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>

void outputLine(int, char*, float);

main()
{
 ifstream inClientFile("clients.dat", ios::in);

 if (! inClientFile) {
 cerr << "Файл не может быть открыт" << endl;
 exit (1);
 }

 int account;
 char name[10];
 float balance;

 cout << setiosflags(ios::left) << setw(10) << "Счет"
 << setw(13) << "Имя" << "Баланс" << endl;

 while (inClientFile >> account >> name >> balance)
 outputLine(account, name, balance);

 return 0;
}
void outputLine(int acct, char *name, float bal)
{
 cout << setiosflags(ios::left) << setw(10) << acct
 << setw(13) << name << setw(7) << setprecision(2)
 << setiosflags(ios::showpoint | ios::right)
 << bal << endl;
}
```

---

| Счет | Имя   | Баланс |
|------|-------|--------|
| 100  | Jones | 24.98  |
| 200  | Doe   | 345.67 |
| 300  | White | 0.00   |
| 400  | Stone | -42.16 |
| 500  | Rich  | 224.62 |

Рис. 14.7. Чтение и печать последовательного файла

Для последовательного поиска данных в файле программа обычно начинает чтение данные с начала файла и читает все данные последовательно до тех пор, пока не будут найдены требуемые данные. Это может привести к необходимости обрабатывать файл последовательно несколько раз в течение выполнения программы (каждый раз с начала файла). Как класс *istream*, так и класс *ostream* содержат функции-элементы для позиционирования *указателя позиции файла* (это порядковый номер следующего байта в файле, который должен быть считан или записан). Этими функциями-элементами являются *seekg* (позиционировать для извлечения из потока) для класса *istream* и *seekp* (позиционировать для помещения в поток) для класса *ostream*. Любой объект класса *istream* имеет так называемый указатель «*get*», который показывает номер в файле очередного вводимого байта; а любой объект класса *ostream* имеет указатель «*set*», который показывает номер в файле очередного выводимого байта. Оператор

```
inClientFile.seekg(0);
```

позиционирует указатель позиции файла на начало файла (позиция 0), присоединенного к *inClientFile*. Аргумент функции *seekg* обычно является целым типа *long*. Второй аргумент, который может быть задан, показывает так называемое *направление позиционирования*. Направление позиционирования может быть *ios::beg* (по умолчанию) для позиционирования относительно начала потока, *ios::cur* — для позиционирования относительно текущей позиции в потоке и *ios::end* — для позиционирования относительно конца потока. Указатель позиции файла является целым числом, которое устанавливает позицию в файле как число байтов от начальной позиции в файле (иногда это называют *смещением* от начала файла). Приведем несколько примеров позиционирования указателя позиции в файле для извлечения из потока:

```
// Позиционирование fileObject на n-ый байт
// полагаем ios::beg
fileObject.seekg(n);

// Позиционирование fileObject на n байтов вперед
fileObject.seekg(n, ios::cur);

// Позиционирование fileObject на y-ый байт от конца файла
fileObject.seekg(y, ios::end);

// Позиционирование fileObject на конец файла
fileObject.seekg(0, ios::end);
```

Те же самые операции могут быть выполнены с помощью функции-элемента *seekg* класса *ostream*. Функции-элементы *tellg* и *tellp* возвращают текущие позиции соответственно указателя взять из потока «*get*» и указателя поместить в поток «*set*». Следующий оператор присваивает переменной *location* типа *long* значение указателя «*get*».

```
location = fileObject.tellg();
```

Программа на рис. 14.8 позволяет менеджеру по кредитам отображать на экране информацию для клиентов с нулевым балансом (т.е. клиентов, у которых нет перед компанией задолженности), информацию по кредитному балансу (т.е. клиентов, которым должна компания) и информацию

по дебетовому сальдо (т.е. клиентов, у которых имеется задолженность перед компанией за товары и услуги, полученные в прошлом). Программа отображает меню и позволяет менеджеру по кредитам вводить одну из трех опций получения соответствующей информации по кредитам. Опция 1 выводит список счетов с нулевым балансом. Опция 2 выводит список счетов с кредитным балансом. Опция 3 выводит список счетов с дебетовым сальдо. Опция 4 завершает выполнение программы. Пример вывода приведен на рис. 14.9.

```
// fig14_8.cpp
// Программа запроса кредитной информации
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>

void outputLine(int, char*, float);

main ()
{
 ifstream inClientFile ("clients.dat ", ios:: in);

 if (! inClientFile) {
 cerr << "Файл не может быть открыт" << endl;
 exit(1);
 }

 cout << "Запрос на ввод" << endl
 << " 1 - Список счетов с нулевым балансом" << endl
 << " 2 - Список счетов в кредитном балансом" << endl
 << " 3 - Список счетов с дебетовым сальдо" << endl
 << " 4 - Конец счета" << endl << "? ";

 int request;
 cin >> request;

 while (request != 4) {
 int account;
 char name[10];
 float balance;

 inClientFile >> account >>name >>balance;

 switch (request) {
 case 1:
 cout << endl
 << "Счета с нулевым балансом:" << endl;
 while (! inClientFile.eof()) {
 if (balance == 0)
 outputLine(account, name, balance);
 inClientFile >> account >> name >> balance;
 }
 break;
 }
 }
}
```

Рис. 14.8. Программа запроса кредитной информации (часть 1 из 2)

```

 case 2:
 cout << endl << "Счета с кредитовым балансом"
 << endl;

 while (! inClientFile.eof ()) {
 if (balance < 0)
 outputLine(account, name, balance);

 inClientFile >> account >> name >> balance;
 }

 break;
 case 3:
 cout << endl << "Счета с дебетовым сальдо "
 << endl;

 while (! inClientFile.eof ()) {
 if (balance > 0)
 outputLine(account, name, balance);

 inClientFile >> account >> name >> balance;
 }

 break;
 }
 inClientFile.clear(); // Установка на начало файла
 inClientFile.seekg(0); //
 cout << endl << "?";
 cin >> request;
}

cout << "Конец счета." << endl;
return 0;
}

void outputLine(int acct, char *name, float bal)
{
 cout << setiosflags(ios::left) << setw(10) << acct
 << setw(13) << name << setw(7) << setprecision(2)
 << setiosflags(ios::showpoint | ios::right)
 << bal << endl;
}

```

**Рис. 14.8.** Программа запроса кредитной информации (часть 2 из 2)

## 14.6. Обновление файлов последовательного доступа

Данные, которые форматируются и записываются файл последовательного доступа, как показано в разделе 14.4, не могут быть модифицированы без риска разрушения других данных в файле. Например, если имя «White» требуется заменить на «Worthington», то прежнее имя не может быть просто перезаписано. Запись для White была помещена в файл как

300 White 0.00

Если эта запись перезаписывается начиная с той же самой позиции в файле, но имеет более длинное имя, то новая запись принимает вид:

300 Worthington 0.00

Новая запись содержит на шесть символов больше, чем первоначальная запись. Следовательно, символы после второго символа «о» в имени «Worthington» будут записаны поверх начала следующей записи в файле. Проблема в том, что в модели форматированного ввода-вывода, используемой операциями поместить в поток << и извлечь из потока >>, поля и, следовательно, записи могут быть различных размеров. Например, 7, 14, -117, 2074 и 27383 — все эти числа являются целыми типа int, и каждое из них в «сыром виде» хранится в одинаковом количестве байтов, но когда эти целые числа выводятся как форматированный текст на экран или в файл на диске, то они занимают поля разных размеров. Следовательно, модель форматированного ввода-вывода обычно не применима для обновления записей на месте.

**Запрос на ввод**

- 1 - Список счетов с нулевым балансом
- 2 - Список счетов с кредитным балансом
- 3 - Список счетов с дебетовым сальдо
- 4 - Конец счета

? 1

**Счета с нулевым балансом:**

300 White 0.00

? 2

**Счета с кредитным балансом:**

400 Stone -42.16

? 3

**Счета с дебетовым сальдо:**

|     |       |        |
|-----|-------|--------|
| 100 | Jones | 24.98  |
| 200 | Doe   | 345.67 |
| 500 | Rich  | 224.62 |

? 4

**Конец счета.**

Рис. 14.9. Пример вывода программы запроса кредитной информации на рис. 14.8

Такое обновление может быть реализовано, но оно затруднено. Например, для того, чтобы изменить предыдущее имя, следует записи до 300 White 0.00 в файле последовательного доступа скопировать в новый файл, затем записать в этот новый файл обновленную запись, а затем скопированы в новый файл записи после 300 White 0.00. Это требует обработки всех записей в файле при обновлении одной записи. Такой метод может быть приемлемым только в случае обновления в одном проходе многиз записей.

## 14.7. Файлы произвольного доступа

До сих пор мы рассказывали, как создавать файлы последовательного доступа и определять с их помощью местоположение требуемой информа-

ции. Файлы последовательного доступа являются неподходящими для приложений с так называемым «немедленным доступом», в которых конкретная запись информации должна быть локализована немедленно. Такими распространенными приложениями с немедленным доступом являются, например, системы резервирования авиабилетов, банковские системы, система терминалов для производства платежей в месте совершения покупок, банковские автоматы и другие типы *систем по обработке запросов*, которые требуют оперативного доступа к конкретным данным. Банк, в котором у вас имеется счет, может иметь сотни тысяч или даже миллионы клиентов, и тем не менее, когда вы пользуетесь банковским автоматом, ваш счет проверяется на наличие достаточных средств в течение нескольких секунд. Этот тип немедленного доступа становится возможным с помощью *файлов произвольного доступа*. Отдельные записи файла произвольного доступа могут быть доступны непосредственно (и быстро) без поиска среди других записей.

Как уже было сказано, C++ не налагает требований на структуру файлов. Так что приложение, в котором предполагается использовать файлы произвольного доступа, должно буквально создать их. Для создания файлов произвольного доступа может быть использовано множество методов. Может быть наиболее простым из них является требование, чтобы все записи в файле были одинаковой фиксированной длины.

Использование записей с фиксированной длиной дает возможность программе легко определять точное местоположение любой записи относительно начала файла как функцию от размера записи и ключа записи. В дальнейшем мы скоро увидим, как это облегчает доступ к отдельным записям даже в больших по размерам файлах.

Рис. 14.10 демонстрирует представление в C++ файла произвольного доступа, образованного из записей фиксированной длины (длина каждой записи 100 байт). Файл произвольного доступа подобен железнодорожному поезду со многими вагонами, одни из которых заняты, а другие свободны.

Данные могут быть вставлены в файл прямого доступа без разрушения других данных файла. Данные, которые уже в нем хранятся, могут быть изменены или удалены без перезаписи всего файла. В следующих разделах будет объяснено, каким образом можно создавать файл произвольного доступа, вводить в него данные, считывать данные как последовательно, так и произвольно, обновлять и удалять данные, в которых нет необходимости.

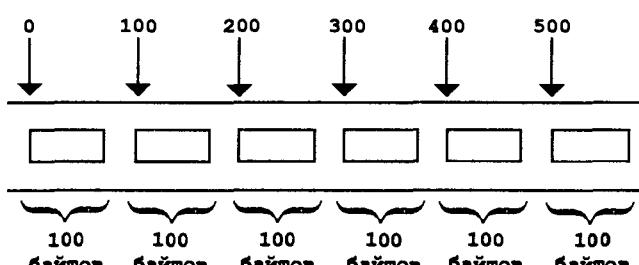


Рис. 14.10. Представление файла произвольного доступа на языке C++

## 14.8. Создание файла произвольного доступа

Функция-элемент `write` класса `ostream` выводит фиксированное число байтов, начиная от заданного места в памяти, в заданный поток. Когда поток связан с файлом, данные пишутся, начиная с позиции в файле, определяемой при помощи указателя позиции файла `«put»`. Функция-элемент `read` класса `istream` вводит фиксированное число байтов из заданного потока в область памяти, начиная с указанного адресса. Если поток связан с файлом, байты вводятся начиная с позиции в файле, определенной при помощи указателя позиции файла `«get»`.

Теперь при записи в файл переменной целого типа `number` вместо использования оператора

```
outFile << number;
```

который может напечатать для 4-байтовой целой переменной от 1 до 11 символов (до 10 разрядов плюс знак, каждый из которых требует 1 байта памяти), можно использовать оператор

```
outFile.write((char *) &number, sizeof(number));
```

который всегда записывает 4 байта (на компьютере с 4-байтовыми целыми). Функция `write` ожидает первый аргумент типа `char *` и поэтому мы использовали приведение к типу `(char *)`. Второй аргумент функции `write` является целым типа `size_t` и определяет число байтов, которые должны быть записаны. Как будет видно в дальнейшем, функция `read` класса `istream` может быть использована для последующего чтения 4 байтов обратно в переменную целого типа `number`.

Программа обработки файла произвольного доступа в редких случаях записывает единственное поле в файл. Обычно программы записывают за раз по одному объекту типа `struct` или `class`, как показано в примерах, приведенных ниже.

Рассмотрим постановку следующей задачи:

*Создайте программу по обработке счетов по кредитам, способную хранить до 100 записей фиксированной длины, для компании, которая может иметь до 100 клиентов. Каждая запись должна состоять из номера счета, который будет использован в качестве ключа записи, фамилии и имени клиента и его баланса. Программа должна быть способна обновлять счет, вставлять новый счет, уничтожать счет и выводить список всех записей по счетам в форматированный текстовый файл для их последующей печати.*

Следующие несколько разделов знакомят с методами, необходимыми для создания данной программы по обработке счетов по кредитам. Рис. 14.11 демонстрирует открытие файла произвольного доступа, описание формата записи с помощью `struct` и запись данных на диск. Эта программа задает начальные значения всем 100 записям файла "credit.dat" в виде незаполненных записей типа `struct`, используя функцию `write`. Каждая незаполненная запись типа `struct` содержит номер счета 0, нулевую строку (представляемую при помощи пустых кавычек) в качестве фамилии и имени и 0.0 в качестве баланса. В файл первоначально записано столько незаполненных записей, сколько счетов предполагается хранить, для того, чтобы в дальнейшем программа могла определять, содержит ли любая запись данные, или она является незаполненной.

```

// fig14_11.cpp
// Последовательность создания файла произвольного доступа
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

struct clientData {
 int acctNum;
 char lastName[15];
 char firstName[10];
 float balance;
};

main ()
{
 ofstream outCredit("credit.dat", ios::out);

 if (! outCredit) {
 cerr << "Файл не может быть открыт." << endl;
 exit(1);
 }

 clientData blankClient = {0, "", "", 0.0};

 for (int i = 1; i <=100; i++)
 outCredit.write((char *) &blankClient, sizeof(blankClient));

 return 0;
}

```

**Рис. 14.11.** Последовательность создания файла произвольного доступа

На рис. 14.11 оператор

```
outCredit.write((char *) &blankClient, sizeof(blankClient));
```

вызывает структуру **blankClient** размером **sizeof(blankClient)**, чтобы осуществить запись в файл **credit.dat**, связанный с объектом **outCredit** класса **ofstream**. Вспомним, что оператор **sizeof** возвращает размер в байтах объекта, заключенного в скобки (см. главу 5).

## 14.9. Произвольная запись данных в файл произвольного доступа

Программа, приведенная на рис. 14.12, записывает данные в файл **"credit.dat"**. Она использует комбинацию функций **seekp** и **write** класса **ostream** для сохранения данных в точно определенном месте в файле. Функция **seekp** устанавливает указатель файла «put» в заданную позицию в файле, а затем функция **write** выводит данные. Пример выполнения показан на рис. 14.13.

Оператор

```
outCredit.seekp((client.acctNum - 1) * sizeof(client));
```

устанавливает указатель файла «put» для объекта **outCredit** в позицию байта, вычисленного посредством (**client.acctNum - 1**) \* **sizeof(client)**. Поскольку номер счета лежит в пределах от 1 до 100, то при определении позиции записи из номера счета вычитается 1. Таким образом, для записи 1 указатель позиции файла устанавливается на байт файла 0. Отметим, что объект **outCredit** класса **ofstream** открывается в режиме открытия файла **ios::ate**. Указатель позиции файла «put» первоначально устанавливается на конец файла, но данные могут быть записаны в любое место в файле.

```
// fig14_12.cpp
// Запись в файл произвольного доступа

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

struct clientData { int acctNum;
 char lastName[15];
 char firstName[10];
 float balance;
};

main()
{
 ofstream outCredit("credit.dat", ios::ate);

 if (! outCredit) {
 cerr << "Файл не может быть открыт." << endl;
 exit(1);
 }

 cout << "Введите номер счета "
 << "(от 1 до 100, 0 - конец ввода)" << endl << "? ";
 clientData client;
 cin >> client.acctNum;

 while (client.acctNum > 0 && client.acctNum <= 100) {
 cout << "Введите фамилию, имя, баланс" << endl << "? ";

 cin >> client.lastName >> client.firstName
 >> client.balance;
 outCredit.seekp((client.acctNum - 1) * sizeof(client));
 outCredit.write((char *)&client, sizeof(client));
 cout << "Введите номер счета " << endl << "? ";
 cin >> client.acctNum;
 }
}

return 0;
}
```

Рис. 14.12. Произвольная запись данных в файл произвольного доступа

```

Ведите номер счета (от 1 до 100, 0 - конец ввода)
? 37
Ведите фамилию, имя, баланс
? Barker Doug 0.00
Ведите номер счета
? 29
Ведите фамилию, имя, баланс
? Brown Nancy -24.54
Ведите номер счета
? 96
Ведите фамилию, имя, баланс
? Stone Sam 34.98
Ведите номер счета
? 88
Ведите фамилию, имя, баланс
? Smith Dave 258.34
Ведите номер счета
? 33
Ведите фамилию, имя, баланс
? Dunn Stacey 314.33
Ведите номер счета
? 0

```

Рис. 14.13. Пример выполнения программы, приведенной на рис. 14.12

## 14.10. Последовательное чтение данных из файла произвольного доступа

В предыдущих разделах был создан файл произвольного доступа и были записаны данные в этот файл. В этом разделе создадим программу, которая последовательно читает файл от начала до конца, и печатает те записи, которые содержат данные. Такие программы дают определенные выгоды. Попробуйте определить, в чем они заключаются; мы вернемся к этому в конце данного раздела.

Функция `read` класса `istream` вводит в объект определенное число байтов с текущей позиции в указанном потоке. Например, оператор

```
inCredit.read((char *) &client, sizeof(client));
```

из программы на рис. 14.4 считывает число байтов, определяемое при помощи `sizeof(client)` из файла, связанного с объектом `inCredit` класса `ifstream`, и сохраняет данные в структуре `client`. Заметим, что функция `read` требует первый аргумент типа `char *`. Поскольку `&client` имеет тип `clientData *`, то `&client` должен быть приведен к типу `char *`.

Программа на рис. 14.14 последовательно читает каждую запись в файле "credit.dat", проверяет, содержатся ли в ней данные, и выводит на экран форматированные выходные данные для записей, содержащих данные. Условие

```
! inCredit.eof()
```

использует функцию-элемент `eof` объекта `ios` для определения конца файла и вызывает завершение выполнения структуры `while`. Введенные данные из файла выводятся на экран функцией `outputLine`, которая принимает два аргумента — объект `ostream` и структуру `clientData`.

```
// fig14_14.cpp
// Последовательное чтение из файла произвольного доступа
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>

struct clientData {
 int acctNum;
 char lastName[15];
 char firstName[10];
 float balance;
};

void outputLine(ostream&, clientData);

main()
{
 ifstream inCredit("credit.dat", ios::in);

 if (! inCredit) {
 cerr << "Файл не может быть открыт." << endl;
 exit(1);
 }

 cout << setiosflags(ios::left) << setw(6) << "Счет"
 << setw(16) << "Фамилия" << setw(11)
 << "Имя" << setiosflags(ios::right)
 << setw(10) << "Баланс" << endl;

 clientData client;

 inCredit.read((char *)&client, sizeof(clientData));

 while (! inCredit.eof()) {

 if (client.acctNum != 0)
 outputLine(cout, client);

 inCredit.read((char *)&client, sizeof(clientData));
 }

 return 0;
}

void outputLine(ostream &output, clientData c)
{
 output << setiosflags(ios::left) << setw(6) << c.acctNum
 << setw(16) << c.lastName << setw(11) << c.firstName
 << setw(10) << setprecision(2)
 << setiosflags(ios::showpoint | ios::right)
 << c.balance << endl;
}
```

Рис. 14.14. Последовательное чтение с помощью файла произвольного доступа  
(часть 1 из 2)



| Счет | Фамилия | Имя    | Баланс |
|------|---------|--------|--------|
| 29   | Brown   | Nancy  | -24.54 |
| 33   | Dunn    | Stacey | 314.33 |
| 37   | Barker  | Doug   | 0.00   |
| 88   | Smith   | Dave   | 258.34 |
| 96   | Stone   | Sam    | 34.98  |

Рис. 14.14. Последовательное чтение с помощью файла произвольного доступа (часть 2 из 2)

Итак, какие это дает дополнительные преимущества? Если вы посмотрите результаты вывода, то заметите, что записи перечислены в виде упорядоченной последовательности (по номеру счета). Это является простым следствием того, что мы хранили эти записи в файле, используя методы прямого доступа. По сравнению с пузырьковой сортировкой (глава 4) сортировка с использованием методов прямого доступа заведомо быстрее. Скорость, достигаемая при создании файла, вполне достаточна для сохранения всех мыслимых данных. Правда, файл может оказаться сильно разреженным, что приводит к нерациональному расходу памяти. Таким образом, это еще один пример компромисса в соотношении «память — время»: использование больших объемов памяти позволяет создать более быстрый алгоритм сортировки.

## 14.11. Пример: программа обработки запросов

Теперь рассмотрим реальную программу обработки запросов (рис. 14.15), использующую файл произвольной выборки для действительно «мгновенного» доступа. Программа оперирует с информацией по банковским счетам. Программа обновляет существующие счета, добавляет новые счета, уничтожает старые, создает текстовый файл для форматированной печати всех текущих счетов. Допустим, что выполнена программа на рис. 14.11 и создан файл credit.dat, а программой на рис. 14.12 выполнена вставка в файл начальных данных.

Программа имеет пять опций. Опция 1 вызывает функцию `textFile`, которая создает текстовый файл с именем `print.txt` для последующей форматированной печати. Функция `textFile` принимает объект `fstream` в качестве аргумента, который используется при вводе данных из файла `credit.dat`. Функция `textFile` использует функцию-элемент `read` класса `istream` и методы последовательного доступа к файлу (см. рис. 14.14) для ввода данных из файла `credit.dat`. Функция `outputLine`, которая была обсуждена в разделе 14.10, используется для вывода данных в файл `print.txt`. Заметим, что функция `textFile` использует функцию-элемент `seekg` класса `istream` для того, чтобы указатель позиции указывал на начало файла. После выбора опции 1 файл `print.txt` будет содержать:

| Счет | Фамилия | Имя    | Баланс |
|------|---------|--------|--------|
| 29   | Brown   | Nancy  | -24.54 |
| 33   | Dunn    | Stacey | 314.33 |
| 37   | Barker  | Doug   | 0.00   |
| 88   | Smith   | Dave   | 258.34 |
| 96   | Stone   | Sam    | 34.98  |

Опция 2 вызывает функцию `updateRecord` для обновления счета. Функция будет обновлять только существующую запись, поэтому функция сначала определяет, не является ли указанная запись незанятой. Запись считывается в структуру `client` с помощью функции-элемента `read` класса `istream`, затем `client.acctNum` сравнивается с нулем, чтобы определить, содержит ли запись какую-либо информацию. Если `client.acctNum` — нуль, то печатается сообщение, констатирующее, что запись является пустой, и на экран выводятся элементы меню. Если запись содержит какие-либо сведения, то функция `updateRecord` выводит запись на экране, используя функцию `outputLine`, затем вводит данные, вычисляет новый баланс и перезаписывает данных в файле. Типичный вывод для этой опции приведен ниже:

```
Введите счет , который следует обновить (1 - 100): 37
37 Barker Doug 0.00
```

```
Введите расход (+) или доплату (-): +87.99
37 Barker Doug 87.99
```

Опция 3 вызывает функцию `newRecord` для добавления нового счета в файл. Если пользователь вводит номер существующего счета, то `newRecord` выводит сообщение о том, что счет уже имеется, и выводит на экран элементы меню. Эта функция добавляет новый счет таким же образом, как программа на рис. 14.12. Типичный вывод для опции 3 следующий:

```
Введите новый номер счета (1 - 100): 22
Введите фамилию, имя, баланс
? Johnston Sarah 247.45
```

Опция 4 вызывает функцию `deleteRecord` для удаления записи из файла. Пользователю печатается приглашение ввести номер счета. Только существующая запись может быть удалена. Поэтому, если указанный счет является пустым, выводится сообщение об ошибке. Если счет существует, то он заново инициализируется путем копирования в файл незаполненной записи (`blankClient`). На экран выводится сообщение о том, что запись удалена. Типичный вывод для опции 4 следующий:

```
Введите номер счета для удаления (1 - 100): 29
Счет № 29 удален
```

Опция 5 завершает выполнение программы. Программа показана на рис. 14.15. Файл "credit.dat" открывается путем создания объекта `fstream` для чтения и записи попеременным использованием режимов `ios::in` и `ios::out`.

```
// fig14_15. cpp
// Эта программа последовательно читает файл произвольного
// доступа, обновляет данные, уже записанные в файл, создает
// новые данные, которые размещаются в файле, и удаляет
// уже имеющиеся в файле старые данные
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
```

Рис. 14.15. Программа банковских счетов (часть 1 из 5)

```
struct clientData{
 int acctNum;
 char lastName[15];
 char firstName[10];
 float balance;
};

int enterChoice(void);
void textFile(fstream&);
void updateRecord(fstream&);
void newRecord(fstream&);
void deleteRecord(fstream&);
void outputLine (ostream&, clientData);

main()
{
 fstream inOutCredit("credit.dat", ios::in | ios::out);

 if (!inOutCredit) {
 cerr << "Файл не может быть открыт."
 << endl;
 exit (1);
 }

 int choice;

 while ((choice = enterChoice ()) != 5) {

 switch (choice) {
 case 1:
 textFile(inOutCredit);
 break;
 case 2:
 updateRecord(inOutCredit);
 break;
 case 3:
 newRecord(inOutCredit);
 break;
 case 4:
 deleteRecord(inOutCredit);
 break;
 default:
 cerr << "Incorrect choice" << endl;
 break;
 }

 inOutCredit. clear (); // сброс индикатора
 // конца файла
 }

 return 0;
}
```

Рис. 14.15. Программа банковских счетов (часть 2 из 5)

```

// Приглашение выбрать раздел меню
int enterChoice(void)

{
 cout << endl << "Выберите:" << endl
 << "1 - создание текстового форматированного файла счетов"
 << endl
 << " с именем \"print.txt\" для печати" << endl
 << "2 - изменение счета" << endl
 << "3 - добавление нового счета" << endl
 << "4 - удаление счета" << endl
 << "5 - конец работы" << endl << "? ";
}

int menuChoice;
cin >> menuChoice;
return menuChoice;
}

// Создание форматированного текстового файла для печати
void textFile(fstream &readFromFile)
{
 ofstream outPrintFile("print.txt", ios::out);

 if (!outPrintFile) {
 cerr << "Файл не может быть открыт." << endl;
 exit (1);
 }

 outPrintFile << setiosflags(ios::left) << setw(6) << "Счет"
 << setw(16) << "Фамилия" << setw(11) << "Имя"
 << setiosflags(ios::right) << setw(10) << "Баланс"
 << endl;
 readFromFile.seekg(0);

 clientData client;
 readFromFile.read((char *)&client, sizeof(client));

 while (!readFromFile.eof()) {
 if (client.acctNum != 0)
 outputLine(outPrintFile, client);

 readFromFile.read((char *) &client, sizeof (client));
 }
}

// Изменение баланса счета
void updateRecord(fstream &updateFile)
{
 int account;

 do {
 cout << "Введите счет, который следует обновить (1-100): ";
 cin >>account;
 } while (account < 1 || account > 100);
}

```

Рис. 14.15. Программа банковских счетов (часть 3 из 5)

```
updateFile.seekg((account - 1) * sizeof(client));

clientData client;
updateFile.read((char *)&client, sizeof(client));

if (client.acctNum != 0) {
 outputLine(cout, client);
 cout << endl << "Введите расход (+) или доплату (-): "

 float transaction;
 cin >> transaction;
 client.balance += transaction;
 outputLine(cout, client);
 updateFile.seekp((account - 1) * sizeof(client));
 updateFile.write((char *)&client, sizeof(client));
}
else
 cerr << "Счет № " << account
 << " не заполнен." << endl;
}

// Создание и вставка новой записи
void newRecord(fstream &insertInFile)
{
 cout << "Введите новый номер счета (1 - 100): ";

 int account;
 cin >> account;
 insertInFile.seekg((account - 1) + sizeof(client));

 clientData client;
 insertInFile.read((char *)&client, sizeof(client));

 if (client.acctNum == 0) {
 cout << "Введите фамилию, имя, баланс" << endl << "? ";
 cin >> client.lastName >> client.firstName >> client.balance;
 client.acctNum = account;
 insertInFile.seekp((account - 1) * sizeof(clientData));
 insertInFile.write((char *)&client, sizeof(clientData));
 }
 else
 cerr << "Счет № " << account
 << " уже содержит информацию." << endl;
}

// Удаление существующей записи
void deleteRecord(fstream &deleteFromFile)
{
 cout << "Введите номер счета для удаления (1 - 100): ";

 int account;
 cin >> account;
 deleteFromFile.seekg((account - 1) * sizeof(client));

 clientData client;
 deleteFromFile.read((char *)&client, sizeof(client));
```

Рис. 14.15. Программа банковских счетов (часть 4 из 5)

```

if (client.acctNum != 0) {
 clientData blankClient = (0, "", "", 0);

 deleteFromFile.seekp((account - 1) * sizeof(client));
 deleteFromFile.write((char *)&blankClient, sizeof(client));
 cout << "Счет № " << account << " удален." << endl;
}
else
 cout << "Счет № " << account << " пустой." << endl;
}

// Вывод строки с информацией о клиенте
void outputLine(ostream &output, clientData c)
{
 output << setiosflags(ios::left) << setw(6) << c.acctNum
 << setw(16) << c.lastName << setw(11) << c.firstName
 << setiosflags(ios::showpoint | ios::right)
 << setw(10) << setprecision(2) << c.balance << endl;
}

```

Рис. 14.15. Программа банковских счетов (часть 5 из 5)

## 14.12. Обработка потоков строк

В дополнение к стандартному потоку ввода-вывода и файловому потоку ввода-вывода в C++ имеется возможность для ввода из символьных массивов в память и вывода в символьный массив из памяти. Эта возможностям часто называется *форматированным вводом-выводом в память или обработкой потоков строк*.

Ввод символьного массива поддерживается при помощи класса **istrstream**. Вывод в символьный массив поддерживается с помощью класса **ostrstream**. Класс **istrstream** наследует функциональные возможности класса **istream**. Класс **ostrstream** наследует функциональные возможности класса **ostream**. Программы, которые используют форматированный ввод-вывод в память, должны включать заголовочные файлы **<strstream.h>** и **<iostream.h>**.

Одним из применений этих методов является проверка достоверности данных. Программа может читать строку целиком из входного потока в массив символов. Затем может подключаться процедура, проверяющая достоверность данных, корректирующая (или восстанавливающая) их, если это необходимо. А затем программа может обрабатывать вводимые из символьного массива данные, зная, что все они соответствующим образом отформатированы.

Вывод в символьный массив является прекрасным способом использования мощных возможностей форматирования потоков в C++. Данные могут быть подготовлены в символьном массиве, имитирующем формат экранного редактора. Такой массив может быть записан в файл на диск для сохранения изображения экрана.

Объект **ostrstream** может использоваться двумя способами. Первый способ использует динамически размещаемый символьный массив вывода, который пересыпается в объект **ostrstream**. Как только вывод завершен, объект может быть «заморожен» функцией-элементом **str**, чтобы в дальнейшем для вывода не надо было делать пересылок в этот объект. Функция-элемент **str** возвращает указатель типа **char \*** на начало объекта массива в памяти. Этому указателю можно присвоить значение переменной типа **char \*** и ссылаться на него как на любой символьный массив в C++.

Рис. 14.16 демонстрирует динамически размещаемый объект `ostrstream`. Программа создает объект `outputString` класса `ostream` и использует операцию поместить в поток для вывода набора строк и численных значений в объект. Манипулятор потока `ends` помещает нулевой символ ('\0') в конце вывода данных в `outputString`. Затем программа замораживает содержимое потока `outputString` функцией-элементом `str` и присваивает переменной указателю `result` указатель на начало массива в памяти. Далее массив выводится в виде строки. Программа проверяет состояние потока `outputString` перед тем, как поместить в него информацию. После того, как функция-элемент `str` вызвана, предпринимается следующая попытка поместить в поток.

Второй способ использования объекта класса `ostrstream` — передача ему трех аргументов для конструктора класса `ostrstream`: символьного массива, его длины и режима открытия потока (`ios::out` или `ios::app`). Если задан режим `ios::app`, то символьный массив воспринимается как строка, заканчивающаяся нулевым символом, и информация помещается, начиная с позиции этого нулевого символа.

```
// fig14_16.cpp
// Использование динамически размещаемого объекта
// класса ostrstream.
#include <iostream.h>
#include <strstream.h>
main()
{
 ostrstream outputString;
 char *s1 = "Вывод нескольких типов данных ",
 *s2 = "в объект класса ostrstream:",
 *s3 = "\n double: ",
 *s4 = "\n int: ",
 *s5 = "\надрес int: ",
 *result;

 double d = 123.4567;
 int i = 22;

 cout << "Состояние потока outputString"
 << " перед помещением в него данных: "
 << outputString.rdstate() << endl;

 outputString << s1 << s2 << s3 << d
 << s4 << i << s5 << &i << ends;
 result = outputString.str();

 outputString << "ПРОВЕРКА ПОМЕЩЕНИЯ В ПОТОК ПОСЛЕ ВЫЗОВА str ";

 cout << endl << "Состояние outputString после вызова str"
 << endl << "и следующей попытки помещения данных в поток:"
 << outputString.rdstate() << endl << endl
 << "Динамически созданная строка содержит: " << endl
 << endl << result << endl;

 return 0;
}
```

Рис. 14.16. Использование динамически размещаемого объекта класса `ostrstream` (часть 1 из 2)

Состояние потока `outputString` перед помещением в него данных: 0

Состояние `outputString` после вызова `str`  
и следующей попытки помещения данных в поток: 4

Динамически созданная строка содержит:

Вывод нескольких типов данных в объект класса `ostrstream`:

```
double: 123.4567
int: 22
адрес int: 0x8e80ffe0
```

Рис. 14.16. Использование динамически размещаемого объекта класса `ostrstream` (часть 2 из 2)

```
// fig14_17.cpp
// Демонстрация объекта класса ostrstream,
// использующего заранее определенный массива.
#include <iostream.h>
#include <strstream.h>

main ()
{
 const int size = 15;
 char buffer[size];

 ostrstream outputString(buffer, size, ios::out);
 cout << "Состояние потока outputString перед помещением
 в него данных: "
 << outputString.rdstate() << endl;

 outputString << "Проверка " << 123 << ends;

 cout <<"Содержимое буфера: " << endl << buffer;

 outputString << "ПОПЫТКА ВСТАВИТЬ ПОСЛЕ КОНЦА МАССИВА";
 cout << endl << endl << "Состояние outputString после попытки"
 << endl << "вставки после конца строки: "
 << outputString.rdstate() << endl;

 return 0;
}
```

Состояние потока `outputString` перед помещением в него данных: 0  
Содержимое буфера:  
Проверка 123

Состояние `outputString` после попытки  
вставки после конца строки: 4

Рис. 14.17. Демонстрация объекта класса `ostrstream`, использующего заранее определенный массив

**Рис. 14.17** демонстрирует использование объекта класса **ostrstream** с заранее определенным массивом. Объявление

```
ostrstream outputString(buffer, size, ios::out);
```

создает объект **ostrstream**, который будет использовать символьный массив **buffer** с количеством элементов **size** для хранения вывода, направляемого в поток **outputString**. Поскольку размер массива известен заранее, поток **outputString** может осуществить свой собственный поиск ошибок. Если предпринимается попытка вывести данные в поток **outputString**, причем данные должны храниться за концом массива **buffer**, тогда будет установлен **failbit** (это обсуждалось в главе 11) и операция вывода не выполнится. Программа выводит строку, целое и завершающий нулевой символ в **buffer**, а затем выводит содержимое **buffer**. Программа проверяет состояние потока **outputString** перед помещением в него информации и вслед за попыткой поместить информацию после конца массива **buffer**.

Объект **istrstream** вводит данные из символьного массива в памяти в переменные программы. Данные хранятся в объекте **istrstream** как символы; ввод из объекта **istrstream** работает аналогично вводу из любого файла или из стандартного входного потока. Завершающий нулевой символ интерпретируется этим объектом как **eof**.

**Рис. 14.18** демонстрирует ввод из объекта **istrstream**. Объявление

```
istrstream inputBuffer(input, size);
```

создает объект **inputBuffer** класса **istrstream**. Два аргумента задают массив (**input**), из которого данныечитываются, и число элементов в массиве (**size**). Входной массив содержит следующие данные:

Ведите тест 123 4.7 A

которые при считывании в программу в качестве входных данных состоят из двух строк ("Ведите" и "тест"), целого значения (123), значения с плавающей точкой (4.7) и символа ('A'). Эти данные вводятся в переменные **string1**, **string2**, **i**, **d** и **c** соответственно операцией извлечь из потока, а затем выводятся в **cout**. Программа проверяет состояние потока **inputBuffer** перед попыткой извлечения или после нее, пока не остается данных во входном массиве **input**.

```
// fig14_18.cpp
// Демонстрация ввода из объекта класса istrstream.
#include <iostream.h>
#include <strstream.h>

main()
{
 const int size = 80;
 char input [size] = "Ведите тест 123 4.7 A";

 istrstream inputBuffer(input, size);

 cout << "Состояние inputBuffer перед операцией извлечения: "
 << inputBuffer.rdstate() << endl << endl;
```

**Рис. 14.18.** Демонстрация ввода из объекта класса **istrstream** (часть 1 из 2)

```

char string1 [size], string2 [size];
int i;
double d;
char c;
inputBuffer >> string1 >> string2 >> i >> d >> c;

cout << "Извлекаются следующие данные " << endl
 << "из объекта класса istrstream:" << endl
 << " Стока: " << string1 << endl
 << " Стока: " << string2 << endl
 << "Integer: " << i << endl
 << " Double: " << d << endl
 << " Char: " << c << endl << endl;

// Попытка чтения из пустого потока
long l;
inputBuffer >> l;
cout << " Состояние inputBuffer после извлечения из пустого
 потока: "
 << inputBuffer.rdstate() << endl;

return 0;
}

```

**Состояние inputBuffer перед операцией извлечения:** 0

**Извлекаются следующие данные  
из объекта класса istrstream:**  
**Строка:** Введите  
**Строка:** тест  
**Integer:** 123  
**Double:** 4.7  
**Char:** A

**Состояние inputBuffer после извлечения из пустого потока:** 2

Рис. 14.18. Демонстрация ввода из объекта класса **istrstream** (часть 2 из 2)

## 14.13 Ввод-вывод объектов

В этой главе и главе 11 мы обсуждали объектно-ориентированный стиль ввода-вывода в языке C++. Но наши примеры были посвящены вводу-выводу традиционных типов данных, а не объектов определенных пользователем классов. В главе 8 мы увидели, как вводить и выводить объекты классов, используя перегрузку операций. Мы рассмотрели ввод объекта с помощью перегрузки операции `взять из потока >>` для соответствующих классов `istream`. Мы также рассмотрели вывод объекта с помощью перегрузки операции `поместить в поток <<` для соответствующих классов `ostream`. В обоих случаях вводились и выводились только данные-элементы объекта и в обоих случаях в форме, ориентированной на объекты некоего абстрактного типа данных. Функции-элементы объекта внутренне доступны в компьютере и сочетаются со значениями данных, когда эти данные вводятся посредством перегруженной операции `поместить в поток`.

Когда данные-элементы объекта выводятся в файл на диске, мы теряем, в известном смысле, информацию о типе объекта. Мы имеем только данные на диске, но не имеем информацию о типах этих данных. Если программе, которая собирается считывать эти данные, известно, какому типу объекта они соответствуют, тогда данные просто читаются в объект этого типа.

Интересная проблема возникает, когда мы храним объекты разных типов в одном и том же файле. Как мы можем различать их (или совокупность данных-элементов), когда мы их читаем в программе? Проблема, конечно, состоит в том, что объекты обычно не имеют полей типов (более детально это было рассмотрено в главе 9, «Виртуальные функции и полиморфизм»).

Один из подходов, который может быть использован для перегруженной операции вывода, это выводить перед каждой совокупностью данных-элементов, представляющих один объект, код типа. Тогда объект ввода будет всегда начинать с чтения поля кода типа и при помощи оператора `switch` вызывать соответствующую перегруженную функцию. Хотя этому решению не достает элегантности полиморфного программирования, оно обеспечивает работоспособный механизм для сохранения объектов в файлах и их извлечения по мере необходимости.

## Резюме

- Все элементы данных, обрабатываемые компьютером, сводятся к комбинациям нулей и единиц.
- Наименьший элемент данных в компьютере может принимать значения 0, либо 1. Такой элемент данных называется битом.
- Цифры, буквы и специальные символы называются символами. Множество всех символов, используемых для написания программ и представления элементов данных на конкретном компьютере, называется набором символов компьютера. Каждый символ в компьютерном наборе представляется как совокупность из восьми нулей и единиц (называемом байтом).
- Поле — это группа символов (или байтов), имеющая некоторый смысл.
- Запись — это группа связных полей.
- По крайней мере одно поле в записи выбирается в качестве ее ключа для идентификации записи как однозначно соответствующей определенному человеку или сущности, то есть делает ее уникальной среди других записей в файле.
- Последовательный доступ — наиболее распространенный метод доступа к данным в файле.
- Совокупность программ, предназначенных для создания баз данных и управления ими, называется системой управления базой данных — СУБД.
- В C++ каждый файл рассматривается как последовательный поток байтов.
- Каждый файл завершается некоторым машинно-зависимым маркером конца файла.
- Потоки обеспечивают каналы связи между файлами и программами.

- Для осуществления ввода-вывода файлов в C++ в программу должны быть включены заголовочные файлы `<iostream.h>` и `<fstream.h>`. Файл `<fstream.h>` включает определения классов потоков `ifstream`, `ofstream` и `fstream`.
- Файлы открываются просто путем обращения к объектам классов `ifstream`, `ofstream` и `fstream`.
- В C++ не предъявляется никаких требований к структуре файла. Таким образом, в C++ не существуют понятия, вроде «записи». Программист должен подбирать структуру файла, исходя из требований, предъявляемых конкретным приложением.
- Файлы открываются на вывод путем создания объекта класса `ofstream`. Этому объекту передаются два аргумента: имя файла и режим открытия файла. Для объекта класса `ofstream` режим открытия файла может быть либо `ios::out` — для вывода данных в файл, либо `ios::app` — для присоединения данных в конец файла. Существующие файлы, открываемые в режиме `ios::out`, усекаются — вся информация в них теряется. Несуществующие файлы создаются.
- Функция-операция `operator!` — элемент класса `ios`, возвращает ненулевое значение (истина), если либо `failbit`, либо `badbit` были установлены при выполнении для потока операции `open`.
- Функция-операция `operator void *` — элемент класса `ios` преобразует поток в указатель, который можно сравнивать с 0 (нулевым указателем). Если либо `failbit`, либо `badbit` были установлены для потока, то возвращается 0 (ложь).
- Программы могут не обрабатывать никаких файлов, либо обрабатывать один или несколько файлов. Каждый файл имеет уникальное имя и связан с соответствующим объектом файлового потока. Все функции по обработке файлов должны ссылаться на файл через соответствующий объект.
- Указатель «`get`» показывает позицию в файле, из которой производится последующий ввод, а указатель «`put`» показывает позицию в файле, в которую разместятся следующие выходные данные. Как класс `istream`, так и класс `ostream` имеют функции-элементы для позиционирования указателя позиции файла. Этими функциями являются функция-элемент `seekg` («`seek get`») для класса `istream` и функция-элемент `seekp` («`seek put`») для класса `ostream`.
- Функции-элементы `tellp` и `tellg` возвращают текущие позиции соответственно указателей «`put`» и «`get`».
- Удобным средством при разработке файлов произвольного доступа является применение записей только фиксированной длины. Используя этот метод, программа может быстро определить точное местоположение записи относительно начала файла.
- Данные могут быть помещены в файл произвольного доступа без разрушения других данных в файле. Данные могут быть обновлены или удалены без перезаписи всего файла.
- Функция-элемент `write` класса `ostream` выводит в указанный поток некоторое число байтов, начиная с заданной позиции в файле. Когда

поток связан с файлом, то данные записываются, начиная с позиции, определенной указателем «put».

- Функция-элемент **read** класса **istream** вводит некоторое число байтов из указанного потока в область памяти, начиная с заданного адреса. Байты берутся из потока, начиная с позиции, определенной указателем «get».
- Функция **write** ожидает первый аргумент типа **char \***, так что этот аргумент должен быть приведен к типу **char \***, если он является указателем другого типа. Второй аргумент является целым и задает число байтов, которые должны быть записаны.
- Выполняемая на этапе компиляции унарная операция **sizeof** возвращает размер в байтах объекта, указанного в круглых скобках; операция **sizeof** возвращает целое без знака.
- Функция-элемент **read** класса **istream** вводит в объект заданное число байтов из указанного потока; функция-элемент **read** ожидает первый аргумент типа **char \***.
- Функция-элемент **eof** класса **ios** определяет, достигнут ли признак конца файла в заданном потоке. Признак конца файла устанавливается, когда попытка считывания данных оканчивается неудачей.
- В C++ имеются средства ввода-вывода для ввода в массивы символов в памяти и вывода из массивов символов в памяти. Эти средства часто называются форматированным вводом-выводом в память или обработкой потоков строк.
- Ввод из массива символов поддерживается классом **istrstream**. Вывод в массив символов поддерживается классом **ostrstream**.
- Класс **istrstream** является производным от класса **istream**. Класс **ostrstream** производным от класса **ostream**.
- Программы, которые используют форматированный ввод-вывод в память, должны включать заголовочный файл **<strstream.h>** в дополнение к заголовочному файлу **<iostream.h>**.
- Применение объекта класса **ostrstream** может быть реализовано двумя способами. При первом способе объекту класса **ostrstream** передается динамически размещаемый массив символов, в который осуществляется вывод. Когда вывод в объект класса **ostrstream** завершен, этот объект может быть «заморожен» с помощью функции-элемента **str**. Функция-элемент **str** возвращает указатель типа **char \*** на начало массива в памяти.
- Второй путь использования объекта класса **ostrstream** — передача его конструктору трех аргументов: массива символов, размера этого массива и режима открытия файла (**ios::out** или **ios::app**). Данные выводятся в заданный массив. Если задан режим **ios::app**, то массив символов должен быть строкой, заканчивающейся нулевым символом и вставка данных начинается с этого нулевого символа, но ограничивается размером массива.
- Объект класса **istrstream** вводит данные из массива символов в памяти в переменные программы. Нулевой завершающий символ интерпретируется как конец файла.

## Терминология

|                                                               |                                                                |
|---------------------------------------------------------------|----------------------------------------------------------------|
| <code>cerr</code> (стандартный небуферизованный вывод ошибок) | обработка потока строк                                         |
| <code>cin</code> (стандартный ввод)                           | поле                                                           |
| <code>clog</code> (стандартный буферизованный вывод ошибок)   | поток                                                          |
| <code>cout</code> (стандартный вывод)                         | поток ввода                                                    |
| алфавит                                                       | режим открытия файла <code>ios::app</code>                     |
| база данных                                                   | режим открытия файла <code>ios::ate</code>                     |
| байт                                                          | режим открытия файла <code>ios::in</code>                      |
| бит                                                           | режим открытия файла <code>ios::noreplace</code>               |
| выходной поток                                                | режим открытия файла <code>ios::out</code>                     |
| двоичный разряд                                               | режим открытия файла <code>ios::trunc</code>                   |
| десятичная цифра                                              | символьное поле                                                |
| заголовочный файл <code>fstream.h</code>                      | система управления базой                                       |
| заголовочный файл <code>strstream.h</code>                    | данных (СУБД)                                                  |
| закрыть файл                                                  | специальный символ                                             |
| запись                                                        | указатель позиции файла                                        |
| иерархия данных                                               | усечение существующего файла                                   |
| имя файла                                                     | файл                                                           |
| класс <code>fstream</code>                                    | файл последовательного доступа                                 |
| класс <code>ifstream</code>                                   | файл произвольного доступа                                     |
| класс <code>istream</code>                                    | форматированный ввод-вывод                                     |
| класс <code>istrstream</code>                                 | в память                                                       |
| класс <code>ofstream</code>                                   | функция-элемент <code>close</code>                             |
| класс <code>ostream</code>                                    | функция-элемент <code>open</code>                              |
| класс <code>ostrstream</code>                                 | функция-элемент <code>operator void *</code>                   |
| ключ записи                                                   | функция-элемент <code>operator!</code>                         |
| конец файла                                                   | функция-элемент <code>seekg</code> класса                      |
| манипулятор потока <code>ends</code>                          | <code>istream</code>                                           |
| маркер конца файла                                            | функция-элемент <code>seekg</code> класса                      |
| набор символов                                                | <code>ostream</code>                                           |
| начальная точка позиционирования                              | функция-элемент <code>tellg</code> класса <code>istream</code> |
| <code>ios::beg</code>                                         | функция-элемент <code>tellp</code> класса                      |
| начальная точка позиционирования                              | <code>ostream</code>                                           |
| <code>ios::cur</code>                                         | функция-элемент потока строк <code>str</code>                  |
| начальная точка позиционирования                              | числовое поле                                                  |
| <code>ios::end</code>                                         |                                                                |

## Типичные ошибки программирования

- 14.1. Открытие существующего файла для вывода (`ios::out`) в то время, как пользователь на самом деле желает сохранить этот файл; содержимое файла отбрасывается без предупреждения.
- 14.2. Использование неверного объекта класса `ofstream` при ссылке на файл.
- 14.3. Не открывается файл перед попыткой сослаться на него в программе.

## Хороший стиль программирования

14.1. Открывайте файл для только для ввода (используя `ios::in`), если содержимое файла не должно быть модифицировано. Это способствует предотвращению непреднамеренной модификации содержимого файла. Это пример принципа наименьших привилегий.

## Советы по повышению эффективности

14.1. Закрывайте явным образом каждый файл, как только станет понятным, что программа не будет обращаться к этому файлу снова. Это поможет сократить используемые ресурсы, которые программа продолжает потреблять длительное время после того, когда ей уже не надо обращаться к данному файлу. Этот прием делает программу также более ясной.

## Упражнения для самопроверки

14.1. Заполните пробелы в следующих утверждениях:

- a) Все элементы данных, обрабатываемые компьютером, в конечном итоге сводятся к комбинациям \_\_\_\_\_.
- b) Наименьший элемент данных, который может обрабатываться компьютером, называется \_\_\_\_\_.
- c) \_\_\_\_\_ — это группа связных записей.
- d) Цифры, буквы и специальные символы называются \_\_\_\_\_.
- e) Группа связных файлов называется \_\_\_\_\_.
- f) Функция-элемент \_\_\_\_\_ классов файловых потоков `fstream`, `ifstream` и `ostream` закрывает файл.
- g) Функция-элемент \_\_\_\_\_ класса `istream` читает символ из заданного потока.
- h) Функции-элементы класса `istream` \_\_\_\_\_ и \_\_\_\_\_ читают строку из заданного потока.
- i) Функция-элемент \_\_\_\_\_ классов потоков `istream` и `ostream` открывает файл.
- j) Функция-элемент класса `istream` \_\_\_\_\_ обычно используется в приложениях для чтения данных из файла произвольного доступа.
- k) Функции-элементы \_\_\_\_\_ и \_\_\_\_\_ классов `istream` и `ostream` устанавливают соответствующий указатель позиции в заданную позицию соответственно во входном и выходном потоках.

14.2. Укажите, справедливы или нет следующие утверждения. Если они ошибочны, укажите почему.

- a) Функция-элемент `read` не может быть использована для чтения данных из объекта ввода `cin`.
- b) Программист обязан явным образом создавать объекты `cin`, `cout`, `cerr` и `clog`.

- c) Программа должна явным образом вызывать функцию `close`, чтобы закрыть файл, связанный с объектами `ifstream`, `ofstream` или `fstream`.
- d) Если указатель позиции файла показывает на позицию в последовательном файле, отличную от начала файла, то для считывания с начала файла он должен быть закрыт и заново открыт.
- e) Функция-элемент `write` класса `ostream` может записывать в стандартный поток вывода `cout`.
- f) Данные в файле последовательного доступа всегда обновляются без перезаписи соседних данных.
- g) Чтобы найти требуемую запись, необходимо просмотреть все записи в файле произвольного доступа.
- h) Записи в файлах произвольного доступа должны быть одной длины.
- i) Функции-элементы `seekg` и `seekp` проводят поиск относительно начала файла.

14.3. Предполагайте, что каждый из нижеперечисленных операторов относится к одной и той же программе.

- a) Напишите оператор, который открывает файл "oldmast.dat" для ввода; используйте объект `inOldMaster` класса `ifstream`.
- b) Напишите оператор, который открывает файл "trans.dat" для ввода; используйте объект `inTransaction` класса `ifstream`.
- c) Напишите оператор, который открывает (или создает) файл "newmast.dat" для вывода; используйте объект `outNewMaster` класса `ofstream`.
- d) Напишите оператор, который считывает запись из файла "oldmast.dat". Запись состоит из целого `accountNum`, строки `name` и числа с плавающей запятой `currentBalane`; используйте объект `inOldMaster` класса `ifstream`.
- e) Напишите оператор, который считывает запись из файла "trans.dat". Запись состоит из целого `accountNum` и числа с плавающей запятой `dollarAmount`; используйте объект `inTransaction` класса `ifstream`.
- f) Напишите оператор, который заносит запись в файл "newmast.dat". Запись состоит из целого `accountNum`, строки `name` и числа с плавающей запятой `currentBalance`; используйте объект `outNewMaster` класса `ofstream`.

14.4. Найдите ошибку и покажите, каким образом исправить ее в перечисленных ниже высказываниях:

- a) Файл "payables.dat", на который ссылаются с помощью объекта `outPayable` класса `ofstream`, не должен быть открыт.  
`outPayable << account << company << endl;`
- b) Следующий оператор должен читать запись из файла "payables.dat". Объект `inPayable` класса `ifstream` ссылается на этот файл,

а объект `inReceivable` класса `istream` ссылается на файл "receivable.dat".

```
inReceivable >> account >> company >> amount;
```

с) Файл "too.dat" должен быть открыт для добавления данных в файл без уничтожения текущих данных.

```
ofstream outTools("tools.dat", ios::out);
```

### Ответы на упражнения для самопроверки

**14.1.** а) единиц и нулей. б) бит. с) Файл. д) символами. е) базой данных. ф) `close`. г) `get`. х) `get`, `getline`. и) `open`. ж) `read`. к) `seekg`, `seekp`.

**14.2.** а) Неверно. Функция `read` может быть использована для чтения данных из любого объекта потока, производного от `istream`.

б) Неверно. Эти четыре потока создаются автоматически. Для использования потоков в файл должен быть включен заголовочный файл `<iostream.h>`. Он содержит объявления всех этих объектов потоков.

с) Неверно. Файлы закрываются, когда выполняются деструкторы объектов классов `ifstream`, `ofstream` или `fstream`, а это происходит, когда объекты потоков выходят из области действия или перед завершением выполнения программы; но, все же, хорошим стилем программирования является закрытие всех файлов явным образом с помощью функции `close`, когда уже нет потребности в этих файлах.

д) Неверно. Для установки указателей позиции «`put`» и «`get`» на начало файла могут быть использованы функции-элементы `seekg` и `seekp`.

е) Верно.

ф) Неверно. В большинстве случаев записи последовательного файла не имеют одинаковой длины. Следовательно, вполне возможно, что обновление записи приведет к необходимости перезаписать остальные данные.

г) Верно.

х) Неверно, но обычно, действительно, записи в файле произвольного доступа имеют одинаковую длину.

и) Неверно. Возможен поиск от начала файла, от его конца или от текущей записи файла.

**14.3.** а) `ifstream inOldMaster("oldmast.dat", ios::in);`

б) `ifstream Transaction("trans.dat", ios::in);`

с) `ofstream outNewMaster("newmast.dat", ios::out);`

д) `inOldMaster >> accountNum >> name >> currentBalance;`

е) `inTransaction >> accountNum >> dollarAmount;`

ж) `outNewMaster << accountNum << name << currentBalance;`

**14.4.** а) Ошибка: файл "payables.dat" должен быть открыт до попытки вывести данные в поток.

Исправление: используйте для открытия "payables.dat" на вывод функцию-элемент **open** класса **ostream**.

b) Ошибка: используется неверный объект **inPayable** класса **ifstream** для чтения из файла "payables.dat".

Исправление: используйте объект **inPayable** класса **istream** для обращения к файлу "payables.dat".

c) Ошибка: содержимое файла отбрасывается, потому что файл открыт для вывода (**ios::out**).

Исправление: для добавления данных в файл или откройте его для обновления (**ios::ate**), или откройте для добавления в конец (**ios::app**).

## Упражнения

14.5. Заполните пробелы в приведенных ниже примерах:

- a) Компьютеры хранят большие объемы данных в устройствах внешней памяти таких, как \_\_\_\_\_.
- b) \_\_\_\_\_ состоит из нескольких полей.
- c) Поле, которое может содержать только цифры, буквы и пробелы, называются \_\_\_\_\_ полем.
- d) Для обеспечения поиска заданных записей в файле одно поле в каждой записи выбирается в качестве \_\_\_\_\_.
- e) Подавляющее большинство сведений в компьютерной системе хранится в \_\_\_\_\_.
- f) Группа связных символов, имеющая некоторый смысл, называется \_\_\_\_\_.
- g) Объектами стандартных потоков, объявленных при помощи заголовочного файла <iostream.h>, являются \_\_\_\_\_, \_\_\_\_\_ и \_\_\_\_\_.
- h) Функция-элемент \_\_\_\_\_ класса **ostream** выводит символ в заданный поток.
- i) Функция-элемент \_\_\_\_\_ класса **ostream** обычно используется для записи данных в файл произвольного доступа.
- j) Функция-элемент \_\_\_\_\_ класса **istream** изменяет позицию указателя позиции файла.

14.6. Установите, какие из перечисленных ниже утверждений являются истинными, а какие — ложными (и для ложных, объясним причину).

- a) Впечатляющие функциональные возможности компьютера по существу сводятся к манипуляциям с нулями и единицами.
- b) Пользователи предпочтут манипулировать с битами вместо символов и полей, поскольку биты более компактны.
- c) Пользователи формируют программы и элементы данных символами; компьютеры затем манипулируют с этими данными и обрабатывают эти символы как группы из нулей и единиц.
- d) Почтовый пятизначный код является примером числового поля.
- e) Адрес человека обычно принимается в приложениях для компьютера как текстовое поле.

- f) Элементы данных представляются в компьютерах в виде иерархии данных, в которой элементы данных становятся все больше и сложнее по мере продвижения от полей к символам, затем к битам и т.д.
- g) Ключ записи идентифицирует запись как принадлежащую конкретному полю.
- h) Большинство организаций хранит всю свою информацию в единственном файле для облегчения процесса компьютерной обработки.
- i) Каждый оператор, который обрабатывает файл программы на C++, явным образом обращается к этому файлу по имени.
- j) Когда программа создает файл, он автоматически сохраняется для последующих ссылок.

**14.7.** В упражнении 14.3 читатель должен был записать ряд простых операторов. Фактически эти операторы формируют ядро важного типа программы по обработке файлов, а именно, программы согласования файлов. При обработке коммерческих данных обычно в каждой прикладной системе имеется несколько файлов. В счетах по дебиторской задолженности, в частности, обычно основной файл содержит детальную информацию о каждом клиенте, например, такую: имя клиента, адрес, номер телефона, баланс по неоплаченным счетам, кредитный лимит, условия дисконтирования, соглашение по контракту и, возможно, краткое изложение сведений о последних закупках и наличных платежах.

Когда поступает информация об операциях (например, совершены продажи, по почте получены счета по наличным платежам), она вводится прямо в файл. В конце определенного расчетного периода (некоторых компаниях — это месяц, в других — неделя или день) файл текущих операций (в упражнении 14.3 он назван "trans.dat") вносится в основной файл (названный "oldmast.dat" в упражнении 14.3) и таким образом обновляются записи счетов по покупкам и платежам. На время обновления основной файл переписывается в новый файл (названный "newmast.dat"), который затем используется в конце следующего расчетного периода для начала нового процесса обновления.

Программа согласования файлов сталкивается с определенными трудностями, которые не существуют в программах с единственным файлом. Например, согласование не всегда происходит. Клиент основного файла может не выполнить никаких сделок и платежей за данный расчетный период и, следовательно, ни одна запись для этого пользователя не появится в файле текущих операций. Аналогично, клиент, который совершил некоторые операции, может быть только что вошел в данное сообщество и тогда компания еще не создавала запись для этого клиента в основном файле.

Используйте операторы, написанные в упражнении 14.3, как основу для написания полноценной программы по кредитной задолженности с согласованием файлов. Используйте номер счета в каждом файле как ключ записи для целей согласования. Полагайте, что каждый файл является файлом с последовательным доступом с записями, хранимыми в порядке возрастания номера счета.

Когда происходит согласование (т.е. записи с одним номером счета имеются как в основном файле, так и в файле текущих сделок), добавляйте сумму в долларах из файла текущих сделок к сальдо в основном файле и заносите запись в "newmast.dat". (Полагайте, что расходы отображаются положительными суммами, а доплаты — отрицательными). Когда имеется основная запись для данного счета, но нет соответствующей записи по текущим сделкам, то просто запись основного файла записывается в файл "newmast.dat". Когда существует запись по текущим сделкам, но нет соответствующей основной записи, то печатайте сообщение «Нет соответствующей записи для счета номер...» (заполните многоточие номером текущей записи).

- 14.8.** После написания программы упражнения 14.7 напишите простую программу создания контрольных данных для ее тестирования. Используйте следующий пример данных:

Основной 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 текущих записей

| Номер счета | Сумма долларов |
|-------------|----------------|
| 100         | 27.14          |
| 300         | 62.11          |
| 400         | 100.56         |
| 900         | 82.17          |

- 14.9.** Выполните программу упражнения 14.7, используя файлы тестовых данных, созданные в упражнении 14.8. Отпечатайте новый основной файл. Проверьте, правильно ли обновлены счета.

- 14.10.** Возможно (и очень часто) имеется несколько текущих записей с одинаковым ключом записи. Это происходит из-за того, что клиент мог совершить за расчетный период несколько операций. Перепишите вашу программу сравнения счетов из упражнения 14.7, чтобы обеспечить возможность обработки нескольких текущих записей с одним ключом. Модифицируйте тестовые данные упражнения 14.8, включив в них следующие дополнительные текущие записи:

| Номер счета | Сумма долларов |
|-------------|----------------|
| 300         | 83.89          |
| 700         | 80.78          |
| 700         | 1.53           |

**14.11.** Напишите ряд операторов для выполнения каждой из приведенных ниже операций. Полагайте, что определена структура

```
struct person {
 char lastName[15];
 char firstName[15];
 char age[2];
};
```

и открыт соответствующий файл произвольного доступа.

- Инициализируйте файл "nameage.dat" со 100 записями, содержащими LastName = "unassigned", firstName = "" и age= "0".
- Ведите 10 фамилий, имен и соответствующие возрасты, запишите эти данные в файл.
- Обновите записи, которые имеют указанные сведения, а если таких сведений нет, то сообщите пользователю «Нет сведений».
- Удалите запись, которая содержит информацию, путем повторной инициализации этой записи.

**14.12.** Вы являетесь владельцем склада металлических изделий и нуждаетесь в инвентаризации, которая сказала бы вам, сколько всего различных изделий вы имеете, какое количество каждого из них у вас на руках и стоимость каждого из них. Напишите программу, которая бы создала файл произвольного доступа "hardware.dat" на сотню пустых записей, позволяла бы вводить данные по каждому изделию, давала бы вам возможность получать список всех изделий, удалять записи по изделиям, которых у вас уже нет, и позволяла бы обновлять любую информацию в файле. Ключом должен быть идентификационный номер изделия. Используйте следующую информацию для начала работы с вашим файлом:

| Номер записи | Название инструмента | Количество | Стоимость |
|--------------|----------------------|------------|-----------|
| 3            | Шлифовальный станок  | 7          | 57.98     |
| 17           | Молоток              | 76         | 11.99     |
| 24           | Механический лобзик  | 21         | 11.00     |
| 39           | Газонокосилка        | 3          | 79.50     |
| 56           | Электропила          | 18         | 99.99     |
| 68           | Отвертка             | 106        | 6.99      |
| 77           | Кузнецкий молот      | 11         | 21.50     |
| 83           | Гаечный ключ         | 34         | 7.50      |

**14.13.** Модифицируйте программу генерации номера телефона, которую вы написали в главе 4, с тем, чтобы она записывала свои выходные данные в файл. Это позволит вам читать этот файл, когда вам это будет удобно. Если у вас имеется компьютерный словарь, модифицируйте вашу программу для поиска в словаре до тысячи слов, состоящих каждое из семи символов. Некоторые интересные комбинации из 7 букв, созданные этой программой, могут состоять из двух и более слов. Например, номер телефона 8432677 соответствует "THEBOSS". Модифицируйте вашу программу для использо-

зования компьютерного словаря при проверке каждого возможного слова из 7 букв с тем, чтобы посмотреть, не состоит ли оно из слова из 1 буквы, следом за которым идет слово из 6 букв; затем проверьте, не состоит ли оно из слова из 2 символов, следом за которым идет слово из 5 букв и т.д.

- 14.14.** Напишите программу, которая использует операцию `sizeof` для определения числа байтов различных типов данных в используемой вами системе компьютера. Напишите результат в файл "datasize.dat", чтобы позднее его можно было распечатать. Формат результата в файле должен быть следующий:

| Тип данных                      | Размер |
|---------------------------------|--------|
| <code>char</code>               | 1      |
| <code>unsigned char</code>      | 1      |
| <code>short int</code>          | 2      |
| <code>unsigned short int</code> | 2      |
| <code>int</code>                | 4      |
| <code>unsigned int</code>       | 4      |
| <code>long int</code>           | 4      |
| <code>unsigned long int</code>  | 4      |
| <code>float</code>              | 4      |
| <code>double</code>             | 8      |
| <code>long double</code>        | 16     |

Замечание: размер встроенных типов данных на вашем компьютере может отличаться от приведенного выше списка.

## г л а в а

---

15

# Структуры данных



## Ц е л и

- Научиться создавать связные структуры данных, используя указатели, классы с самоадресацией и рекурсию.
- Научиться динамически выделять и освобождать память для создания и уничтожения объектов со структурами данных.
- Научиться создавать и манипулировать динамически-ми структурами данных, такими, как связные списки, очереди, стеки и бинарные деревья.
- Понять работу различных приложений со связными структурами данных.
- Понять, каким образом следует создавать многократно используемые структуры данных с помощью шаблонов классов, наследования и композиции.

## План

- 15.1. Введение**
- 15.2. Классы с самоадресацией**
- 15.3. Динамическое выделение памяти**
- 15.4. Связные списки**
- 15.5. Стеки**
- 15.6. Очереди**
- 15.7. Деревья**

*Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения • Специальный раздел: создание вашего собственного компилятора*

### 15.1. Введение

Мы изучили такие *структуры данных* фиксированного размера, как одномерные индексированные массивы, двумерные индексированные массивы и структуры `struct`. В этой главе вводятся *динамические структуры данных*, которые нарастают и сокращаются в процессе выполнения программы. *Связные списки* являются наборами элементов данных, «выстроеными в линейку»; операции вставки элементов данных и их удаления могут осуществляться в любом месте связного списка. *Стеки* играют существенную роль в компиляторах и операционных системах; операции вставки и удаления проводятся в конце стека, то есть в его *вершине*. *Очереди* представляют собой линейку элементов, ждущих, чтобы их обслужили; операция вставки проводится в конце очереди (происходит обращение к последнему элементу, называемому также *хвостом* очереди), а операция удаления из очереди проводится в ее начале (называемом также *головой* очереди). *Бинарные деревья* способствуют высокоскоростному поиску и быстрой сортировке данных, эффективному удалению дубликатов элементов данных, отображению системы каталогов файлов а также компиляции выражений на машинный язык. У этих структур данных имеется много других любопытных приложений.

Мы обсудим каждый из этих основных типов структур данных и их применение в программах, создающих и манипулирующих этими структурами данных. Для создания и упаковки этих структур данных, для их многократного использования и сопровождения мы будем применять классы, шаблоны классов, механизм наследования и композицию.

Примеры этой главы являются практическими программами, которые вы можете использовать в курсах повышенной сложности и в промышленных приложениях. Эти программы являются особенно мощными благодаря использованию указателей. Упражнения данной главы включают обширный набор полезных приложений.

Мы воодушевляем вас на смелую попытку создать сложный проект, описанный в специальном разделе этой главы «Создание вашего собственного компилятора». Вы постоянно пользуетесь компилятором для трансляции ваших программ, написанных на языке C++, в машинный код и выполнения этих программ на вашем компьютере. А в этом проекте вы действительно построите ваш собственный компилятор. Он будет читать файл из операторов, записанных на элементарном, но тем не менее мощном языке высокого уровня, аналогичном ранним версиям популярного языка БЕЙСИК. Ваш компилятор будет транслировать эти операторы в файл инструкций машинного языка Простотрон (ЯМП), который вы изучали в специальном разделе главе 5 «Создание вашего собственного компьютера». Ваша программа, моделирующая Простотрон, будет выполнять программу на ЯМП, созданную вашим же компилятором! Реализация этого проекта, интенсивно использующего объектно-ориентированный подход, даст вам замечательную возможность наилучшего использования всего того, что вы изучили в этой области. В этом специальном разделе мы пройдемся по спецификациям языка высокого уровня и рассмотрим алгоритмы, необходимые для преобразования каждого типа оператора на языке высокого уровня в язык машинных команд. Если вы отважитесь принять брошенный вам вызов, то вы можете внести множество различных новшеств как в компилятор, так и в программу, моделирующую Простотрон и используемую в упражнениях.

## 15.2. Классы с самоадресацией

*Классы с самоадресацией* содержат элемент указатель, который указывает на объект того же типа класса. Например, описание

```
class Node {
public:
 Node(int);
 void setData(int);
 int getData() const;
 void setNextPtr(const Node *);
 const Node *getNextPtr() const;
private:
 int data;
 Node *nextPtr;
};
```

определяет тип класса **Node**. Тип **Node** имеет два закрытых данных-элементов: целый элемент **data** и элемент указатель **nextPtr**. Элемент указатель **nextPtr** указывает на объект типа **Node** — объект того же типа, который здесь

объявляется; такие классы мы будем называть «классами с самоадресацией». Элемент `nextPtr` используется как *связь*, то есть `nextPtr` может быть использован для «связи» объекта типа `Node` с другим объектом того же типа. У класса типа `Node` имеется также пять функций-элементов: конструктор, который принимает данные целого типа для инициализации элемента `data`, функция-элемент для установки значения `data`, функция-элемент `getData`, возвращающая значение `data`, функция-элемент `setNextPtr` для установки значения элемента указателя `nextPtr`, а также функция-элемент `getNextPtr`, которая возвращает значение элемента указателя `nextPtr`.

Объекты классов с самоадресацией могут связываться друг с другом, формируя такие полезные структуры данных, как списки, очереди, стеки и деревья. На рис. 15.1 показаны два объекта класса с самоадресацией, связанных вместе для создания списка. Заметим, что обратный слэш (\), изображающий нулевой указатель (0), помещен в элемент связи второго объекта только чтобы было ясно, что эта связь не указывает на какой-либо другой объект. Этот символ имеет чисто иллюстративное назначение и никак не связан с символом обратного слэша ('\'), используемым в языке C++. Нулевой указатель обычно отражает конец структуры данных так же, как нулевой символ ('\0') отражает конец строки.

#### Типичная ошибка программирования 15.1

Указатель связи в последнем узле не устанавливается на нуль (0).



Рис. 15.1. Два связанных объекта класса с самоадресацией

### 15.3. Динамическое выделение памяти

Создание и поддержание динамических структур данных требует *динамического распределения памяти*: возможности в процессе выполнения программы увеличивать область памяти для хранения новых узлов и освобождать ресурсы памяти, в которых уже нет необходимости. Пределы динамического выделения памяти ограничены только объемом доступной физической памяти или доступной виртуальной памяти в системах с виртуальной памятью. Впрочем, часто эти пределы намного меньше из-за того, что свободная память делиится при совместном доступе к ней многих пользователей.

Операции `new` и `delete` являются основными при динамическом распределении памяти. Операция `new` принимает в качестве аргумента тип динамически размещаемого объекта и возвращает указатель на объект этого типа. Например, оператор

```
Node *newPtr = new Node(10);
```

выделяет область памяти размером `sizeof(Node)` байтов и сохраняет указатель на эту область памяти в элементе `newPtr`. Если такого объема свободной памяти не имеется, то операция `new` возвращает нулевой указатель. Число 10 представляет собой число размещаемых объектов данных.

Операция **delete** освобождает область памяти, выделенную операцией **new**, то есть эта область памяти возвращается системе и она может быть опять распределена в дальнейшем. Для освобождения памяти, динамически выделенной предыдущим оператором, используется оператор

```
delete newPtr;
```

Заметим, что сам указатель **newPtr** не удаляется, исчезает только область памяти, на которую указывал **newPtr**.

В следующих разделах рассматриваются списки, стеки, очереди и деревья. Эти структуры данных создаются и поддерживаются с помощью динамического распределения памяти и использования классов с самоадресацией.

### **Замечание по мобильности 15.1**

Размер объектов класса не обязательно равен сумме своих данных-элементов. Это происходит из-за различных машинно-зависимых требований по выравниванию границ областей памяти (см. главу 16) и по другим причинам.

### **Типичная ошибка программирования 15.2**

Предположение о том, что размер объекта класса является простой суммой размеров его данных-элементов.

### **Хороший стиль программирования 15.1**

Используя операцию **new** проверьте, не вернула ли она нулевой указатель. Выполните соответствующую обработку ошибки, если операция **new** не выделила область памяти.

### **Типичная ошибка программирования 15.3**

Не осуществляется возвращение динамически выделенной памяти, когда эта память уже более не требуется. Это может явиться причиной преждевременного переполнения памяти. Иногда это явление называют «утечкой памяти».

### **Хороший стиль программирования 15.2**

Когда память, которая динамически выделена операцией **new**, более не требуется, используйте операцию **delete** для немедленного освобождения памяти.

### **Типичная ошибка программирования 15.4**

Освобождение операцией **delete** памяти, которая не была выделена динамически операцией **new**.

### **Типичная ошибка программирования 15.5**

Ссылка на область памяти, которая была освобождена.

## 15.4. Связные списки

Связный список является линейным набором объектов классов с самоадресацией, называемых *узлами*, связанных при помощи указателей *связи* и поэтому определяемых термином «связный список». Доступ к связному списку осуществляется через указатель на первый узел списка. Последующие узлы доступны через указатели *связи*, хранящиеся в каждом узле. В соответствии с соглашением указатель *связи* в последнем узле списка устанавливается на нуль для того, чтобы отметить конец списка. Данные в связном списке хранятся динамически, то есть каждый узел создается по мере необходимости. Узлы могут содержать данные любого типа, включая объекты других классов. Стеки и очереди также являются линейными структурами данных и, как мы увидим, являются частными случаями связного списка. Деревья являются нелинейной структурой данных.

Списки данных могут храниться и в массивах, но связные списки представляют некоторые преимущества. Применение связных списков является уместным в том случае, когда число элементов, которые должны быть представлены в структуре данных, заранее не известно. Размер «обычного» массива в C++ не может быть изменен, поскольку он зафиксирован на этапе компиляции. «Обычные» массивы могут переполняться. Связные списки могут переполняться только в том случае, если у системы нет достаточного места для удовлетворения запросов по динамическому выделению памяти.

### Совет по повышению эффективности 15.1

Можно объявить размер массива таким, чтобы он мог вместить большее число элементов, чем ожидается. Но это может привести к избыточному расходу памяти. Связные списки могут обеспечивать более рациональное использование памяти. Связные списки позволяют программе настраиваться во время счета.

Связные списки могут сортироваться просто путем вставки каждого нового элемента в соответствующую позицию в списке. При этом существующие элементы списка не надо перемещать.

### Совет по повышению эффективности 15.2

Операции вставки и удаления в отсортированном массиве могут быть продолжительными по времени, так как все элементы, следующие за вставляемым или удаляемым, должны быть соответствующим образом сдвинуты.

### Совет по повышению эффективности 15.3

Элементы массива хранятся в памяти непрерывно (по соседству друг с другом). Это предоставляет возможность мгновенного доступа к любому элементу массива, поскольку адрес любого элемента может быть вычислен непосредственно путем определения его позиции по отношению к началу массива. Связные списки не представляют возможности для такого мгновенного доступа к своим элементам.

Узлы связных списков обычно физически не хранятся в памяти по соседству друг с другом. Однако, логически они как бы хранятся подряд. На рис. 15.2 представлен связный список с несколькими узлами.

### Совет по повышению эффективности 15.4

Использование вместо массивов динамического распределения памяти для структур данных, которые могут увеличиваться или уменьшаться во время счета, способствует экономическому использованию ресурсов памяти. Однако имейте ввиду, что указатели занимают некоторое место в памяти и что динамическое распределение приводит к нерациональному использованию памяти при обращениях к функциям.

Программа, приведенная на рис. 15.3 (выходные данные, полученные в результате выполнения этой программы, показаны на рис. 15.4), использует шаблон класса `List` (см. главу 12 «Шаблоны») для манипуляций со списком данных целого типа и списком данных с плавающей запятой. Программа драйвер (`driver.cpp`) дает возможность использовать 5 режимов работы: 1) вставка значения в начало списка (функция `insertAtFront`); 2) вставка значения в конец списка (`insertAtBack`); 3) удаление значения из начала списка (функция `removeFromFront`); 4) удаление значения из конца списка (функция `removeFromBack`); 5) завершение обработки списка. Детальное обсуждение этой программы будет дано ниже. В упражнении 15.20 предлагается реализовать рекурсивную функцию, которая печатает связный список в обратной последовательности; а в упражнении 15.21 предлагается реализовать рекурсивную функцию поиска указанного элемента данных в связном списке.

Программа, приведенная на рис. 15.3, состоит из двух шаблонов класса: `ListNode` и `List`. Каждый объект, инкапсулированный в шаблон класса `List`, является связным списком объектов класса `ListNode`. Шаблон класса `ListNode` состоит из закрытых элементов `data` и `nextPtr`. Элемент `data` класса `ListNode` хранит значение типа `NODETYPE`, этот тип параметра передается шаблону класса. Элемент указатель `nextPtr` класса `ListNode` хранит указатель на следующий объект класса `ListNode` в связном списке.

Шаблон класса `List` состоит из закрытых элементов `firstPtr` (указатель на первый объект списка класса `ListNode`) и `lastPtr` (указатель на последний объект списка класса `ListNode`). Конструктор по умолчанию задает обоим указателям начальные значения 0. Деструктор обеспечивает уничтожение всех объектов класса `ListNode`, входящих в объект класса `List`, при уничтожении самого этого объекта класса `List`. Основными функциями-элементами шаблона класса `List` являются `insertAtFront`, `insertAtBack`, `removeFromFront` и `removeFromBack`. Функция `isEmpty` называется *предикатной функцией* или *предикатом* — она, во всех случаях не изменяет список, а только устанавливает, не является ли он пустым (т.е. не является ли указатель на первый узел списка нулевым). Если список пуст, то возвращается 1; в противном случае, возвращается 0. Функция `print` выводит на экран содержимое списка.

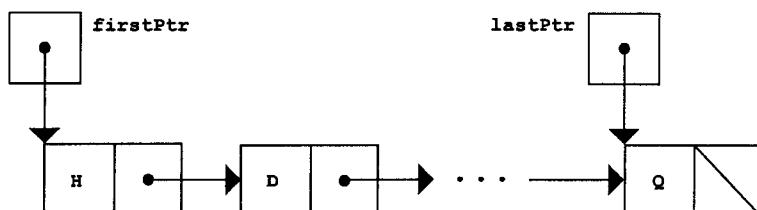


Рис. 15.2. Графическое представление списка

```

// LISTND.H: Определение шаблона ListNode
#ifndef LISTND_H
#define LISTND_H

template<class NODETYPE>
class ListNode {
 friend class List<NODETYPE>; // объявление List
 // дружественной функцией

public:
 ListNode(const NODETYPE &); // конструктор
 NODETYPE getData() const; // возвращает данные в узле
private:
 NODETYPE data; // данные
 ListNode *nextPtr; // следующий узел в списке
};

// Конструктор
template<class NODETYPE>
ListNode<NODETYPE>::ListNode(const NODETYPE &info)
{
 data = info;
 nextPtr = 0;
}

// Возвращает копию данных в узле
template<class NODETYPE>
NODETYPE ListNode<NODETYPE>::getData() const { return data; }
#endif

// LIST.H: Определение шаблона класса List
#ifndef LIST_H
#define LIST_H
#include <iostream.h>
#include <assert.h>
#include "timel.h"

template<class NODETYPE>
class List {
public:
 List(); // конструктор
 ~List(); // деструктор
 void insertAtFront(const NODETYPE &);
 void insertAtBack(const NODETYPE &);
 int removeFromFront(NODETYPE &);
 int removeFromBack(NODETYPE &);
 int isEmpty() const;
 void print() const;
private:
 ListNode<NODETYPE> *firstPtrg // указатель
 // на первый узел
 ListNode<NODETYPE> *lastPtr; // указатель на последний
 // узел
 ListNode<NODETYPE> *getNewNode(const NODETYPE &); // утилита
};


```

Рис. 15.3. Управление связным списком (часть 1 из 6)

```
// Конструктор с умолчанием
template<class NODETYPE>
List<NODETYPE>::List() { firstPtr = lastPtr = 0; }

// Деструктор
template<class NODETYPE> List<NODETYPE>::~List()
{
 if (! isEmpty()) { // Список не пустой
 cout << "Удаление узлов ... " << endl;

 ListNode<NODETYPE> *currentPtr = firstPtr, *tempPtr;

 while (currentPtr != 0) { // удаление
 tempPtr = currentPtr;
 cout << tempPtr->data << endl;
 currentPtr = currentPtr->nextPtr;
 delete tempPtr;
 }
 }

 cout << "Все узлы удалены" << endl << endl;
}

// Вставка узла в начало списка
template<class NODETYPE>
void List<NODETYPE>::insertAtFront(const NODETYPE &value)
{
 ListNode<NODETYPE> *newPtr = getNewNode(value);

 if (isEmpty()) // Список пустой
 firstPtr = lastPtr = newPtr;
 else { // Список не пустой
 newPtr->nextPtr = firstPtr;
 firstPtr = newPtr;
 }
}

// Вставка узла в конец списка
template<class NODETYPE>
void List<NODETYPE>::insertAtBack(const NODETYPE &value)
{
 ListNode<NODETYPE> *newPtr = getNewNode(value);

 if (isEmpty()) // Список пустой
 firstPtr = lastPtr = newPtr;
 else { // Список не пустой
 lastPtr->nextPtr = newPtr;
 lastPtr = newPtr;
 }
}
```

Рис. 15.3. Управление связным списком (часть 2 из 6)

```

// Удаление узла из начала списка
template<class NODETYPE>
int List<NODETYPE>::removeFromFront (NODETYPE &value)
{
 if (isEmpty()) // список пустой
 return 0; // неудачное удаление
 else (
 ListNode<NODETYPE> *tempPtr = firstPtr;

 if (firstPtr == lastPtr)
 firstPtr = lastPtr = 0;
 else
 firstPtr = firstPtr->nextPtr;

 value = tempPtr->data; // перемещение данных
 delete tempPtr;
 return 1; // успешное удаление
 }
}

// Удаление узла из конца списка
template<class NODETYPE>
int List<NODETYPE>::removeFromBack (NODETYPE &value)
{
 if (isEmpty())
 return 0; // неудачное удаление
 else {
 ListNode<NODETYPE> *tempPtr = lastPtr;

 if (firstPtr == lastPtr)
 firstPtr = lastPtr = 0;
 else {
 ListNode<NODETYPE> *currentPtr = firstPtr;

 while (currentPtr->nextPtr != lastPtr)
 currentPtr = currentPtr->nextPtr;

 lastPtr = currentPtr;
 currentPtr->nextPtr = 0;
 }

 value = tempPtr->data;
 delete tempPtr;
 return 1; // удачное удаление
 }
}

// Является ли список пустым ?
template<class NODETYPE>
int List<NODETYPE>::isEmpty() const { return firstPtr == 0; }

```

Рис. 15.3. Управление связным списком (часть 3 из 6)

```
// Возвращение указателя на ближайший узел
template<class NODETYPE>
ListNode<NODETYPE> *List<NODETYPE>::getNewNode(const NODETYPE &value)
{
 ListNode<NODETYPE> *ptr = new ListNode<NODETYPE>(value);

 assert(ptr != 0);
 return ptr;
}

// Отображение на экране содержимого List
template<class NODETYPE>
void List<NODETYPE>::print() const
{
 if (isEmpty())
 cout <<"Список пуст" <<endl <<endl;
 return;
}

ListNode<NODETYPE> *currentPtr = firstPtr;
cout << "Список состоит из: ";

while (currentPtr != 0) {
 cout << currentPtr->data << ' ';
 currentPtr = currentPtr->nextPtr;
}

cout << endl << endl;
}

#endif

// DRIVER.CPP
// Проверка класса List

#include <iostream.h>
#include "boss.h"

void testIntegerList(); // проверка списка целых чисел
void testFloatList(); // проверка списка чисел
 // с плавающей запятой
void instructions(); // инструкции для пользователя

main()
{
 testIntegerList(); // проверка списка целых чисел
 testFloatList(); // проверка списка чисел
 // с плавающей запятой

 return 0;
}
```

Рис. 15.3. Управление связанным списком (часть 4 из 6)

```
// Функция проверки списка целых чисел

void testIntegerList()
{
 cout << "Проверка списка целых чисел" << endl;

 List<int> integerList;

 instructions();

 int choice, value;

 do {
 cout << "? ";
 cin >> choice;
 switch (choice) {

 case 1:
 cout << "Введите целое: ";
 cin >> value;
 integerList.insertAtFront(value);
 integerList.print();
 break;

 case 2:
 cout << "Введите целое: ";
 cin >> value;
 integerList.insertAtBack(value);
 integerList.print();
 break;

 case 3:
 if (integerList.removeFromFront(value))
 cout << value << " удаляется из списка" << endl;
 integerList.print();
 break;

 case 4:
 if (integerList.removeFromBack(value))
 cout << value << " удаляется из списка" << endl;
 integerList.print();
 break;
 }
 } while (choice != 5);

 cout << "Конец проверки списка целых чисел" << endl;
}
```

Рис. 15.3. Управление связным списком (часть 5 из 6)

```
// Функция проверки списка чисел с плавающей запятой

void testFloatList()
{
 cout << "Проверка списка чисел с плавающей запятой" << endl;
 List<float> floatList;
 instructions();
 int choice;
 float value;

 do{
 cout << "? ";
 cin >> choice;

 switch (choice) {
 case 1:
 cout << "Введите число с плавающей запятой: ";
 cin >> value;
 floatList.insertAtFront(value);
 floatList.print ();
 break;
 case 2:
 cout << "Введите число с плавающей запятой: ";
 cin >> value;
 floatList.insertAtBack(value);
 floatList.print ();
 break;
 case 3:
 if (floatList.removeFromFront(value))
 cout << value << " удаляется из списка" << endl;
 floatList.print ();
 break;
 case 4:
 if (floatList.removeFromBack(value))
 cout << value << " удаляется из списка" << endl;
 floatList.print ();
 break;
 }
 } while (choice != 5);

 cout << "Конец проверки списка чисел с плавающей запятой"
 << endl;
}

void instructions()
{
 cout << "Выберите:" << endl
 << "1 - вставить в начало списка" << endl
 << "2 - вставить в конец списка" << endl
 << "3 - удалить из начала списка " << endl
 << "4 - удалить из конца списка " << endl
 << "5 - завершить обработку списка" << endl;
}
```

Рис. 15.3. Управление связанным списком (часть 6 из 6)

Проверка списка целых чисел

Выберите:

- 1 - вставить в начало списка
  - 2 - вставить в конец списка
  - 3 - удалить из начала списка
  - 4 - удалить из конца списка
  - 5 - завершить обработку списка << endl;
- ? 1

Ведите целое: 1

Список состоит из: 1

? 1

Ведите целое: 2

Список состоит из: 2 1

? 2

Ведите целое: 3

Список состоит из: 2 1 3

? 2

Ведите целое: 4

Список состоит из: 2 1 3 4

? 3

2 удаляется из списка

Список состоит из: 1 3 4

? 3

1 удаляется из списка

Список состоит из: 3 4

? 4

4 удаляется из списка

Список состоит из: 3

? 4

3 удаляется из списка

Список пуст

? 5

Конец проверки списка целых чисел

Список уничтожен

Проверка списка чисел с плавающей запятой

Выберите:

- 1 - вставить в начало списка
- 2 - вставить в конец списка
- 3 - удалить из начала списка
- 4 - удалить из конца списка
- 5 - завершить обработку списка

Рис. 15.4. Пример вывода для программы, приведенной на рис. 15.3 (часть 1 из 2)

```
? 1
Введите число с плавающей запятой: 1.1
Список состоит из: 1.1

? 1
Введите число с плавающей запятой: 2.2
Список состоит из: 2.2 1.1

? 2
Введите число с плавающей запятой: 3.3
Список состоит из: 2.2 1.1 3.3

? 2
Введите число с плавающей запятой: 4.4
Список состоит из: 2.2 1.1 3.3 4.4

? 3
2.2 удаляется из списка
Список состоит из: 1.1 3.3 4.4

? 3
1.1 удаляется из списка
Список состоит из: 3.3 4.4

? 4
4.4 удаляется из списка
Список состоит из: 3.3

? 4
3.3 удаляется из списка
Список пуст

? 5
Конец проверки списка чисел с плавающей запятой
```

Список уничтожен

Рис. 15.4. Пример вывода для программы, приведенной на рис. 15.3 (часть 2 из 2)

### Хороший стиль программирования 15.3

Присваивайте нулевое значение ( 0 ) указателю элементу нового узла. Указатели должны быть инициализированы перед тем, как они используются.

Теперь мы обсудим детально функции-элементы класса **List**. Функция **insertAtFront** (см. рис. 15.5) вставляет новый узел в начало списка. Функция осуществляет это за несколько шагов:

- 1) Вызывается функция **getNewNode**, которой передается **value** — константная ссылка на значение вставляемого узла.
- 2) Функция **getNewNode** использует операцию **new** для создания нового узла списка и возвращает указатель на этот узел в **Ptr**. Если этот указатель имеет ненулевое значение, то функция **getNewNode** возвращает указатель на вновь помещенный узел и передает его в **newPtr** в функции **insertAtFront**.

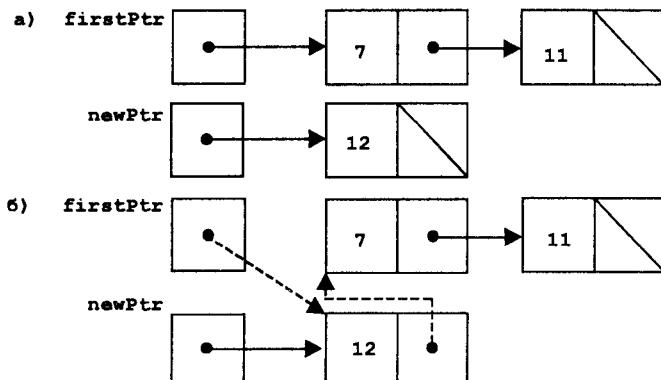


Рис. 15.5. Выполнение функции `insertAtFront`

- 3) Если список пустой, тогда и указатель `firstPtr`, и указатель `lastPtr` устанавливаются равными `newPtr`.
- 4) Если список не является пустым, тогда узел, на который указывает `newPtr`, как бы вставляется в начало списка путем копирования указателя `firstPtr` в указатель `newPtr->nextPtr`, так что этот новый узел будет теперь указывать в качестве следующего тот, который ранее являлся первым в списке; кроме того, в результате копирования `newPtr` в указатель `firstPtr` новое значение `firstPtr` будет указывать на новый узел как на первый в списке.

Рис. 15.5 иллюстрирует работу функции `insertAtFront`. Рис. 15.5 а) показывает состояние списка и нового узла до выполнения `insertAtFront`. Пунктирные стрелки на рис. 15.5 б) иллюстрируют шаги 2 и 3 при выполнении функции `insertAtFront`, которые позволяют узлу, содержащему число 12, стать новым первым узлом списка.

Функция `insertAtBack` (см. рис. 15.6) помещает новый узел в конец списка. Функция выполняется в несколько шагов:

- 1) Вызывается функция `getNewNode`, которой передается `value` — константная ссылка на значение вставляемого узла.
- 2) Функция `getNewNode` использует операцию `new` для создания нового узла списка и возвращает указатель на этот узел в `Ptr`. Если этот указатель имеет ненулевое значение, то функция `getNewNode` возвращает указатель на вновь помещенный узел и передает его в `newPtr` в функции `insertAtBack`.
- 3) Если список пустой, тогда и указатель `firstPtr`, и указатель `lastPtr` устанавливаются равными `newPtr`.
- 4) Если список не является пустым, тогда узел, на который указывает `newPtr`, как бы вставляется в конец списка путем копирования указателя `newPtr` в указатель `lastPtr->nextPtr`, так что теперь на этот новый узел будет указывать как на следующий тот узел, который ранее был последним в списке; кроме того, в результате копирования `newPtr` в указатель `lastPtr` новое значение `lastPtr` будет указывать на новый узел как на последний в списке.

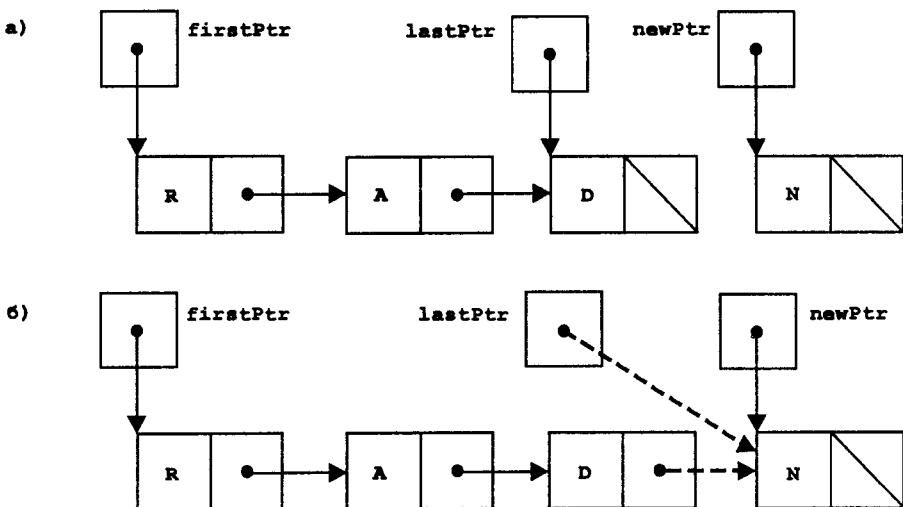
Рис. 15.6. Графическая иллюстрация выполнения функции `insertAtBack`

Рис. 15.6 иллюстрирует выполнение функции `insertAtBack`. Рис. 15.6 а) показывает состояние списка и нового узла до выполнения функции `insertAtBack`. Пунктирные стрелки на рис. 15.6 б) иллюстрируют действия функции `insertAtBack`, которая позволяет добавить новый узел в конец уже заполненного списка.

Функция `removeFromFront` (см. рис. 15.7) удаляет первый узел списка и копирует значение этого узла в параметр ссылки. Функция возвращает 0, если предпринята попытка удаления узла из пустого списка, и возвращает 1, если удаление было успешным. Функция выполняется в несколько шагов:

- 1) Создается экземпляр указателя `tempPtr` как копия указателя `firstPtr`. Указатель `tempPtr` будет использован для освобождения области памяти удаляемого узла.
- 2) Если `firstPtr` равен `lastPtr`, то есть до начала удаления в списке имелся только один элемент, то указателям `firstPtr` и `lastPtr` присваиваются нулевые значения (список становится пустым).
- 3) Если в списке до начала удаления имелось более одного узла, то указатель `lastPtr` остается неизменным, а указатель `firstPtr` устанавливается равным `firstPtr->nextPtr`, то есть теперь `firstPtr` указывает на второй узел первоначального списка (теперь этот узел становится первым).
- 4) После того, как все эти манипуляции с указателями завершены, в параметр ссылку `value` копируется элемент `data` удаляемого узла.
- 5) Операция `delete` освобождает область памяти, выделенную для узла, на который указывает `tempPtr`.
- 6) Возвращается 1, что указывает на успешное удаление.

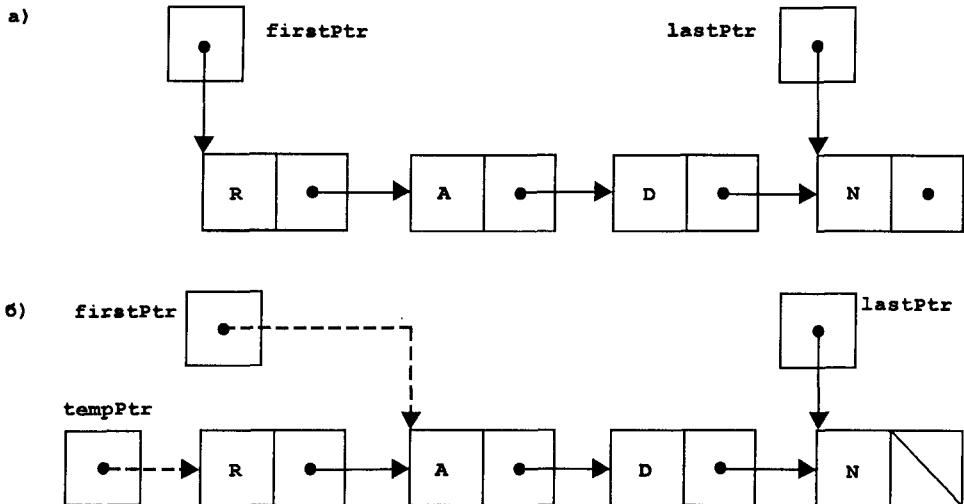


Рис. 15.7. Графическое представление выполнения функции `removeFromFront`

Рис. 15.7 иллюстрирует выполнение функции `removeFromFront`.

Рис. 15.7 а) показывает состояние списка до удаления. На рис. 15.7 б) показаны реализованные операции с указателями.

Функция `removeFromBack` (см. рис. 15.8) удаляет последний узел списка и копирует значение узла в параметр ссылки. Эта функция возвращает 0, если была осуществлена попытка удаления из пустого списка, и возвращает 1, если удаление было успешным. Функция выполняется в несколько шагов:

- 1) Создается экземпляр указателя `tempPtr` как копии указателя `lastPtr`. Указатель `tempPtr` будет использован для освобождения области памяти удаляемого узла.
- 2) Если `firstPtr` равен `lastPtr`, то есть до начала удаления в списке имелся только один элемент, то указателям `firstPtr` и `lastPtr` присваиваются нулевые значения (список становится пустым).
- 3) Если в списке до начала удаления имелось более одного узла, то создается экземпляр указателя `currentPtr` как копия указателя `firstPtr`.
- 4) Теперь осуществляется просмотр списка с помощью указателя `currentPtr`, пока он не укажет на узел перед последним узлом. Это делается в цикле `while`, который осуществляет замену указателя `currentPtr` на `currentPtr->nextPtr` до тех пор, пока `currentPtr->nextPtr` не станет равным `lastPtr`.
- 5) Осуществляется копирование `currentPtr` в `lastPtr` для того, чтобы убрать последний узел из списка.
- 6) Указатель `currentPtr->nextPtr`, т.е. указатель связи в новом последнем узле списка, устанавливается равным нулю.
- 7) После того, как завершены все манипуляции с указателями, в параметр ссылку `value` копируется элемент `data` удаляемого узла.

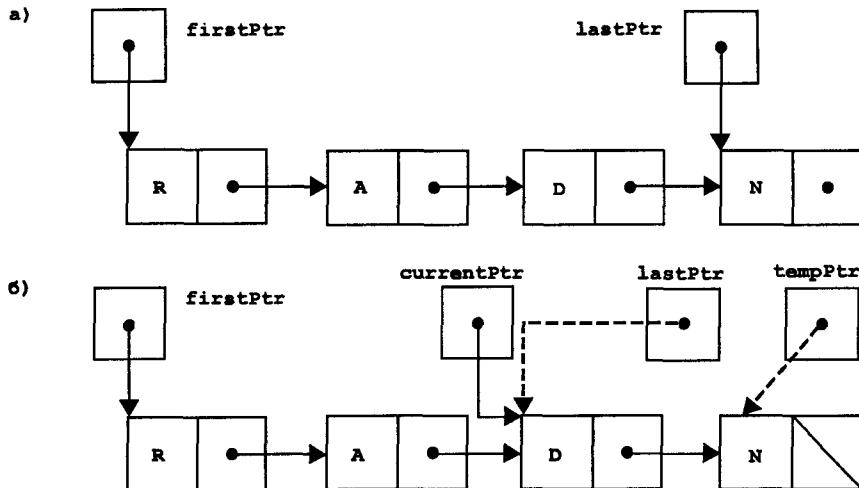


Рис. 15.8. Графическое представление выполнения функции `removeFromBack`

8) Операция `delete` освобождает область памяти, выделенную для узла, на который указывает `tempPtr`.

9) Возвращается 1, что указывает на успешное удаление.

Рис. 15.8 иллюстрирует выполнение функции `removeFromFront`. На рис. 15.8 а) показано состояние списка до операции удаления. На рис. 15.8 б) показаны реализованные операции с указателями.

Функция `print` сначала определяет, не является ли список пустым. Если он пустой, то функция `print` печатает сообщение «Список пуст» и завершает работу. В противном случае, она печатает данные, входящие в список. Функция создает экземпляр указателя `currentPtr` как копию указателя `firstPtr` и затем печатает сообщение «Список состоит из: ». Пока указатель `currentPtr` не является нулевым, в цикле печатается `currentPtr->data` и указателю `currentPtr` присваивается значение `currentPtr->nextPtr`. Заметим, что, если указатель связи в последнем узле списка не является нулевым, то алгоритм печати будет ошибочно выводить на печать данные после конца списка. Алгоритм печати является одинаковым для связанных списков, стеков и очередей.

## 15.5. Стеки

**Стек** является частным случаем связного списка, в котором новые узлы могут добавляться в стек и удаляться (выталкиваться) из него только на его вершине. По этой причине, стек является структурой данных типа *последним вошел — первым вышел* (*last-in, first-out* — LIFO). Элемент связи в последнем узле стека устанавливается на нуль для того, чтобы показать дно стека.

### Типичная ошибка программирования 15.6

Не осуществляется установка указателя связи на нуль в узле, являющемся дном стека.

Основными функциями-элементами, используемыми для манипуляций со стеком, являются `push` и `pop`. Функция `push` добавляет новый узел на вершину стека. Функция `pop` удаляет (выталкивает) узел из вершины стека, сохраняет удаляемое значение в переменной ссылке, которая передается вызывающей программе, и возвращает 1, если действие функции `pop` было успешным, или 0 в противном случае.

У стеков имеется много интересных приложений. Например, когда происходит вызов функций, вызываемая функция должна знать, как вернуться в вызывающую функцию; в этом случае адрес возврата помещается в стек. Если происходит ряд обращений к функциям, то последовательность адресов возврата помещается в стек по принципу «последним вошел — первым вышел» для того, чтобы каждая функция могла вернуться в свою вызывающую функцию. Стеки поддерживают как рекурсивные вызовы функций, так и обычные нерекурсивные.

У стеков имеется область памяти, выделяемая для автоматических переменных при каждом обращении к функции. Когда функция возвращается в свою вызывающую функцию, эта область для автоматических переменных указанной функции удаляется из стека и эти переменные более не известны программе.

Стеки используются компиляторами в процессе вычисления выражений и для генерации машинного кода. В упражнениях приводится несколько применений стеков, включая их использование для создания полноценно работающего компилятора.

Воспользуемся тесной связью между списками и стеками для реализации класса стеков путем повторного использования класса списков. Применим две разновидности повторного использования. Сначала, мы реализуем класс стеков при помощи механизма скрытого наследования класса списков. Затем реализуем аналогично действующий класс стеков при помощи композиции путем включения класса списков в качестве скрытого элемента класса стеков. Конечно, все структуры данных в этой главе, включая эти два класса стеков, реализуются как шаблоны (см. главу 12 «Шаблоны»), чтобы иметь возможность повторно использовать их в дальнейшем.

Программа на рис. 15.9 (вывод данных этой программы показан на рис. 15.10) создает шаблон класса стеков главным образом посредством скрытого наследования шаблона класса списков. Потребуем, чтобы у стека имелись функции-элементы `push`, `pop`, `isEmpty` и `printStack`. Заметим, что они, по существу, являются соответственно функциями шаблона класса списков `insertAtFront`, `removeFromFront`, `isEmpty` и `print`. Конечно, шаблон класса списков включает и другие функции-элементы (например, `insertAtBack` и `removeFromBack`), которые нам не хотелось бы делать доступными через открытый интерфейс класса стеков. Так что когда мы указываем, что шаблон класса стеков должен наследовать шаблон класса списков, то задаем скрытое наследование. Это приводит к тому, что все функции-элементы шаблона класса списков в шаблоне класса стеков становятся скрытыми. Когда мы реализуем функции-элементы стека, они должны вызывать соответствующие функции-элементы класса списков: `push` должна вызывать `insertAtFront`, `pop` — `removeFromFront`, `isEmpty` — `isEmpty`, а функция-элемент `printStack` должна вызывать `print`.

```
// STACK.H
// Определение шаблона класса Stack,
// производного от класса List
#ifndef STACK_H
#define STACK_H

#include "list.h"

template<class STACKTYPE>
class Stack : private List<STACKTYPE> {
public:
 void push(const STACKTYPE &d) { insertAtFront(d); }
 int pop(STACKTYPE &d) { return removeFromFront(d); }
 const isStackEmpty() const { return isEmpty(); }
 void printStack() const { print(); }
};

#endif
// STACKDRV.CPP
// Драйвер проверки шаблона класса Stack
#include <iostream.h>
#include "stack.h"

main ()
{
 Stack<int> intStack;
 int popInteger;
 cout << "обработка стека с целыми числами" << endl;

 for (int i = 0; i < 4; i++) {
 intStack.push(i);
 intStack.printStack();
 }

 while (!intStack.isStackEmpty()) {
 intStack.pop(popInteger);
 cout << popInteger << " выталкивается из стека" << endl;
 intStack.printStack();
 }

 Stack<float> floatStack;
 float val = 1.1, popFloat;

 cout << "обработка стека с числами с плавающей точкой" << endl;

 for (i = 0; i < 4; i++) {
 floatStack.push(val);
 floatStack.printStack();
 val += 1.1;
 }

 while (!floatStack.isStackEmpty()) {
 floatStack.pop(popFloat);
 cout << popFloat << " выталкивается из стека" << endl;
 floatStack.printStack();
 }

 return 0;
}
```

Рис. 15.9. Простая программа стека

```
Обработка стека с целыми числами

Список состоит из: 0

Список состоит из: 1 0

Список состоит из: 2 1 0

Список состоит из: 3 2 1 0

3 выталкивается из стека
Список состоит из: 2 1 0

2 выталкивается из стека
Список состоит из: 1 0

1 выталкивается из стека
Список состоит из: 0

0 выталкивается из стека
Список пуст

обработка стека с числами с плавающей точкой

Список состоит из: 1.1

Список состоит из: 2.2 1.1

Список состоит из: 3.3 2.2 1.1

Список состоит из: 4.4 3.3 2.2 1.1

4.4 выталкивается из стека
Список состоит из: 3.3 2.2 1.1

3.3 выталкивается из стека
Список состоит из: 2.2 1.1

2.2 выталкивается из стека
Список состоит из: 1.1

1.1 выталкивается из стека
Список пустой

Все узлы удалены
Все узлы удалены
```

Рис. 15.10. Пример вывода программы, приведенной на рис. 15.9

Шаблон класса стеков используется в функции `main` для реализации стека `intStack` типа `Stack<int>` с целыми числами. Целые значения от 0 до 3 помещаются в стек `intStack` и затем выталкиваются из него. Шаблон класса стеков используется также для реализации стека со значениями с плавающей запятой `floatStack` типа `Stack<float>`. Значения с плавающей точкой 1.1, 2.2, 3.3 и 4.4 помещаются в стек `floatStack` и затем выталкиваются из него.

Другим способом реализации шаблона класса стеков является повторное использование шаблона класса списков посредством создания композиции. Программа, приведенная на рис. 15.11, использует заголовочные файлы *list.h* и *listnd.h*. Она также использует ту же самую программу драйвер, что и предыдущая программа для стека, за исключением нового заголовочного файла *stack\_c.h*, который заменяет заголовочный файл *stack.h*. Вывод является тем же самым. Определение шаблона класса стеков теперь включает объект-элемент *s* типа *List<STACKTYPE>*.

```
// STACK_C.H
// Определение класса Stack как композиции объектов класса List
#ifndef STACK_C
#define STACK_C

#include "list.h"

template<class STACKTYPE>
class Stack {
public:
 // Конструктор отсутствует; инициализируется конструктором
 // класса List
 void push(const STACKTYPE &);
 int pop(STACKTYPE &);
 int isEmpty() const;
 void printStack() const;
 .
private:
 List<STACKTYPE> s;
};

// Значение помещается в стек
template<class STACKTYPE>
void Stack<STACKTYPE>::push(const STACKTYPE &value)
{ s.insertAtFront(value); }

// Значение выталкивается из стека
template<class STACKTYPE>
int Stack<STACKTYPE>::pop(STACKTYPE &value)
{ return s.removeFromFront(value); }

// Является ли стек пустым ?
template<class STACKTYPE>
int Stack<STACKTYPE>::isEmpty() const { return s.isEmpty(); }

// Отображение содержимого стека
template<class STACKTYPE>
void Stack<STACKTYPE>::printStack() const { s.print(); }

#endif
```

Рис. 15.11. Пример программы стека, использующей композицию

## 15.6. Очереди

Другой стандартной структурой данных является очередь. Очередь аналогична очереди людей в супермаркете: первый клиент в ней обслуживается первым, а другие клиенты занимают очередь с конца и ожидают, когда их обслужат. Узлы очереди удаляются только из головы очереди, а помещаются в очередь только в ее *хвосте*. По этой причине, очередь — это структура данных типа «*первым вошел — первым вышел*» (*first-in, first-out* — FIFO). Операции поместить в очередь и удалить из нее известны как *enqueue* и *dequeue* соответственно.

У очередей имеется множество применений в вычислительных системах. У большинства компьютеров имеется только один процессор; поэтому в один и тот же момент времени может быть обслужен только один пользователь. Запросы других пользователей помещаются в очередь. Каждый запрос постепенно продвигается в очереди вперед по мере того, как происходит обслуживание пользователей. Запрос в начале очереди является очередным кандидатом на обслуживание.

Очереди также применяются для поддержания процесса буферизации потоков данных, выводимых на печать. В многопользовательской среде может быть только один принтер. У многих пользователей в процессе выполнения программ могут создаться данные, которые следует распечатать. Эти выходные данные записываются в буферный файл на диске (подобно наматывающейся на катушку нити), где они ожидают в очереди, пока принтер не станет для них доступным.

Информационные пакеты также ожидают своей очереди в компьютерных сетях. Пакет может поступить на узел сети в любой момент времени, а затем он должен быть отправлен к следующему узлу сети по направлению своего конечного пункта назначения. Узел маршрутизации (т.е. узел, хранящий информацию о маршруте) направляет в каждый момент времени один пакет; поэтому пакеты помещаются в очередь до тех пор, пока программа маршрутизации не обработает их.

Файл-сервер в компьютерной сети обрабатывает запросы многих клиентов сети. Серверы имеют ограниченную пропускную способность обслуживания запросов клиентов. Когда такая пропускная способность превышена, запрос клиента ожидает своей очереди.

```
// QUEUE.H
// Определение шаблона класса Queue (производного от класса List)
#ifndef QUEUE_H
#define QUEUE_H
#include "list.h"

template<class QUEUETYPE>
class Queue: private List<QUEUETYPE> {
public:
 void enqueue(const QUEUETYPE &d) { insertAtBack(d); }
 int dequeue(QUEUETYPE &d) { return removeFromFront(d); }
 int isQueueEmpty() const { return isEmpty(); }
 void printQueue() const { print(); }
};

#endif
```

Рис. 15.12. Обработка очереди (часть 1 из 2)

```
// QUEUEDRV.CPP
// Драйвер для проверки шаблона класса Queue
#include <iostream.h>
#include "queue.h"

main ()
{
 Queue<int> intqueue;
 int dequeueInteger;
 cout << "обработка очереди целых чисел" << endl;

 for (int i = 0; i < 4; i++) {
 intqueue.enqueue(i);
 intqueue.printqueue();
 }

 while (! intqueue.isEmpty()) {
 intqueue.dequeue(dequeueInteger);
 cout << dequeueInteger << " удаляется из очереди" << endl;
 intQueue.printqueue();
 }

 Queue<float> floatQueue;
 float val = 1.1, dequeueFloat;

 cout << "обработка очереди чисел с плавающей точкой" << endl;

 for (i = 0; i < 4; i++) {
 floatqueue.enqueue(val);
 floatQueue.printQueue();
 val += 1.1;
 }

 while (!floatQueue.isEmpty()) {
 floatqueue.dequeue(dequeueFloat);
 cout << dequeueFloat << " удаляется из очереди" << endl;
 floatQueue.printQueue();
 }

 return 0;
}
```

Рис. 15.12. Обработка очереди (часть 2 из 2)

### Типичная ошибка программирования 15.7

Отсутствие установки указателя связи в последнем узле очереди в нуль (**0**).

Программа, приведенная на рис. 15.12 (вывод данных показан на рис. 15.13), создает шаблон класса очередей используя скрытое наследование шаблона класса списков. Потребуем, чтобы очередь имела функции-элементы `enqueue`, `dequeue`, `isEmpty` и `printQueue`. Заметим, что эти функции-элементы, по существу, являются функциями-элементами шаблона класса списков `insertAtBack`, `removeFromFront`, `isEmpty` и `print` соответственно.

обработка очереди с целыми числами

Список состоит из: 0

Список состоит из: 0 1

Список состоит из: 0 1 2

Список состоит из: 0 1 2 3

0 удаляется из очереди

Список состоит из: 1 2 3

1 удаляется из очереди

Список состоит из: 2 3

2 удаляется из очереди

Список состоит из: 3

3 удаляется из очереди

Список пуст

обработка очереди со значениями с плавающей точкой

Список состоит из: 1.1

Список состоит из: 1.1 2.2

Список состоит из: 1.1 2.2 3.3

Список состоит из: 1. 2.2 3.3 4.4

1.1 удаляется из очереди

Список состоит из: 2.2 3.3 4.4

2.2 удаляется из очереди

Список состоит из: 3.3 4.4

3.3 удаляется из очереди

Список состоит из: 4.4

4.4 удаляется из очереди

Список пуст

Все узлы удалены

Все узлы удалены

Рис. 15.13. Пример вывода программы, приведенной на рис. 15.12

Конечно, шаблон класса списков включает и другие функции-элементы (`insertAtFront` и `removeFromBack`), которые не желательно было бы делать доступными для класса очередей через открытый интерфейс. Так что когда мы указываем, что шаблон класса очередей должен наследовать шаблон класса списков, то задаем скрытое наследование. Это приводит к тому, что все функции-элементы шаблона класса списков становятся скрытыми в шаблоне класса

очередей. Когда мы реализуем функции-элементы очереди, они должны вызывать соответствующие функции-элементы класса списков: `enqueue` должна вызывать `insertAtBack`, `dequeue` — `removeFromFront`, `isQueueEmpty` — `isEmpty`, а функция-элемент `printQueue` должна вызывать `print`.

Шаблон класса очередей используется в функции `main` для реализации очереди с данными целого типа `intQueue` типа `Queue<int>`. Целые числа от 0 до 3 помещаются в очередь к `intQueue`, а затем удаляются из нее по принципу «первым вошел — последним вышел». Затем шаблон класса очередь используется для реализации очереди со значениями с плавающей запятой `floatQueue` типа `Queue<float>`. Значения с плавающей запятой 1.1, 2.2, 3.3 и 4.4 помещаются в очередь `floatQueue`, а затем удаляются из нее по тому же принципу «первым вошел — последним вышел».

## 15.7. Деревья

Связные списки, стеки и очереди являются линейными структурами данных. Дерево является нелинейной двумерной структурой данных с особыми свойствами. Узлы дерева могут содержать два и более указателей связи. В этом разделе рассмотрены бинарные деревья (см. рис. 15.14) — деревья, узлы которых содержат по два указателя (ни один из них, один или оба могут быть нулевыми). Корневой узел является первым узлом дерева. Каждый указатель связи в корневом узле ссылается на дочерний узел или узел-потомок. Левый узел-потомок является первым узлом в левом поддереве, а правый узел-потомок является первым узлом в правом поддереве. Узлы-потомки, порожденные одним каким-либо узлом, называются родственными узлами (или узлами-братьями). Узел без узлов-потомков называется листом дерева или концевым узлом. Специалисты по вычислительной технике обычно рисуют деревья сверху-вниз, начиная с корневого узла, то есть противоположно тому, как растут деревья в природе.

### Типичная ошибка программирования 15.8

Отсутствие установки указателей связи в нуль (**0**) в концевых узлах дерева.

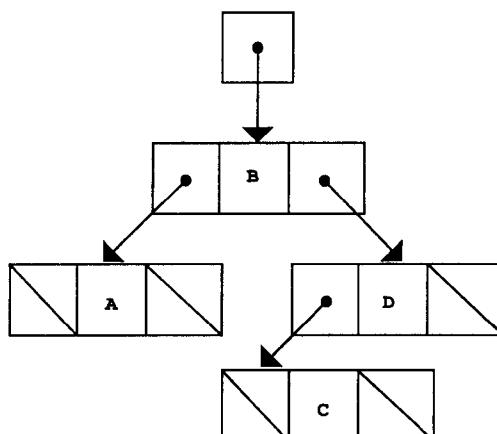


Рис. 15.14. Графическое представление дерева

В этом разделе создается особое бинарное дерево, которое называется *деревом двоичного поиска (дихотомии)*. Дерево двоичного поиска (без дублирующих значений узлов) имеет характерную особенность, заключающуюся в том, что значение в любом узле левого поддерева меньше значения в его родительском узле, а значение в любом узле его правого поддерева больше значения в его родительском узле. На рис. 15.15 показано дерево двоичного поиска с 12-тью значениями. Заметим, что вид дерева двоичного поиска, соответствующий набору меняющихся данных, зависит от последовательности, в которой значения помещаются в дерево.

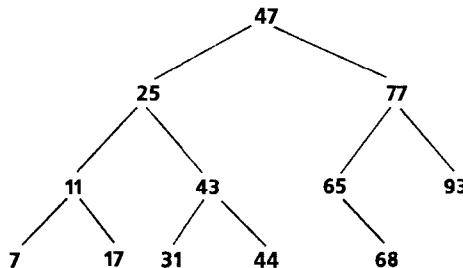


Рис. 15.15. Дерево двоичного поиска

Программа, приведенная на рис. 15.16 (ее вывод представлен на рис. 15.17), создает дерево двоичного поиска и обходит его тремя способами: *последовательным (симметричным) обходом (inorder)*, *обходом в ширину (прямым обходом) (preorder)* и *обратным обходом (postorder)*. Программа генерирует 10 случайных чисел и помещает каждое из них в дерево, за исключением дублирующих значений, которые отбрасываются.

Давайте рассмотрим программу бинарного дерева, приведенную на рис. 15.16. Функция `main` начинает свою работу с создания экземпляра дерева двоичного поиска с данными целого типа `intTree` типа `Tree<int>`. Программа запрашивает 10 целых чисел, каждое из которых помещается в бинарное дерево путем обращения к `insertNode`. Затем программа выполняет последовательный (симметричный) обход, обход в ширину и обратный обход (эти варианты будут объяснены ниже) дерева `intTree`. После этого программа создает экземпляр дерева для данных с плавающей запятой `floatTree` типа `Tree<float>`. Программа запрашивает 10 данных с плавающей запятой, каждое из которых помещается в бинарное дерево путем обращения к `insertNode`. После этого программа выполняет последовательный (симметричный) обход, обход в ширину и обратный обход дерева `floatTree`.

Рассмотрим теперь описания шаблона класса и функций-элементов. Начнем с шаблона класса `TreeNode`, который объявляет в качестве дружественного шаблон класса `Tree`. Класс `TreeNode` имеет в качестве закрытых данных значения в узлах `data` и указатели `leftPtr` (указатель на узлы левого поддерева) и `rightPtr` (указатель на узлы правого поддерева). Конструктор задает в качестве начального значения `data` значение, которое он получает в качестве аргумента, и устанавливает указатели `leftPtr` и `rightPtr` в нуль (таким образом, узел инициализируется как концевой). Функция-элемент `getData` возвращает значение `data`.

Класс `Tree` имеет в качестве закрытых данных `rootPtr` — указатель на корневой узел дерева. У класса имеются открытые функции-элементы `in-`

`sertNode` (которая помещает узел в дерево), `preOrderTraversal`, `inOrderTraversal` и `postOrderTraversal`, каждая из которых обходит дерево заданным способом. Каждая из этих функций-элементов вызывает свою собственную отдельную рекурсивную функцию-утилиту для выполнения соответствующих операций над внутренней структурой дерева. Конструктор класса `Tree` задает указателю `rootPtr` нулевое значение, чтобы показать, что дерево в исходном состоянии является пустым.

Функция-утилита `insertNodeHelper` класса `Tree` рекурсивно помещает узел в дерево. Узел может быть помещен в дерево двоичного поиска только в качестве концевого узла. Если дерево является пустым, то создается новый экземпляр класса `TreeNode`, который инициализируется, и узел помещается в это дерево.

Если дерево не является пустым, то программа сравнивает вставляемое в дерево значение со значением `data` в корневом узле. Если вставляемое значение меньше значения в корневом узле, то программа рекурсивно вызывает утилиту `insertNodeHelper` для того, чтобы вставить значение в левое поддерево. Если вставляемое значение больше значения в корневом узле, то программа рекурсивно обращается к `insertNode` для того, чтобы поместить значение в правое поддерево. Если вставляемое значение равно значению данных в корневом узле, то программа печатает сообщение «дубликат» и возвращается, не вставив этот дубликат в дерево.

```
// TREE NODE.H: Определение класса TreeNode
#ifndef TREENODE_H
#define TREENODE_H

template<class NODETYPE> class TreeNode {
 friend class Tree<NODETYPE>;
public:
 TreeNode(const NODETYPE &); // конструктор
 NODETYPE getData() const; // возвращение данных
private:
 TreeNode *leftPtr; // указатель на левое поддерево
 NODETYPE data;
 TreeNode *rightPtr; // указатель на правое
 // поддерево
};

// Конструктор
template<class NODETYPE>
TreeNode<NODETYPE>::TreeNode(const NODETYPE &d)
{
 data = d;
 leftPtr = rightPtr = 0;
}

// Возвращение копии значений данных
template<class NODETYPE>
NODETYPE TreeNode<NODETYPE>::getData() const { return data; }
#endif
```

Рис. 15.16. Создание бинарного дерева и его обход (часть 1 из 4)

```
// TREE.H: Определение шаблона класса Tree

#ifndef TREE_H
#define TREE_H

#include <iostream.h>
#include <assert.h>
#include "treenode.h"

template<class NODETYPE>
class Tree {
public:
 Tree();
 void insertNode(const NODETYPE &);
 void preOrderTraversal() const;
 void inOrderTraversal() const;
 void postOrderTraversal() const;
private:
 TreeNode<NODETYPE> *rootPtr;

 // функции утилиты
 void insertNodeHelper(TreeNode<NODETYPE> **, const NODETYPE &);
 void preOrderHelper(TreeNode<NODETYPE> *) const;
 void inOrderHelper(TreeNode<NODETYPE> *) const;
 void postOrderHelper(TreeNode<NODETYPE> *) const;
};

template<class NODETYPE>
Tree<NODETYPE>::Tree() { rootPtr = 0; }

template<class NODETYPE>
void Tree<NODETYPE>::insertNode(const NODETYPE Value)
{ insertNodeHelper(&rootPtr, value); }

// Эта функция принимает указатель на указатель,
// так что указатель может быть модифицирован.
template<class NODETYPE>
void Tree<NODETYPE>::insertNodeHelper (TreeNode<NODETYPE> **ptr,
 const NODETYPE &value)
{
 if (*ptr == 0) { // пустое дерево
 *ptr = new TreeNode<NODETYPE>(value);
 assert(*ptr != 0);
 }
 else { // дерево не пустое
 if (value < (*ptr)->data)
 insertNodeHelper(&((*ptr)->leftPtr), value);
 else

 if (value > (*ptr)->data)
 insertNodeHelper(&((*ptr)->rightPtr), value);
 else
 cout << value << " дубль" << endl;
 }
}
```

**Рис. 15.16.** Создание бинарного дерева и его обход (часть 2 из 4)

```
template<class NODETYPE>
void Tree<NODETYPE>::preOrderTraversal() const
{ preOrderHelper(rootPtr); }

template<class NODETYPE>
void Tree<NODETYPE>::preOrderHelper(TreeNode<NODETYPE> *ptr) const
{
 if (ptr != 0) {
 cout << ptr->data << ' ';
 preOrderHelper(ptr->leftPtr);
 preOrderHelper(ptr->rightPtr);
 }
}

template<class NODETYPE>
void Tree<NODETYPE>::inOrderTraversal() const
{ inOrderHelper(rootPtr); }

template<class NODETYPE>
void Tree<NODETYPE>::inOrderHelper(TreeNode<NODETYPE> *ptr) const
{
 if (ptr != 0) {
 inOrderHelper(ptr->leftPtr);
 cout << ptr->data << ' ';
 inOrderHelper(ptr->rightPtr);
 }
}

template<class NODETYPE>
void Tree<NODETYPE>::postOrderTraversal() const
{ postOrderHelper(rootPtr); }

template<class NODETYPE>
void Tree<NODETYPE>::postOrderHelper(TreeNode<NODETYPE> *ptr) const
{
 if (ptr != 0) {
 postOrderHelper(ptr->leftPtr);
 postOrderHelper(ptr->rightPtr);
 cout << ptr->data << ' ';
 }
}

#endif

// Драйвер проверки класса Tree
#include <iostream.h>
#include <iomanip.h>
#include "tree.h"

main()
{
 Tree<int> intTree;
 int intValue;
```

Рис. 15.16. Создание бинарного дерева и его обход (часть 3 из 4)

```

cout << "Введите 10 целых чисел: " << endl;
for (int i = 0; i < 10; i++) {
 cin >> intVal;
 intTree.insertNode(intVal);
}

cout << endl << "Обход в ширину" << endl;
intTree.preOrderTraversal();

cout << endl << "Последовательный обход" << endl;
intTree.inOrderTraversal();
cout << endl << "Обратный обход" << endl;
intTree.postOrderTraversal();

Tree<float> floatTree;
float floatVal;

cout << endl << endl
 << "Введите 10 чисел с плавающей точкой: "
 << endl << setiosflags(ios::fixed | ios::showpoint)
 << setprecision(1);
for (i = 0; i < 10; i++) {
 cin >> floatVal;
 floatTree.insertNode(floatVal);
}

cout << endl << "Обход в ширину" << endl;
floatTree.preOrderTraversal();

cout << endl << "Последовательный обход" << endl;
floatTree.inOrderTraversal();

cout << endl << "Обратный обход" << endl;
floatTree.postOrderTraversal()

return 0;
}

```

**Рис. 15.16.** Создание бинарного дерева и его обход (часть 4 из 4)

Каждая из функций-элементов `inOrderTraversal`, `preOrderTraversal` и `postOrderTraversal` обходит дерево (см. рис. 15.18) и печатает значения в узлах.

Функция-элемент `inOrderTraversal` осуществляет последовательный обход дерева, выполняя следующие шаги:

- 1) Обход левого поддерева с помощью `inOrderTraversal`.
- 2) Обработка значения в узле (т.е. печать значения в этом узле).
- 3) Обход правого поддерева с помощью `inOrderTraversal`.

Значение в узле не обрабатывается до тех пор, пока не будут обработаны значения в его левом поддереве. Функция-элемент `inOrderTraversal` для дерева, показанного на рис. 15.18, выдает следующие значения:

6 13 17 27 33 42 48

**Введите 10 целых чисел:**  
 50 25 75 12 33 67 88 6 13 68

**Обход в ширину:**  
 50 25 12 6 13 33 75 67 68 88

**Последовательный обход:**  
 6 12 13 25 33 50 67 68 75 88

**Обратный обход:**  
 6 13 12 33 25 68 67 88 75 50

**Введите 10 чисел с плавающей точкой:**  
 39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

**Обход в ширину:**  
 39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5

**Последовательный обход:**  
 1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5

**Обратный обход:**  
 1.1 .4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2

**Рис. 15.17.** Пример вывода данных программы, приведенной на рис. 15.16

Заметим, что функция-элемент **inOrderTraversal** печатает узловые значения в порядке их возрастания. Процесс создания дерева двоичного поиска в действительности сортирует данные — поэтому этот процесс называется *сорттировкой бинарного дерева*.

Функция-элемент **preOrderTraversal** осуществляет обход дерева в ширину (прямой обход), выполняя следующие шаги:

- 1) Обработка значения в узле (печать).
- 2) Обход левого поддерева с помощью **preOrderTraversal**.
- 3) Обход правого поддерева с помощью **preOrderTraversal**.

Значение в каждом узле обрабатывается, когда обход дерева проходит через этот узел. После того, как значение в данном узле обработано, обрабатываются значения в левом поддереве, а потом — в правом. Функция-элемент **preOrderTraversal** для дерева, показанного на рис. 15.18, выдает следующие значения:

27 13 6 17 42 33 48

Функция-элемент **postOrderTraversal** осуществляет обратный обход дерева, выполняя следующие шаги:

- 1) Обход левого поддерева с помощью **postOrderTraversal**.
- 2) Обход правого поддерева с помощью **postOrderTraversal**.
- 3) Обработка значения в узле.

Значение в каждом узле не печатается до тех пор, пока не распечатаны значения в его узлах потомках. Функция-элемент **postOrderTraversal** для дерева, показанного на рис. 15.18, выдает следующие значения:

6 17 13 33 48 42 27

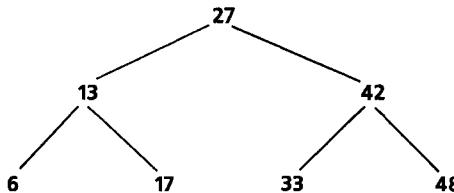


Рис. 15.18. Дерево двоичного поиска

Дерево двоичного поиска упрощает процесс *удаления дубликатов*. Если дерево создано, попытка поместить в него дублирующее значение будет распознана при каждом решении «следовать налево» или «следовать направо». Значение дубликата в этот момент может просто отбрасываться.

Процесс поиска значения в бинарном дереве, соответствующему некоторому ключевому значению, также является быстрым. Если дерево является плотно упакованным, то каждый уровень содержит примерно вдвое больше элементов, чем предыдущий уровень. Таким образом, дерево двоичного поиска с  $n$  — элементами имеет максимум  $\log_2 n$  уровней и, таким образом, для того, чтобы найти соответствующее значение или убедиться, что его нет, требуется произвести максимум  $\log_2 n$  сравнений. Это означает, например, что при проведении поиска в плотно упакованном дереве, содержащем 1000 элементов, требуется провести не более 10 сравнений, поскольку  $2^{10} > 1000$ . При поиске в плотно упакованном дереве, содержащем 1 000 000 элементов, требуется провести не более 20 сравнений, поскольку  $2^{20} > 1 000 000$ .

В упражнениях представлены алгоритмы для проведения некоторых операций с бинарным деревом, а именно: удаление элемента из дерева, печать дерева в двумерном формате и послойный обход дерева. Послойный обход дерева подразумевает его просмотр по уровням, начиная с корневого узла. На каждом уровне дерева обход производится слева направо. Другие упражнения с бинарным деревом позволяют ему хранить дублирующие значения, вставлять в дерево строки и определять количество уровней в дереве.

## Резюме

- Классы с самоадресацией содержат элементы, называемые указателями связи, которые, в свою очередь, указывают на объекты того же класса.
- Классы с самоадресацией позволяют объединять объекты класса в стеки, очереди, списки и деревья.
- Динамическое выделение памяти резервирует во время выполнения программы блок байтов в памяти для хранения объекта.
- Связный список является линейной набором объектов класса с самоадресацией.
- Связный список является динамической структурой данных — длина списка может увеличиваться или уменьшаться по мере необходимости.
- Размер связных списков может наращиваться до тех пор, пока не исчерпана память.
- Связные списки обеспечивают простые механизмы вставки и удаления данных путем манипулирования указателями.

- Стеки и очереди являются частным случаем связных списков.
- Новые узлы стека добавляются в него или удаляются только на вершине стека. По этой причине стек относится к структурам данных типа «последним вошел — первым вышел» («last-in, first-out» — LIFO).
- Элемент указатель связи последнего узла списка устанавливается в нуль для того, чтобы обозначить дно стека.
- Двумя основными операциями, которые используются для управления стеком, являются `push` и `pop`. Операция `push` создает новый узел и помещает его в вершину стека. Операция `pop` удаляет (выталкивает) узел из вершины стека, освобождает выделенную для него память и возвращает удаленное значение.
- В структуре данных очередь узлы удаляются из головы (начала) очереди и добавляются в ее хвост (в конец). По этой причине очередь относится к структурам данных типа «первым вошел — первым вышел» («first-in, first-out» — FIFO). Операции добавления в очередь и удаления из нее известны как `enqueue` и `dequeue`.
- Деревья являются двумерными структурами данных, требующими два или более указателей связи на узел.
- Бинарные деревья имеют два указателя связи в каждом узле.
- Корневой узел является первым узлом дерева.
- Каждый из указателей в корневом узле обращается к узлу-потомку (дочернему узлу). Левый узел-потомок является первым узлом левого поддерева, а правый узел-потомок является первым узлом правого поддерева. Порожденные одним каким-либо узлом узлы-потомки называются родственными узлами или узлами-братьями. Любой узел дерева, который не имеет каких-либо узлов-потомков, называется листом дерева или концевым узлом.
- Дерево двоичного поиска характеризуется тем, что значение в левом узле-потомке меньше значения в его родительском узле, а значение в правом узле-потомке больше или равно значению в его родительском узле. Если нет дублирующих значений, то значение в правом узле-потомке просто больше значения в его родительском узле.
- При последовательном (симметричном) обходе двоичного дерева проводится последовательный поиск в левом поддереве, обрабатывается значение в корневом узле, а затем проводится последовательный поиск в правом поддереве. Значение в узле не обрабатывается до тех пор, пока не закончится обработка значений в его левом поддереве.
- При обходе в ширину (прямом обходе) обрабатывается значение в корневом узле, проводится поиск в ширину в левом дереве, а затем проводится аналогичный поиск в правом поддереве. Значение в каждом узле обрабатывается, как только этот узел встречается при выполнении обхода.
- При обратном обходе проводится обратный обход в левом поддереве, затем проводится обратный обход в правом поддереве, а затем обрабатывается значение в корневом узле. Значение в каждом узле не обрабатывается до тех пор, пока не будут обработаны значения в обоих его поддеревьях.

## Терминология

|                                                |                                       |
|------------------------------------------------|---------------------------------------|
| <b>dequeue</b>                                 | нулевой указатель                     |
| <b>enqueue</b>                                 | обратный обход                        |
| <b>FIFO</b> («первым вошел — первым вышел»)    | обход                                 |
| <b>LIFO</b> («последним вошел — первым вышел») | обход в ширину (прямой обход)         |
| <b>pop</b>                                     | очередь                               |
| <b>push</b>                                    | поддерево                             |
| <b>sizeof</b>                                  | посещение узла                        |
| <b>бинарное дерево</b>                         | последовательный (симметричный) обход |
| <b>вершина</b>                                 | послойный обход бинарного дерева      |
| <b>вставка узла</b>                            | правое поддерево                      |
| <b>голова очереди</b>                          | правый узел-потомок                   |
| <b>двойное разименование</b>                   | родительский узел                     |
| <b>дерево</b>                                  | родственные узлы                      |
| <b>дерево двоичного поиска</b>                 | связный список                        |
| <b>динамические структуры данных</b>           | сортировка бинарного дерева           |
| <b>динамическое выделение памяти</b>           | стек                                  |
| <b>дочерний узел</b>                           | структура с самоадресацией            |
| <b>концевой узел</b>                           | удаление дубликатов                   |
| <b>корневой узел</b>                           | удаление узла                         |
| <b>левое поддерево</b>                         | узел                                  |
| <b>левый узел-потомок</b>                      | узлы-потомки                          |
| <b>линейная структура данных</b>               | указатель на указатель                |
| <b>лист дерева</b>                             | функция предикат                      |
| <b>нелинейная структура данных</b>             | хвост очереди                         |

## Типичные ошибки программирования

- 15.1. Указатель связи в последнем узле не устанавливается на нуль (0).
- 15.2. Предположение о том, что размер объекта класса является простой суммой размеров его данных-элементов.
- 15.3. Не осуществляется возвращение динамически выделенной памяти, когда эта память уже более не требуется. Это может явиться причиной преждевременного переполнения памяти. Иногда это явление называют «утечкой памяти».
- 15.4. Освобождение операцией **delete** памяти, которая не была выделена динамически операцией **new**.
- 15.5. Ссылка на область памяти, которая была освобождена.
- 15.6. Не осуществляется установка указателя связи на нуль в узле, являющемся дном стека.
- 15.7. Отсутствие установки указателя связи в последнем узле очереди в нуль (0).
- 15.8. Отсутствие установки указателей связи в нуль (0) в концевых узлах дерева.

## Хороший стиль программирования

- 15.1. Используя операцию `new` проверьте, не вернула ли она нулевой указатель. Выполните соответствующую обработку ошибки, если операция `new` не выделила область памяти.
- 15.2. Когда память, которая динамически выделена операцией `new`, более не требуется, используйте операцию `delete` для немедленного освобождения памяти.
- 15.3. Присваивайте нулевое значение (0) указателю элементу нового узла. Указатели должны быть инициализированы перед тем, как они используются.

## Советы по повышению эффективности

- 15.1. Можно объявить размер массива таким, чтобы он мог вместить большее число элементов, чем ожидается. Но это может привести к избыточному расходу памяти. Связные списки могут обеспечивать более рациональное использование памяти. Связные списки позволяют программе настраиваться во время счета.
- 15.2. Операции вставки и удаления в отсортированном массиве могут быть продолжительными по времени, так как все элементы, следующие за вставляемым или удаляемым, должны быть соответствующим образом сдвинуты.
- 15.3. Элементы массива хранятся в памяти непрерывно (по соседству друг с другом). Это предоставляет возможность мгновенного доступа к любому элементу массива, поскольку адрес любого элемента может быть вычислен непосредственно путем определения его позиции по отношению к началу массива. Связные списки не предоставляют возможности для такого мгновенного доступа к своим элементам.
- 15.4. Использование вместо массивов динамического распределения памяти для структур данных, которые могут увеличиваться или уменьшаться во время счета, способствует экономному использованию ресурсов памяти. Однако имейте ввиду, что указатели занимают некоторое место в памяти и что динамическое распределение приводит к нерациональному использованию памяти при обращениях к функциям.

## Замечания по мобильности

- 15.1. Размер объектов класса не обязательно равен сумме своих данных-элементов. Это происходит из-за различных машинно-зависимых требований по выравниванию границ областей памяти (см. главу 16) и по другим причинам.

## Упражнения для самопроверки

- 15.1. Заполнить пробелы в следующих утверждениях:

- a) Класс с \_\_\_\_\_ используется для создания динамических структур данных, которые могут увеличиваться или уменьшаться во время выполнения программы.
- b) Операция \_\_\_\_\_ используется для динамического выделения памяти. Эта операция возвращает указатель на выделенную память.
- c) \_\_\_\_\_ является частным случаем связного списка, в котором узлы могут вставляться и удаляться только из его вершины. Эта структура данных возвращает значения в узлах по принципу «последним вошел — первым вышел» (LIFO).
- d) Функция, которая не меняет связный список, а просто его просматривает и определяет, не является ли он пустым, называется \_\_\_\_\_.
- e) Очередь относится к структуре данных типа \_\_\_\_\_.
- f) Указатель на следующий узел в связном списке называется \_\_\_\_\_.
- g) Для освобождения динамически выделенной памяти используется операция \_\_\_\_\_.
- h) \_\_\_\_\_ является частным случаем связного списка, в котором узлы могут быть помещены только в конец списка, а удалены только из его начала.
- i) \_\_\_\_\_ является нелинейной двумерной структурой данных, в которой узлы имеют по два или более указателей связи.
- j) Стек относится к структурам данных типа \_\_\_\_\_, поскольку последний помещенный в него узел удаляется первым.
- k) Узлы \_\_\_\_\_ дерева включают два элемента связи.
- l) Первый узел дерева называется \_\_\_\_\_ узлом.
- m) Каждая связи в дереве указывает на \_\_\_\_\_ или \_\_\_\_\_ данного узла.
- n) Узел дерева, который не имеет узлов-потомков, называется \_\_\_\_\_ или \_\_\_\_\_ узлом.
- o) Четыре алгоритма обхода дерева двоичного поиска, которые были упомянуты в главе, называются соответственно \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ и \_\_\_\_\_.

15.2. Какие существуют различия между связным списком и стеком?

15.3. Какие существуют различия между стеком и очередью?

15.4. Возможно наиболее подходящим названием для этой главы было бы «Повторно используемые структуры данных». Поясните, каким образом каждый из нижеперечисленных объектов и понятий способствуют повторному использованию структур данных:

- a) классы
- b) шаблоны классов
- c) наследование
- d) скрытое наследование
- e) композиция.

15.5. Осуществите вручную последовательный обход, обход в ширину и обратный обход дерева двоичного поиска, приведенного на рис. 15.19.

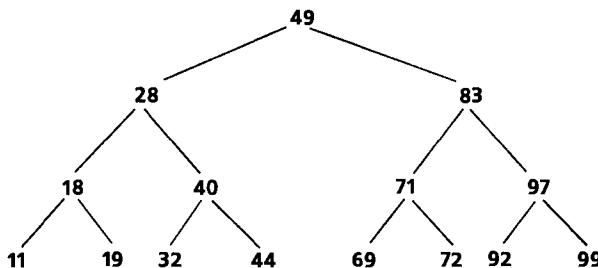


Рис. 15.19. Дерево двоичного поиска с 12-тью узлами

### Ответы на упражнения для самопроверки

- 15.1. a) с самоадресацией; b) new; c) стек; d) предикат, предикатная функция; e) «первым вошел — первым вышел» (FIFO); f) указателем связи; g) delete; h) очередь; i) дерево; j) «последним вошел — первым вышел» (LIFO); k) бинарного; l) корневым; m) узел-потомок, поддерево; n) листом дерева, концевым; o) последовательный (симметричный) обход, обход в ширину (прямой обход), обратный обход, послойный обход.
- 15.2. В связном списке можно помещать узел в любое место списка и удалять узлы из любого места списка. Узлы в стеке могут быть помещены только в его вершину, а удалены только из вершины стека.
- 15.3. Очередь имеет указатели на головной элемент и на хвост очереди, так что узлы могут быть помещены только в конец (хвост) очереди, а удалены только из ее начала (головы). Стек имеет единственный указатель на вершину стека и только в ней выполняются операция вставки и удаления узла.
- 15.4. a) Классы позволяют нам создавать столько объектов структур данных определенного типа (т.е. классов), сколько нам необходимо.  
 b) Шаблоны класса позволяют нам создавать связные классы, каждый из которых базируется на различных типах параметров, и мы можем создавать столько объектов каждого шаблона класса, сколько нам необходимо.  
 c) Наследование позволяет нам повторно использовать код базового класса для производного класса, причем структуры данных производного класса также являются структурами данных базового класса (с открытым интерфейсом).  
 d) Скрытое наследование позволяет повторно использовать часть кода базового класса для создания структуры данных производного класса; поскольку это наследование является скрытым, то открытые функции-элементы базового класса становятся закрытыми в производном классе. Это, в свою очередь, препятствует произвольному доступу объектов производного класса к таким функциям-элементам базового класса, которые в производном классе не применяются.

е) Композиция позволяет повторно использовать код путем создания объектов классов структур данных, являющихся элементами составного класса; если мы создадим закрытый объект-элемент составного класса, то в этом случае открытые функции-элементы объектов не будут доступны через интерфейсы этих скомпонованных объектов.

### 15.5. Последовательный обход:

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

Обход в ширину:

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

Обратный обход:

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

## Упражнения

- 15.6. Напишите программу, которая сцепляет два объекта связных списков данных символьного типа. Программа должна включать функцию `concatenate`, которая принимает в качестве параметров ссылки на оба списка и сцепляет второй список с первым.
- 15.7. Напишите программу, которая объединяет два объекта упорядоченных списков данных целого типа в единый объект упорядоченного списка. Функция `merge` должна принимать ссылки на каждый из объединяемых списков и возвращать ссылку на объединенный объект.
- 15.8. Напишите программу, которая помещает 25 случайных целых чисел в диапазоне от 0 до 100 в объект упорядоченного связного списка. Программа должна вычислять сумму элементов и среднее арифметическое значение данных (как значение с плавающей запятой).
- 15.9. Напишите программу, которая создает объект связного списка из 10 символов, а затем создает второй объект списка, содержащий копию первого списка, но в обратной последовательности.
- 15.10. Напишите программу, которая вводит строку текста и использует объект стека для печати строки в обратной последовательности.
- 15.11. Напишите программу, которая использует объект стека для определения, является ли строка палиндромом (т.е. строкой, которая одинаково читается как в прямом, так и в обратном направлениях). Программа должна игнорировать пробелы и знаки пунктуации в строке.
- 15.12. Стеки используются для компиляции, чтобы облегчить процесс генерации машинных кодов для вычисления выражений. В этом и следующих упражнениях мы изучим, каким образом компиляторы вычисляют арифметические выражения, состоящие только из констант, операций и круглых скобок.

При математической записи в привычной, естественной форме обычно пишут выражения, подобные  $3 + 4$  и  $7/9$ , в которых операции (здесь + и /) записываются между своими операндами. Это называется *инфиксной формой записи*. В компьютерах используется *постфиксная форма записи*, в которой операция записывается справа после двух своих операндов. Приведенные выше выражения в инфиксной форме записываются в постфиксной форме соответственно как  $3\ 4 +$  и  $7\ 9 /$ .

Для вычисления сложного инфиксного выражения компилятор должен сначала преобразовать выражение в постфиксную форму, а затем вычислить его в этой постфиксной форме. Каждый из этих алгоритмов требует всего одного «прохода» выражения слева направо. Кроме того каждый алгоритм использует объект стека для обеспечения необходимых операций и для других целей.

В этом упражнении вы напишете на C++ вариант алгоритма, преобразующего инфиксную запись в постфиксную. В следующем упражнении вы напишете на C++ вариант алгоритма вычисления выражения в постфиксной форме. Позднее в этой главе вы обнаружите, что коды, написанные вами в этих упражнениях, помогут вам реализовать полноценный работающий компилятор.

Напишите программу, которая преобразовывает простое инфиксное арифметическое выражение (предполагайте, что вводится правильное выражение) с одноразрядными целыми числами, например

$(6 + 2) * 5 - 8 / 4$

в постфиксное выражение. Постфиксный вариант предыдущего инфиксного выражения записывается следующим образом:

$6\ 2 + 5 * 8\ 4 / -$

Программа должна читать выражение в символьный массив *infix* и использовать модифицированные варианты функций стеков, реализованные в этой главе, для создания постфиксного выражения в символьном массиве *postfix*. Алгоритм создания постфиксного выражения приведен ниже:

- 1) Поместите в стек левую круглую скобку '('.
- 2) Добавьте правую круглую скобку ')' в конец символьного массива *infix*.
- 3) Пока стек не пустой, читайте символьный массив *infix* слева направо и выполняйте следующие действия:

Если текущий символ в массиве *infix* является цифрой, то копируйте его в следующий элемент массива *postfix*.

Если текущий символ массива *infix* является левой круглой скобкой, то помещайте его в стек.

Если текущий символ в массиве *infix* является операцией, то

выталкивайте из вершины стека операции (если они там есть), пока их приоритет не меньше приоритета текущей операции, и помещайте их в массив *postfix*.

Поместите текущий символ массива *infix* в стек.

Если текущий символ в массиве `infix` является правой круглой скобкой, то

Выталкивайте операции из вершины стека и вставляйте их в массив `postfix`, пока на вершине стека не появится левая круглая скобка.

Вытолкните из стека (и отбросьте) левую круглую скобку.

В выражении допускаются следующие арифметические операции:

- + сложение
- вычитание
- \* умножение
- / деление
- $\wedge$  возведение в степень
- % вычисление остатка

Каждый узел стека содержит элемент данных и указатель на следующий узел стека.

Вы можете при желании обеспечить следующие функциональные возможности:

- a) Функцию `convertToPostfix`, которая преобразует инфиксное выражение в постфиксное.
- b) Функцию `isOperator`, которая определяет, является ли с операцией.
- c) Функцию `precedence`, которая определяет, является ли приоритет `operator1` меньшим, равным или большим, чем приоритет `operator2`. Функция возвращает соответственно -1, 0 и 1.
- d) Функцию `push`, которая помещает значение в стек.
- e) Функцию `pop`, которая выталкивает значение из стека.
- f) Функцию `stackTop`, которая возвращает значение в вершине стека, не выталкивая его из вершины.
- g) Функцию `isEmpty`, которая определяет, является ли стек пустым.
- h) Функцию `printStack`, которая печатает содержимое стека.

### 15.13. Напишите программу, которая вычисляет постфиксное выражение (полагая, что оно правильное), например

6 2 + 5 \* 8 4 /

Программа должна читать постфиксное выражение, состоящее из цифр и операций, в символьный массив. Используя модифицированные варианты функций стека, реализованные ранее в этой главе, программа должна просматривать (сканировать) выражение и вычислять его. Алгоритм программы состоит в следующем:

- 1) Добавьте нулевой символ ('\0') в конец постфиксного выражения. Когда встречается нулевой символ, то никакая дальнейшая обработка данных выражения не требуется.
- 2) Пока '\0' не встретился, читайте выражение слева направо.

Если текущий символ является цифрой, то

Поместите целое значение этого символа в стек (целое значение символа цифры состоит из значения его кода в наборе символов компьютера минус значение кода '0').

Иначе, если текущий символ является *операцией*, то

Вытолкните два последних элемента из стека в переменные *x* и *y*.

Вычислите выражение (у *операция x*).

Поместите результат вычисления в стек.

3) Когда в выражении встретится нулевой символ, вытолкните значение из вершины стека. Это значение и является результатом вычисления постфиксного выражения.

**Замечание.** На шаге 2) алгоритма, если операцией является '/', в вершине стека находится 2 и следующий элемент стека — 8, то 2 выталкивается в *x*, 8 выталкивается в *y*, вычисляется  $8/2$  и результат 4 помещается в стек. Аналогично выполняется и операция '-'. В выражениях допускаются следующие арифметические операции:

- + сложение
- вычитание
- \* умножение
- / деление
- $\wedge$  возвведение в степень
- % вычисление остатка

Каждый узел стека содержит элемент данных и указатель на следующий узел стека.

Вы можете при желании обеспечить следующие функциональные возможности:

- a) Функцию `evaluatePostfixExpression`, которая вычисляет постфиксное выражение.
- b) Функцию `calculate`, которая вычисляет выражение `op1 operator op2`.
- c) Функцию `push`, которая помещает значение в стек.
- d) Функцию `pop`, которая выталкивает значение из стека.
- e) Функцию `isEmpty` которая определяет, является ли стек пустым.
- f) Функцию `printStack`, которая печатает содержимое стека.

**15.14.** Модифицируйте программу вычисления постфиксных выражений из упражнения 15.13 так, чтобы она могла обрабатывать операнды целого типа, значения которых больше 9.

**15.15. (Моделирование супермаркета).** Напишите программу, которая моделирует очередь в супермаркете. Покупатели (объекты) появляются в ней случайным образом в интервале времени от 1 до 4 минут (будем рассматривать только целые значения). Обслуживается очередной покупатель также случайным образом в интервале времени от 1 до 4 минут (тоже будем рассматривать только целые значения). Очевидно, что скорость обслуживания и скорость по-

ступления покупателей в очередь должны быть сбалансированы. Если средняя скорость поступления покупателей в очередь превышает среднюю скорость их обслуживания, то очередь будет бесконечно расти. Даже при сбалансированных скоростях могут случайным образом появляться длинные очереди. Запустите модель супермаркета при условии 12-часового рабочего дня (720 минут), используя следующий алгоритм:

1) Сгенерируйте случайное целое число в диапазоне от 1 до 4, означающее минуту появления первого покупателя.

2) В момент появления первого покупателя:

Определите время обслуживания (случайное целое число от 1 до 4);

Начните обслуживание покупателя;

Спланируйте время появления следующего покупателя (добавьте к текущему времени случайное целое в диапазоне от 1 до 4);

3) Для каждой минуты дня:

Если появился следующий покупатель, то

Поставьте его в очередь;

Спланируйте время появления следующего покупателя;

Если обслужен очередной покупатель, то

Исключите из очереди следующего покупателя;

Определите время его обслуживания.

Теперь запустите ваше моделирование супермаркета в течение 720 минут и ответьте на следующие вопросы:

a) Какое максимальное число покупателей было в очереди?

b) Каково максимальное время ожидания для покупателя?

c) Что происходит, если интервалы времени изменить с 1–4 минут до 1–3 минут?

**15.16.** Модифицируйте программу, приведенную на рис. 15.16, таким образом, чтобы она позволяла объекту двоичного дерева включать дубликаты.

**15.17.** Напишите программу, основанную на программе, приведенной на рис. 15.16, которая вводила бы строку текста, разбивала бы предложение на отдельные слова (вы можете использовать библиотечную функцию `strtok`), вставляла бы слова в дерево двоичного поиска и печатала маршруты при последовательном обходе дерева, обходе в ширину и обратном обходе. Используйте подходы объектно-ориентированного программирования.

**15.18.** В этой главе мы увидели, что удалять дубликаты проще всего непосредственно при создании дерева двоичного поиска. Опишите, каким образом вы могли бы выполнить удаление дубликатов, используя только один одномерный индексированный массив. Сравните эффективность удаления дубликатов с помощью массива с эффективностью удаления дубликатов на основе дерева двоичного поиска.

- 15.19. Напишите функцию `depth`, которая принимает двоичное дерево и определяет число уровней в нем.
- 15.20. (*Рекурсивная печать списка в обратной последовательности*) Напишите функцию-элемент `printListBackwards`, которая рекурсивно выводит элементы связного списка объектов в обратной последовательности. Напишите программу проверки, которая создает отсортированный список целых чисел и печатает этот список в обратной последовательности.
- 15.21. (*Рекурсивный поиск в списке*) Напишите функцию-элемент `searchList`, которая с помощью рекурсии осуществляет поиск объекта связного списка с заданным значением. Если значение найдено, функция должна возвращать указатель на него; в противном случае, должен быть возвращен нулевой указатель. Используйте вашу функцию в программе проверки, которая создает список целых чисел. Программа должна запрашивать пользователя значения, помещаемые в список.
- 15.22. (*Удаление из двоичного дерева*) В этом упражнении мы обсудим удаление элементов из дерева двоичного поиска. Алгоритм удаления не является таким простым, как алгоритм вставки. При удалении элемента могут быть три случая: элемент находится в концевом узле дерева (не имеет узлов-потомков); элемент находится в узле, в которого есть один узел-потомок; элемент находится в узле, у которого имеется два узла-потомка.

Если удаляемый элемент находится в концевом узле, то он удаляется и соответствующий указатель в его родительском узле устанавливается на нулевое значение.

Если удаляемый элемент находится в узле с одним узлом-потомком, то он удаляется и указатель в его родительском узле устанавливается на этот узел-потомок. В этом случае узел-потомок занимает в дереве место удаленного.

Последний случай — самый сложный. Когда удаляется узел с двумя узлами-потомками, другой узел дерева должен занять его место. Однако, указателю в родительском узле не может быть просто присвоен указатель на один из узлов-потомков удаленного узла. В большинстве случаев, результирующее дерево двоичного поиска без дубликатов должно обладать характерным для деревьев двоичного поиска свойством: *значения в левом поддереве меньше, чем значения в родительском узле, а значения в правом поддереве больше значений в родительском узле*.

Какой узел использовать как замещающий, чтобы сохранить это свойство? Это или узел, содержащий наибольшее значение в дереве, меньшее значения в удаляемом узле, или узел, содержащий наименьшие значения в дереве, большее значения в удаляемом узле. Допустим, что это узел с наименьшим значением. В дереве двоичного поиска наибольшее значение, меньшее значения в родительском узле, размещается в левом поддереве родительского узла и гарантированно находится в самом правом узле этого поддерева. Это узел должен быть размещен путем обхода левого поддерева

сверху вниз и слева направо до тех пор, пока указатель на правый узел-потомок не окажется нулевым. Тогда можно указать на этот замещающий узел, который является либо концевым узлом дерева, либо узлом с одним узлом-потомком, находящимся слева. Если замещающий узел является концевым, то для выполнения операции удаления предпринимаются следующие шаги:

- 1) Указатель на удаляемый узел сохраняется во временной переменной указателе (этот указатель потом используется для освобождения динамически выделенной памяти).
- 2) Указатель в родительском узле удаляемого узла устанавливается так, чтобы он указывал на замещающий узел.
- 3) Указатель в родительском узле замещающего узла устанавливается на нулевое значение.
- 4) Указатель на правое поддерево в замещающем узле устанавливается так, чтобы указывать на правое поддерево удаляемого узла.
- 5) Удаляется узел, на который указывает временная переменная указатель.

Действия, предпринятые для замещающего узла с левым узлом-потомком аналогичны действиям для замещающего узла без узлов-потомков; но алгоритм должен также перемещать и узел-потомок. Если замещающий узел имеет левый узел-потомок, то при удалении предпринимаются следующие действия :

- 1) Указатель на удаляемый узел сохраняется во временной переменной указателе.
- 2) Указатель в родительском узле удаляемого узла устанавливается так, чтобы он указывал на замещающий узел.
- 3) Указатель в родительском узле замещающего узла устанавливается на левый узел-потомок замещающего узла.
- 4) Указатель на правое поддерево в замещающем узле устанавливается так, чтобы указывать на правое поддерево удаляемого узла.
- 5) Удаляется узел, на который указывает временная переменная указатель.

Напишите функцию `deleteNode`, которая принимает в качестве аргумента указатель на корневой узел объекта дерева и удаляемое значение. Функция должна найти в дереве узел, содержащий удаляемое значение, и использовать рассмотренные алгоритмы для его удаления. Если в дереве не найдено это значение, то функция должна напечатать сообщение, которое поясняет, удалено или нет указанное значение. Модифицируйте программу, приведенную на рис. 15.16, используя эту функцию. После удаления элемента, вызовите функции обходов `inOrder`, `preOrder` и `postOrder` для подтверждения того, что операция удаления была выполнена корректно.

- 15.23. (Дерево двоичного поиска)** Напишите функцию `binaryTreeSearch`, которая ищет заданное значение в объекте дерева двоичного поиска. Функция должна принимать в качестве аргументов указатель на корневой узел двоичного дерева и ключ поиска. Если узел, содержащий ключ поиска, найден, то функция должна вернуть ука-

затель на этот узел; в противном случае функция должна вернуть нулевой указатель.

**15.24. (Послойный обход двоичного дерева)** Программа, приведенная на рис. 15.16, демонстрирует три рекурсивных способа обхода двоичного дерева: последовательный обход, обход в ширину и обратный обход. В этом упражнении представлен послойный обход двоичного дерева, при котором значения узлов печатаются от уровня к уровню, начиная с корневого узла. Значения в узлах печатаются по уровням дерева слева направо. Алгоритм послойного обхода не является рекурсивным алгоритмом. Он использует объект очереди для управления процессом вывода узлов на печать. Этот алгоритм заключается в следующем :

- 1) В очередь вставляется корневой узел.
- 2) Пока в очереди существуют узлы,

    Взять из очереди следующий узел

    Напечатать значение этого узла

    Если указатель на левый узел-потомок имеет ненулевое значение

        Вставить левый узел-потомок в очередь

    Если указатель на правый узел-потомок имеет ненулевое значение

        Вставить правый узел-потомок в очередь.

Напишите функцию-элемент `levelOrder` для выполнения послойного обхода двоичного дерева. Модифицируйте программу, приведенную на рис. 15.16, чтобы использовать эту функцию. (Замечание: вам необходимо также модифицировать и встроить в программу функции обработки очереди на рис. 15.12).

**15.25. (Печать дерева)** Напишите рекурсивную функцию-элемент `outputTree` для отображения объекта двоичного дерева на экране. Функция должна выводить дерево по слоям, начиная с вершины, находящейся на экране слева, и далее продвигаясь вниз по дереву и вправо по экрану. Каждый уровень выводится на экране вертикально. Например, двоичное дерево, показанное на рис. 15.19, должно выводиться следующим образом:

|    |    |
|----|----|
|    | 99 |
|    | 97 |
|    | 92 |
| 83 |    |
|    | 72 |
|    | 71 |
|    | 69 |
| 49 |    |
|    | 44 |
|    | 40 |
|    | 32 |
| 28 |    |
|    | 19 |
|    | 18 |
|    | 11 |

Обратите внимание, что самый правый концевой узел дерева появляется вверху экрана в самом правом столбце, а корневой узел появляется в самой левой позиции. Каждый столбец выводится на пять пробелов правее предыдущего столбца. Функция `outputTree` должна принимать аргумент `totalSpace`, представляющий собой число пробелов, предшествующих выводимому значению ( эта переменная должна начинаться с нулевого значения, так как корневой узел выводится на левой стороне экрана). В функции используется для вывода дерева модифицированный последовательный обход: она начинает с самого правого узла дерева и перемещается влево. Алгоритм сводится к следующему:

Пока указатель на текущий узел имеет ненулевое значение

Рекурсивно вызывается `outputTree` для правого поддерева текущего узла с аргументом `totalSpace + 5`

Используется структура `for` со счетчиком от 1 до `totalSpace` для вывода пробелов

Выводится значение в текущем узле

Устанавливается указатель на левое поддерево текущего узла

Увеличивается значение `totalSpace` на 5.

### Специальный раздел: построение вашего собственного компилятора

В упражнениях 5.18 и 5.19 главы 5 мы ввели понятие языка машины Простотрон (ЯМП) и вы создали простейшую программу, моделирующую выполнение программ, написанных на ЯМП. В этом разделе мы построим компилятор, который осуществляет преобразование программ, написанных на языке высокого уровня, в программы на ЯМП. Этот раздел объединяет все элементы и этапы программирования в единое целое. Вы напишете программы на этом новом языке высокого уровня, скомпилируете эти программы с помощью построенного вами компилятора и выполните эти программы на модели, которую вы построили в упражнении 5.19 главы 5. Теперь, вам следует без колебаний приступить к реализации вашего компилятора с помощью объектно-ориентированного подхода.

- 15.26. (Язык Простотрона)** До того, как мы начнем строить компилятор, обсудим простейший, но очень мощный язык программирования высокого уровня, аналогичный ранним версиям популярного языка БЕЙСИК. Мы назовем этот язык *Языком Простотрона — ЯП*. Каждый *оператор ЯП* состоит из *номера строки* и *инструкции ЯП*. Нумерация строк проводится в порядке возрастания. Каждая инструкция начинается с одной из следующих *команд ЯП* : `rem`, `input`, `let`, `print`, `goto`, `if/goto` или `end` (см. рис. 15.20). Все команды, за исключением команды `end`, могут использоваться повторно. В ЯП вычисляются только целые выражения, использующие операции `+`, `-`, `*` и `/`. Эти операции выполняются с тем же приоритетом, что и в С. Для изменения последовательности выполнения операций в выражениях могут использоваться круглые скобки.

| Команда        | Пример оператора        | Описание                                                                                                                                                    |
|----------------|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>rem</b>     | 50 rem это комментарий  | Любой текст, следующий за командой <b>rem</b> , служит только для комментариев и игнорируется компилятором.                                                 |
| <b>input</b>   | 30 input x              | Отображение на экране знака вопроса как приглашения пользователю ввести целое число с клавиатуры; чтение этого числа и сохранение его в x.                  |
| <b>let</b>     | 80 let u = 4 * (j - 56) | Переменной u присваивается значение <b>4 * (j - 56)</b> . Заметьте, что справа от знака присваивания может появиться произвольное комплексное выражение.    |
| <b>print</b>   | 10 print w              | Отображение на экране значения переменной w                                                                                                                 |
| <b>goto</b>    | 70 goto 45              | Передача управления строке с номером <b>45</b>                                                                                                              |
| <b>if/goto</b> | 35 if i == z goto 80    | Сравнение i и z на равенство и передача управления строке с номером <b>80</b> , если это условие выполнено; в противном случае, переход к следующей строке. |
| <b>end</b>     | 99 end                  | Завершение программы.                                                                                                                                       |

Рис. 15.20. Команды ЯП

Наш компилятор с ЯП распознает только строчные буквы. Все символы в файлах, написанных на ЯП, должны быть строчными (прописные буквы приводят к синтаксической ошибке, если только они не появляются в операторе комментариев **rem**, в которых они просто игнорируются). Имя *переменной* состоит из одной буквы. В ЯП не допускаются описательные имена переменных, поэтому переменные следует объяснять в комментариях, чтобы указать на их назначение в программе. ЯП использует только целые переменные. В ЯП не используется никаких объявлений переменных; просто само упоминание имени переменной в программе соответствует объявлению переменной и заданию ей нулевого начального значения. Синтаксис ЯП не допускает операций со строками (чтение строки, запись строки, сравнение строк и т.д.). Если в программе на ЯП встречается строка (после любой команды, кроме команды **rem**), то компилятор генерирует синтаксическую ошибку. Первая версия нашего компилятора будет предполагать, что программы на ЯП введены корректно. В упражнении 15.29 мы попросим студента модифицировать компилятор так, чтобы он осуществлял контроль синтаксических ошибок.

ЯП использует условный оператор **if/goto** и оператор безусловной передачи управления **goto** для изменения потока управления программы. Если условие в операторе **if/goto** истинно, то управление передается заданной строке программы. В операторе **if/goto** допускаются следующие операции отношения и проверки равенства: **<**, **>**, **<=**, **>=**, **==** или **!=**. Старшинство этих операций в ЯП такое же, как и в языке C++.

Давайте теперь рассмотрим несколько программ, демонстрирующих возможности ЯП. Первая программа (рис. 15.21) читает два целых значения с клавиатуры, заносит эти значения в переменные *a* и *b*, вычисляет и печатает их сумму (сохраненную в переменной *c*).

Программа, приведенная на рис. 15.22, определяет и печатает большее из двух целых чисел. Целые числа вводятся с клавиатуры и заносятся в переменные *s* и *t*. Условный оператор *if/goto* проверяет условие *s >= t*. Если условие истинно, то управление передается строке с номером 90 и на экран выводится *s*; в противном случае выводится *t* и управление передается оператору *end* в строке с номером 99, который завершает программу.

```

10 rem расчет и печать суммы двух целых чисел
15 rem
20 rem ввод двух целых чисел
30 input a
40 input b
45 rem
50 rem сложение целых чисел и занесение результата в c
60 let c = a + b
65 rem
70 rem печать результата
80 print c
90 rem завершение выполнения программы
99 end

```

**Рис. 15.21.** Простая программа для определения суммы двух целых чисел

```

10 rem определение максимального из двух целых чисел
20 input s
30 input t
32 rem
35 rem проверка условия s >= t
40 if s >= t goto 90
45 rem
50 rem если t больше s , то печатается t
60 ptint t
70 goto 99
75 rem
80 rem если s больше или равно t , то печатается s
90 print s
99 end

```

**Рис. 15.22.** Простая программа определения максимального из двух целых чисел

ЯП не имеет структур повторения, подобных структурам *for*, *while* или *do/while* в C++. Однако ЯП может моделировать каждую из этих структур повторения, используя операторы условного перехода *if/goto* и безусловного перехода *goto*. На рис. 15.23 показано использование цикла, управляемого меткой, для вычисления квадратов целых чисел. Каждое целое число вводится с клавиатуры и сохраняется в переменной *j*. Если вводимое значение является сигнальной меткой *-9999*, то управление передается строке с номером 99 и программа завершается. В противном случае квадрат *j* присваивается переменной *k*, которая выводится на экран. Далее про-

исходит передача управления строке с номером 20, в которой осуществляется ввод следующего целого числа.

Используя примеры программ, приведенных на рис. 15.21, 15.22 и 15.23, как руководство к действию, напишите программу на ЯП для выполнения следующих заданий:

- a) Введите три целых числа, определите их среднее значение и напечатайте результат.
- b) Используйте цикл, управляемый меткой, для ввода 10 целых чисел; вычислите и напечатайте сумму этих чисел.
- c) Используйте цикл, управляемый счетчиком, для ввода 7 целых чисел (пусть некоторые из них будут положительными, а другие — отрицательными). Вычислите и напечатайте их среднее арифметическое значение.
- d) Введите ряд целых чисел, определите и напечатайте максимальное из них. При этом первое введенное значение должно показывать, сколько чисел следует обработать.
- e) Введите 10 целых чисел и напечатайте минимальное из них.
- f) Вычислите и напечатайте сумму четных чисел от 2 до 30.
- g) Вычислите и напечатайте результат умножения нечетных чисел от 1 до 9.

```
10 rem вычисление квадратов целых чисел
20 input j
23 rem
25 rem проверка значения метки
30 if j == -9999 goto 99
33 rem
35 rem вычисление квадрата j и присваивания результата k
40 let k = j * j
50 print k
53 rem
55 rem цикл для ввода следующего числа в переменную j
60 goto 20
99 end
```

Рис. 15.23. Вычисление квадратов нескольких чисел

**15.27. (Построение компилятора. Примечание: обязательным условием построения вашего компилятора является выполнение в полном объеме упражнений 5.18 и 5.19 главы 5 и упражнений 15.12, 15.13 и 15.26 главы 15).** Теперь, когда у нас имеется описание языка ЯП (упражнение 15.26), мы можем обсудить способы построения компилятора для ЯП. Сначала рассмотрим процесс преобразования программы на ЯП в программу на ЯМП (см. специальный раздел главы 5) и выполнение ее с помощью моделирующей программы Простотрон (см. рис. 15.24). Файл, содержащий программу на ЯП, читается компилятором и преобразуется в код ЯМП. Это код выводится в файл на диске, содержащий инструкции на ЯМП по одной инструкции в строке. Затем этот файл с программой на ЯМП загружается в программу, моделирующую Простотрон, а результаты ее выполнения пересыпаются в файл на диске и на экран. Заметим, что моделирующая программа Простотрон, созданная в

упражнении 5.19 главы 5, получает входные данные с клавиатуры. Она должна быть модифицирована для чтения из файла, чтобы уметь запускать программы, полученные с помощью нашего компилятора.

Компилятор Простотрона выполняет два *прохода* для преобразования программы на ЯП в программу на ЯМП. При первом проходе создается *таблица симвлических имен* (объект), в которой каждый номер строки (тоже объект), имя переменной (объект) и константа (объект) программы на ЯП хранится со своим типом и соответствующим местом расположения в результирующем коде ЯМП (таблица симвлических имен детально рассматривается ниже). Первый проход генерирует также объекты соответствующих инструкций ЯМП для каждого оператора ЯП. Как мы увидим позже, если программа ЯП содержит операторы, которые передают управление какой-либо строке программы, то в результате первого прохода в программе на ЯМП могут иметься некоторые «незавершенные» инструкции. Второй проход компилятора завершает ранее незавершенные инструкции и выводит результирующую программу на ЯМП в файл.

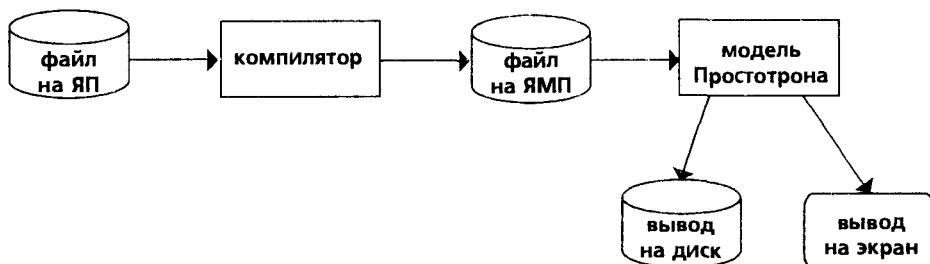


Рис. 15.24. Написание, компиляция и выполнение программы на ЯП

### Первый проход

Компилятор начинает работу с чтения в память первого оператора программы на ЯП. Эта строка должна быть разделена на отдельные лексемы (т.е. смысловые части оператора) для дальнейшей обработки и компиляции (для упрощения этой задачи может быть использована стандартная библиотечная функция `strtok`). Вспомните, что каждый оператор начинается с номера строки, после которого следует команда. После того, как компилятор разбивает оператор на лексемы, те лексемы, которые являются номером строки, переменной или константой, помещаются в таблицу симвлических имен. Следует отметить, что номер строки помещается в таблицу симвлических имен только в случае, если он является первой лексемой в операторе. Объект `symbolTable` (таблица симвлических имен) является массивом объектов типа `tableEntry`, представляющими каждый символ в программе. На число символов, которые могут появиться в программе, ограничений не существует. Следовательно, таблица симвлических имен `symbolTable` для конкретной программы может быть довольно большой по объему. Давайте

создадим таблицу символьических имен `symbolTable` на 100 элементов. Вы можете увеличить или уменьшить ее размер после того, как программа заработает.

Каждый объект `tableEntry` содержит три элемента. Элемент `symbol` является целым, содержащим или код ASCII имени переменной (вспомним, что имена переменных являются одиночными символами), или номер строки, или константу. Элемент `type` является одним из следующих символов, указывающих на тип элемента `symbol`: 'C' — константа, 'L' — номер строки и 'V' — переменная. Элемент `location` содержит ячейку памяти (адрес) Простотрона (от 00 до 99), на которую ссылается данный объект `tableEntry`. Память Простотрона является массивом из 100 целых чисел, в котором хранятся инструкции ЯМП и данные. Если `tableEntry` является номером строки, то `location` указывает элемент массива в памяти Простотрона, в котором начинается инструкция ЯМП для этого оператора ЯП. Для `tableEntry`, являющихся переменными и константами, `location` указывает элемент массива в памяти Простотрона, в котором эта переменная или константа хранится. Переменные и константы размещаются в памяти Простотрона, начиная с последней ячейки по направлению к первой ячейке: первая переменная или константа хранится в ячейке 99, следующая — в ячейке 98 и т.д.

Таблица символьических имен играет решающую роль в преобразовании программ на ЯП в программы на ЯМП. В главе 5 мы узнали, что инструкция ЯМП представляет собой четырехзначное число, состоящее из двух частей: *кода операции* и *операнда*. Код операции определяется командой ЯП. Например, команда ЯП `input` соответствует коду операции ЯМП 10 (читать), а команда `print` — коду 11 (записать). Операнд — это ячейка памяти, содержащая данные, с которыми работает операция (например, операция с кодом 10 читает значение с клавиатуры и заносит его в ячейку памяти, заданную операндом). Компилятор просматривает `symbolTable` и ищет для каждого символа соответствующую ячейку памяти Простотрона, которая и используется для окончательного оформления команды ЯМП.

Компиляция каждого оператора ЯП зависит от его команды. Например, после того, как номер строки оператора `rem` помещается в таблицу символьических имен, остальные символы оператора комментариев компилятор игнорирует, так как они предназначены только для пояснений операторов программы. Операторы `input`, `print`, `goto` и `end` соответствуют командам ЯМП `read`, `write`, `branch` (для указанной ячейки) и `halt`. Операторы, содержащие эти команды ЯП, непосредственно преобразуются в соответствующие инструкции ЯМП (заметим, что оператор `goto` может содержать пока неразрешенную ссылку, если указанный номер строки ссылается на один из последующих операторов в файле программы на ЯП; это иногда называют *ссылкой вперед*).

Когда оператор `goto` компилируется с неразрешенной ссылкой, инструкция ЯМП должна быть помечена флагом, чтобы при втором проходе компилятора программа завершила эту инструкцию.

Флаги хранятся в массиве `flags` из 100 элементов типа `int`, в котором каждому элементу задано начальное значение -1. Если ячейка памяти, на которую ссылается номер строки в программе на ЯП, еще неизвестна, (т.е., если ее нет в таблице символических имен), то номер этой строки заносится в массив `flags` в элемент с индексом, соответствующим незавершенной команде. Операнд незавершенной команды устанавливается временно на 00. Например, оператор безусловного перехода, делающий ссылку вперед, оставляется равным +4000 до второго прохода компилятора, который скоро будет рассмотрен.

Компиляция операторов `if/goto` и `let` является более сложной, чем компиляция других операторов, поскольку это единственные операторы, которые соответствуют более, чем одной инструкции ЯМП. Для оператора `if/goto` компилятор должен генерировать код проверки условия и код передачи управления в зависимости от результатов проверки. К тому же эта передача управления может быть неразрешенной ссылкой. Каждая из операций отношения может быть смоделирована с помощью команд ЯМП *перехода по нулю* (`branchzero`) и *перехода по отрицательному значению* (`branchnegative`).

Для оператора `let` компилятор создает код выполнения произвольных арифметических операций с комплексными числами, содержащими целые переменные и константы. Выражения должны отделять каждый operand от символа операции пробелами. В упражнениях 15.12 и 15.13 представлены алгоритмы преобразования инфиксного выражения в постфиксное и вычисления постфиксного выражения, используемые компиляторами для вычисления выражения. Прежде, чем разрабатывать компилятор, вам следует выполнить эти два упражнения. Когда компилятор встречает выражение, он преобразует его из инфиксной записи в постфиксную, а затем вычисляет постфиксное выражение.

Как компилятор может создавать машинный код для вычисления выражения, содержащего переменные? Алгоритм расчета постфиксного выражения можно модифицировать так, чтобы компилятор не производил вычисление выражения, а генерировал бы инструкции ЯМП. Чтобы ввести это добавление в компилятор, необходимо модифицировать алгоритм вычисления постфиксного выражения, введя в него поиск встречающихся в выражении переменных и констант в таблице (или вставку их в таблицу, если их там еще нет), определение соответствующей ячейки памяти, связанной с этой переменной или константой, и размещение в стеке адреса этой ячейки памяти (*вместо значения константы, как это было в исходном алгоритме*). Когда в постфиксном выражении встречается операция, то из вершины стека выталкиваются два адреса и генерируется инструкция ЯМП, использующая эти адреса как operandы. Результат каждого подвыражения временно сохраняется в памяти и адрес этой временной ячейки помещается обратно в стек, чтобы вычисление постфиксного выражения могло продолжаться. Когда вычисление постфиксного выражения завершено, адрес результата, является единственным оставшимся эле-

ментом в стеке. Он выталкивается из стека и компилятор генерирует инструкцию ЯМП, присваивающую этот результат, хранящийся в этом адресе, той переменной, которая находится в левой части оператора `let`.

### Второй проход

Второй проход компилятора выполняет две задачи: разрешение неразрешенных ссылок и вывод кода ЯМП в файл. Разрешение ссылок производится следующим образом:

- 1) В массиве `flags` ищется неразрешенная ссылка (т.е. элемент со значением, отличным от -1).
- 2) В массив `symbolTable` помещается объект, содержащий элемент, номер которого хранится в массиве `flags` (его тип 'L' — номер строки).
- 3) Адрес из элемента `location` вставляется в инструкцию с неразрешенной ссылкой (вспомним, что у инструкций, содержащих неразрешенную ссылку, имеется operand 00).
- 4) Шаги 1, 2 и 3 повторяются до тех пор, пока не будет достигнут конец массива `flags`.

После того, как процесс разрешения ссылок завершен, весь массив, содержащий код ЯМП, выводится в файл на диске по одной инструкции ЯМП на строку. Этот файл может быть прочитан моделью Простотрона для выполнения (после модификации моделирующей программы, чтобы она читала свои входные данные из файла). Компиляция вашей первой программы на ЯП в файл программы на ЯМП и последующая обработка этого файла несомненно дадут вам удовлетворение.

### Итоговый пример

Следующий пример иллюстрирует полное преобразование программы на ЯП в программу на ЯМП таким образом и в той последовательности, как это выполняется компилятором Простотрона. Допустим, что программа на ЯП вводит целое число и суммирует значения от 1 до этого целого. Программа и инструкции ЯМП, сгенерированные при первом проходе компилятора Простотрона, показаны на рис. 15.25. Таблица символических имен после первого прохода приведена на рис. 15.26.

Большая часть операторов непосредственно преобразуется в команды ЯМП. Исключениями в этой программе являются операторы комментариев, оператор `if/goto` в строке 20 и операторы `let`. Комментарии не транслируются в машинный код. Однако, номер строки комментария помещается в таблицу символических имен на тот случай, если на этот номер будут ссылаться операторы `goto` или `if/goto`. В строке с номером 20 задается условие `y == x`, при выполнении которого происходит передача управления строке с номером 60. Поскольку строка с номером 60 появляется в программе позднее, то при первом проходе номер строки 60 не помещен в таблицу символических имен (номера строк операторов по-

мещаются в таблицу символических имен только если они появляются в качестве первой лексемы в операторе). Следовательно, в этот момент невозможно установить operand инструкции ЯМП передачи управления по нулю, помещаемой в ячейку 03 массива инструкций ЯМП. Компилятор помещает 60 в ячейку 03 массива flags, чтобы показать, что эту инструкцию должен завершить второй проход.

Мы обязаны сохранить этот след адреса, поскольку нет взаимно-однозначного соответствия между операторами Простотрона и инструкциями ЯМП. Например, оператор *if/goto* в строке с номером 20 компилируется в три инструкции ЯМП. Каждый раз, когда генерируется инструкция, мы должны увеличивать *счетчик команд*, указывающий следующую ячейку массива ЯМП. Заметим, что объем памяти Простотрона может создать проблему для программ на ЯП, содержащих много операторов, переменных и констант. Возможно, что компилятор выйдет за пределы памяти. Чтобы проверять этот случай, следует включить в вашу программу *счетчик данных* для хранения адреса следующей переменной или константы в массиве ЯМП. Если значение счетчика команд больше значения счетчика данных, значит массив ЯМП полностью заполнен. В этом случае процесс компиляции должен завершиться и компилятор должен выдать сообщение об ошибке переполнения памяти во время компиляции. Следует отметить, что хотя компилятор освобождает программиста от забот о памяти, сам компилятор должен корректно вычислять местоположение команд и данных в памяти и обнаруживать такие ошибки, как отсутствие свободного места в памяти во время компиляции.

### Пошаговый анализ процесса компиляции

Проанализируем весь процесс компиляции программы на ЯП, приведенной на рис. 15.25. Компилятор читает первую строку программы

```
5 rem sum 1 to x
```

в память. Первая лексема в операторе (номер строки) определяется с помощью *strtok* (см. главы 5 и 16, в которых рассмотрены функции обработки строк в C++). Лексема, возвращенная *strtok*, преобразуется в целое число с помощью *atoi* и символ 5 ищется в таблице символических имен. Если указанный символ не найден, то он помещается в таблицу. Поскольку мы находимся пока в самом начале программы и смотрим первую строку, то никаких символов в таблице еще нет. Поэтому символ 5 помещается в таблицу как тип L (номер строки) и ему присваивается номер первой ячейки массива ЯМП (00). Хотя эта строка является комментарием, в таблице все-таки выделено для нее место (на случай, если на нее будут ссылаться операторы *goto* и *if/goto*). Для оператора *rem* не генерируется никакая команда, поэтому счетчик команд не увеличивается.

| Программа на ЯП         | Ячейка и команда ЯМП                                     | Описание                                                                                                                                                                   |
|-------------------------|----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 5 rem сумма от 1 до x   | нет                                                      | Оператор rem игнорируется                                                                                                                                                  |
| 10 input x              | 00 +1099                                                 | Читать x в ячейку 99                                                                                                                                                       |
| 15 rem проверка y == x  | нет                                                      | Оператор rem игнорируется                                                                                                                                                  |
| 20 if y == x goto 60    | 01 +2098<br>02 +3199<br>03 +4200                         | Загрузить y (98) в аккумулятор<br>Вычесть x (99) из аккумулятора<br>Передать управление при нуле на неразрешенную ссылку                                                   |
| 25 rem увеличение y     | нет                                                      | Оператор rem игнорируется                                                                                                                                                  |
| 30 let y = y + 1        | 04 +2098<br>05 +3097<br>06 +2196<br>07 +2096<br>08 +2198 | Загрузить y (98) в аккумулятор<br>Сложить 1 (97) с аккумулятором<br>Сохранить во временной ячейке 96<br>Загрузить из временной ячейки 96<br>Сохранить аккумулятор в y (98) |
| 35 rem сложение y и t   | нет                                                      | Оператор rem игнорируется                                                                                                                                                  |
| 40 let t = t + y        | 09 +2095<br>10 +3098<br>11 +2194<br>12 +2094<br>13 +2195 | Загрузить t (95) в аккумулятор<br>Сложить y (98) с аккумулятором<br>Сохранить в временной ячейке 94<br>Загрузить из временной ячейки 94<br>Сохранить аккумулятор в t (95)  |
| 45 rem цикл по y        | нет                                                      | Оператор rem игнорируется                                                                                                                                                  |
| 50 goto 20              | 14 +4001                                                 | Передать управления в ячейку 01                                                                                                                                            |
| 55 rem вывод результата | нет                                                      | Оператор rem игнорируется                                                                                                                                                  |
| 60 print t              | 15 +1195                                                 | Вывести t (95) на экран                                                                                                                                                    |
| 99 end                  | 16 +4300                                                 | Завершение программы                                                                                                                                                       |

Рис. 15.25. Инструкции ЯМП, сгенерированные после первого прохода компилятора

Следующим разбивается на лексемы оператор

10 input x

Номер строки 10 помещается в таблицу символьических имен как тип L и ему присваивается первая ячейка массива ЯМП (00, поскольку первым оператором программы был оператор комментария rem и значение счетчика команд не увеличилось). Команда input показывает, что следующей лексемой является переменная (только переменная может присутствовать в операторе input). Поскольку input непосредственно соответствует коду операции ЯМП, то компилятор просто должен определить ячейку для x в массиве ЯМП. Символ x не найден в таблице символьических имен. Поэтому он помещается в таблицу символьических имен в виде кода ASCII для символа x как тип V и ему присваивается адрес 99 в массиве ЯМП (память для хранения данных распределяется в обратном

направлении, начиная с ячейки 99 в порядке убывания номеров ячеек). Теперь для этого оператора может быть сгенерирован код ЯМП. Код операции 10 (код операции чтения в ЯМП) умножается на 100 и к нему добавляется адрес x, определяемый из таблицы символьических имен). Сгенерированная инструкция сохраняется в массиве ЯМП в ячейке 00. Счетчик команд увеличивается на 1, поскольку была сгенерирована одна команда ЯМП.

| Символ | Тип | Ячейка |
|--------|-----|--------|
| 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. 15.26. Таблица символьических имен для программы, приведенной на рис. 15.25

Следующим разбивается на лексемы оператор

```
15 rem check y == x
```

В таблице символьических имен осуществляется поиск номера строки 15 (который не находится). Номер строки помещается в таблицу как тип L и ему присваивается следующая ячейка массива 01 (вспомним, что операторы rem не генерируют код, поэтому значение счетчика команд не увеличивается).

Затем разбивается на лексемы оператор

```
20 if y == x goto 60
```

Номер строки 20 помещается в таблицу символьических имен как тип L и ему присваивается очередная ячейка массива ЯМП 01. Команда if показывает, что должно вычисляться условие. Переменная y не найдена в таблице, поэтому она помещается в нее и ей задается тип V и ячейка ЯМП 98. Затем генерируются инструк-

ции ЯМП для оценки указанного условия. Так как в ЯМП не существует прямого аналога оператора `if/goto`, он должен быть смоделирован расчетом выражения, содержащего `x` и `y`, и условной передачей управления в зависимости от полученного результата. Если `у` равен `x`, то результат вычитания `x` из `y` равен нулю; поэтому можно использовать команду ЯМП *передача управления по нулю branchzero*, которая с помощью анализа результата вычитания промоделирует оператор `if/goto`. Первый шаг этой имитации требует загрузить `y` (из ячейки 98 ЯМП) в аккумулятор. Эту операцию выполняет инструкция `01 +2098`. Затем из значения в аккумуляторе вычитается `x`. Этую операцию выполняет инструкция `02 +3199`. Значение в аккумуляторе может оказаться нулевым, положительным или отрицательным. Так как выполняется операция `==`, мы хотим осуществить передачу управления по нулевому значению. Сначала проводится поиск в таблице номера строки, в которую передается управление (в нашем случае 60), но этот номер не находится. Поэтому 60 помещается в ячейку 03 массива `flags` и генерируется инструкция `03 +4200` (мы не можем добавить к этой инструкции ячейку, которой передается управление, поскольку ячейка массива ЯМП, соответствующая номеру строки 60, еще не известна). Значение счетчика команд увеличивается и становится равным 04.

Компилятор обрабатывает оператор

`25 rem increment y`

Номер строки помещается в таблицу символьических имен как тип `L` и ему присваивается ячейка ЯМП 04. Значение счетчика команд в данном случае не увеличивается.

Когда разбивается на лексемы оператор

`30 let y = y + 1`

номер строки помещается в таблицу символьических имен как тип `L` и ему присваивается ячейка ЯМП 04. Команда `let` показывает, что эта строка является оператором присваивания. Сначала все символы в строке помещаются в таблицу символьических имен (в том случае, если их в таблице еще нет). Целое значение 1 добавляется в таблицу символьических имен как тип `C` и ему присваивается ячейка ЯМП 97. Затем правая часть оператора присваивания преобразуется из инфиксной записи в постфиксную. Далее оценивается постфиксное выражение (`y 1 +`). Символ `y` уже имеется в таблице символьических имен и соответствующая ему ячейка памяти помещается в стек. Символ `1` также уже имеется в таблице символьических имен и соответствующая ему ячейка памяти также помещается в стек. Когда встречается операция `+`, компилятор постфиксного выражения сначала выталкивает из стека правый operand этой операции, затем выталкивает из стека левый operand, после чего генерируются инструкции ЯМП

`04 +2098 (загрузить y)`  
`05 +3097 (прибавить 1)`

Результат выражения сохраняется во временной ячейке (96) с помощью инструкции

06 +2196 (сохранить во временной ячейке)

и адрес этой временной ячейки помещается в стек. Теперь, когда вычисление выражения закончено, необходимо сохранить результат в у (т.е. в переменной в левой части оператора присваивания). Для этого временная ячейка загружается в аккумулятор и затем значение в аккумуляторе сохраняется в у с помощью инструкций

07 +2096 (загрузить временную ячейку)

08 +2198 (сохранить у)

Читатель, конечно, сразу же заметит, что программа на ЯМП получается избыточной. Мы вскоре обсудим этот вопрос.

Когда разбивается на лексемы оператор

35 rem add y to total

номер строки 35 помещается в таблицу символьических имен как тип L и ему присваивается ячейка 09.

Оператор

40 let t = t + y

подобен строке с номером 30. Переменная t помещается в таблицу символьических имен как тип V и связывается с ячейкой ЯМП 95. Генерируются команды с той же логикой и форматом, что и для строки с номером 30: 09 +2095, 10 +3098, 11 +2194 и 13 +2195. Заметим, что результат операции t + y передается на временное хранение в ячейку 94 перед тем, как присваивается переменной t (95). Снова читатель может заметить, что инструкции, расположенные в ячейках 11 и 12, избыточны. И снова обещаем вскорости обсудить этот вопрос.

Оператор

45 rem loop y

является комментарием, так что номер строки 45 помещается в таблицу как тип L и связывается с ячейкой ЯМП 14.

Оператор

50 goto 20

передает управление строке с номером 20. Номер строки 50 помещается в таблицу символьических имен как тип L и ему приписывается ячейка ЯМП 14. Эквивалентом оператора goto в ЯМП является команда *безусловного перехода branch* (40), которая передает управление заданной ячейке ЯМП. Компилятор проводит поиск номера строки 20 и обнаруживает, что он соответствует ячейке ЯМП 01. Код операции (40) умножается на 100 и складывается с адресом 01. В итоге генерируется инструкция 14 +4001.

Обрабатывается оператор

55 rem output result

Номер строки 55 помещается в таблицу символьических имен как тип L и связывается с ячейкой ЯМП 15.

Обрабатывается оператор печати

```
60 print t
```

Номер строки 60 помещается в таблицу символьических имен как тип L и связывается с ячейкой ЯМП 15. Аналогом команды `print` в ЯМП является операция с кодом 11 (`write`). Ячейка, где хранится `t`, определяется из таблицы символьических имен и добавляется к результату, полученному после того, как код операции умножили на 100.

Оператор

```
99 end
```

является последней строкой программы. Номер строки 99 заносится в таблицу символьических имен как тип L и связывается с ячейкой ЯМП 16. Команда `end` приводит к генерации инструкции ЯМП **+4300** (43 — это код команды `halt` в ЯМП), которая записывается как последняя команда в массив ЯМП.

На этом первый проход компилятора завершен. Теперь рассмотрим второй проход. В массиве `flags` осуществляется поиск значений, отличных от -1. В ячейке 03 хранится 60, поэтому компилятор знает, что инструкция 03 не завершена. Компилятор завершает формирование этой инструкции, осуществляя поиск в таблице символьических имен номера строки 60, определяя соответствующую ей ячейку памяти ЯМП и складывая этот адрес с незавершенной инструкцией. В данном случае в процессе поиска определяется, что номер строки 60 соответствует ячейке ЯМП 15, так что завершенная инструкция **03 +4215** заменяет инструкцию **03 +4200**. Теперь компиляция программы на ЯП успешна завершена.

Для построения компилятора, вы должны выполнить следующее:

a) Модифицируйте моделирующую программу Простотрона, написанную вами в упражнении 5.19, чтобы она брала входные данные из файла, указанного пользователем (см. главу 14). Моделирующая программа должна выводить полученные результаты в файл на диске в том же самом формате, что и при выводе на экран. Преобразуйте моделирующую программу так, чтобы она была объектно-ориентированной. В частности, сделайте объектами все внешние устройства. Создайте иерархию классов типов инструкций, используя механизм наследования. Затем введите в программу полиморфизм, обращаясь к каждому объекту-инструкции с вызовом `executeInstruction`, по которому должны выполняться действия соответствующей инструкции языка ЯМП.

b) Модифицируйте алгоритм преобразования инфиксной записи в постфиксную (упражнение 15.12) для обработки многоразрядных целых operandов и operandов имен переменных, состоящих из одного символа. Совет. Для разбиения выражения на лексемы и выделения из него констант и переменных может быть использована функция `strtok` из стандартной библиотеки; константы могут

быть преобразованы из строк в целые числа с помощью функции `atoi` из стандартной библиотеки. (Замечание: представление данных в постфиксном выражении должно быть изменено так, чтобы поддерживались имена переменных и целые константы).

с) Модифицируйте алгоритм вычисления постфиксного выражения для обработки многоразрядных целых операндов и операндов имен переменных. В алгоритме должны быть также выполнены обсуждавшиеся выше изменения, сводящиеся к тому, что вместо выполнения выражения должны генерироваться соответствующие инструкции ЯМП. Совет: для разбиения выражения на лексемы и выделения из него констант и переменных может быть использована функция `strtok` из стандартной библиотеки; константы могут быть преобразованы из строк в целые числа с помощью функции `atoi` из стандартной библиотеки. (Замечание: представление данных в постфиксном выражении должно быть изменено, чтобы поддерживать имена переменных и целые константы).

д) Постройте компилятор. Объедините части (б) и (с) для обработки выражений в операторах `let`. В вашу программу надо включить функцию, которая выполняет первый проход компилятора, и функцию, выполняющую второй проход. Обе функции для выполнения своих задач могут вызывать другие функции. Создайте, по возможности, ваш компилятор объектно-ориентированным.

**15.28. (Оптимизация компилятора ЯП)** При компиляции программы и ее преобразования в коды ЯМП генерируется набор инструкций. Определенные комбинации инструкций часто повторяются, обычно тройками, которые можно назвать правилами вывода — продукциями. Продукция обычно состоит из трех инструкций: `load`, `add` и `store`. Например, на рис. 15.27 показаны пять инструкций ЯМП, которые были сгенерированы при компиляции программы, приведенной на рис. 15.25. Первые три инструкции являются продукцией, которая добавляет 1 к у. Заметим, что инструкции 06 и 07 сохраняют значение аккумулятора во временной ячейке 96, а затем загружают это значение обратно в аккумулятор, чтобы инструкция 08 могла сохранить его в ячейке 98. Часто продукция завершается инструкцией загрузки значения в ту же самую ячейку, в которой оно ранее хранилось. Этот код может быть оптимизирован путем исключения команды сохранения и последующей команды загрузки, которые оперируют с одной и той же ячейкой; это позволит Простотрону быстрее выполнять программу. На рис. 15.28 показан оптимизированный код ЯМП для программы, приведенной на рис. 15.25. Отметим, что в оптимизированном коде на четыре инструкции меньше, что снижает объем требуемой памяти на 25%.

Модифицируйте компилятор таким образом, чтобы он оптимизировал генерируемый им код ЯМП. Вручную сравните неоптимизированный код с оптимизированным и определите, на сколько снизился объем требуемой памяти.

```

04 +2098 (загрузка)
05 +3097 (сложение)
06 +2196 (сохранение)

07 +2096 (загрузка)
08 +2198 (сохранение)

```

Рис. 15.27. Неоптимизированный фрагмент кода программы, приведенной на рис. 15.25

| Программа на ЯП         | Ячейка и команда ЯМП             | Описание                                                                                                     |
|-------------------------|----------------------------------|--------------------------------------------------------------------------------------------------------------|
| 5 rem сумма от 1 до x   | нет                              | Оператор rem игнорируется                                                                                    |
| 10 input x              | 00 +1099                         | Читать x в ячейку 99                                                                                         |
| 15 rem проверка y == x  | нет                              | Оператор rem игнорируется                                                                                    |
| 20 if y == x goto 60    | 01 +2098<br>02 +3199<br>03 +4211 | Загрузить y (98) в аккумулятор<br>Вычесть x (99) из аккумулятора<br>Передать управление при нуле в ячейку 11 |
| 25 rem увеличение y     | нет                              | Оператор rem игнорируется                                                                                    |
| 30 let y = y + 1        | 04 +2098<br>05 +3097<br>06 +2198 | Загрузить y в аккумулятор<br>Сложить 1 (97) с аккумулятором<br>Сохранить аккумулятор в y (98)                |
| 35 rem сложение y и t   | нет                              | Оператор rem игнорируется                                                                                    |
| 40 let t = t + y        | 07 +2096<br>08 +3098<br>09 +2196 | Загрузить t (96) в аккумулятор<br>Сложить y (98) с аккумулятором<br>Сохранить аккумулятор в t (96)           |
| 45 rem цикл по y        | нет                              | Оператор rem игнорируется                                                                                    |
| 50 goto 20              | 10 +4001                         | Передать управления ячейке 01                                                                                |
| 55 rem вывод результата | нет                              | Оператор rem игнорируется                                                                                    |
| 60 print t              | 11 +1196                         | Вывести t (96) на экран                                                                                      |
| 99 end                  | 12 +4300                         | Завершение программы                                                                                         |

Рис. 15.28. Оптимизированный код программы, приведенной на рис. 15.25

**15.29.** (*Модификации компилятора ЯП*) Выполните указанные ниже модификации компилятора ЯП. Некоторые из этих модификаций могут также потребовать внести изменения в моделирующую программу Простотрона, написанную в упражнении 5.19.

- Разрешите использовать в операторах `let` операцию вычисления остатка (%). Необходимо модифицировать также моделирующую программу Простотрона, чтобы включить в нее инструкцию вычисления остатка.
- Разрешите использовать в операторах `let` операцию возведения в степень, применяя для нее символ  $\wedge$ . Необходимо модифицировать также моделирующую программу Простотрона, чтобы включить в нее инструкцию возведения в степень.

- c) Разрешите компилятору распознавать в операторах ЯП буквы в верхнем и нижнем регистрах (например, считать, что 'A' эквивалентно 'a'). Никаких модификаций моделирующей программы не требуется.
- d) Разрешите оператору `input` читать значения сразу нескольких переменных, например, `input x, y`. Никаких модификаций моделирующей программы не требуется.
- e) Разрешите оператору `print` выводить значения сразу нескольких переменных, например, `print x, y`. Никаких модификаций моделирующей программы не требуется.
- f) Добавьте в компилятор проверку синтаксиса, чтобы он выводил сообщения о синтаксических ошибках, встретившихся в программе на ЯП. Никаких модификаций моделирующей программы не требуется.
- g) Разрешите использовать массивы целых чисел. Никаких модификаций моделирующей программы не требуется.
- h) Разрешите определять подпрограммы, задаваемые при помощи команд ЯП `gosub` и `return`. Команда `gosub` передает управление подпрограмме, а команда `return` возвращает управление на оператор, следующий за `gosub`. Это похоже на вызов функции в C++. Одна и та же подпрограмма может быть вызвана с помощью нескольких команд `gosub`, размещенных в разных местах программы. Никаких модификаций моделирующей программы не требуется.
- i) Разрешите использовать структуры повторения вида
- ```
for x = 2 to 10 step 2
    Операторы ЯП
next
```
- Этот оператор `for` организует цикл от 2 до 10 с шагом 2. Стока `next` обозначает конец тела цикла `for`. Никаких модификаций моделирующей программы не требуется.
- j) Разрешите компилятору обрабатывать вводимые и выводимые строки. Это потребует модификации моделирующей программы Простотрона, чтобы она могла обрабатывать и сохранять строки. Совет: Каждое машинное слово Простотрона можно разделить на две группы — на два полуслова, каждое из которых состоит из двухразрядного целого. Каждое двухразрядное целое представляет десятичный эквивалент кода ASCII символа. Добавьте в ЯМП инструкцию, которая будет печатать строку, начиная с определенной ячейки памяти Простотрона. Первое полуслово в этой ячейке содержит число символов в строке (т.е. длину строки). Каждое последующее полуслово содержит двухразрядный десятичный код ASCII очередного символа. Инструкция печати воспринимает длину строки и печатает эту строку, преобразуя каждое двухразрядное значение в его символьный эквивалент.
- k) Разрешите компилятору обрабатывать значения с плавающей запятой в дополнение к обработке целых значений. Моделирующая программа Простотрона также должна быть модифицирована для обработки значений с плавающей запятой.

15.30. (Интерпретатор ЯП) Интерпретатор — это программа, которая поочередно читает операторы кода, написанного на языке высокого уровня, определяет операцию, которая должна быть выполнена с помощью данного оператора, и немедленно выполняет эту операцию. Программу, написанную на языке высокого уровня, не надо при этом преобразовывать в программу на машинном языке. Интерпретаторы работают медленно, поскольку каждый оператор в программе должен быть сначала декодирован. Если операторы включены в тело цикла, то они декодируются на каждом шаге цикла. Более ранние версии языка программирования БЕЙСИК были реализованы именно как интерпретаторы.

Напишите интерпретатор для ЯП, рассмотренного в упражнении 15.26. Для вычисления выражений, входящих в оператор `let`, в программе необходимо использовать разработанный в упражнении 15.12 преобразователь инфиксной формы выражений в постфиксную и программу вычисления постфиксного выражения, разработанную в упражнении 15.13. В этой программе интерпретатора следует придерживаться тех же ограничений ЯП, которые были введены в упражнении 15.26. Проверьте интерпретатор на программах, написанных на ЯП в упражнении 15.26. Сравните результаты выполнения этих программ интерпретатором с результатами компиляции и выполнения этих программ моделирующей программой, построенной в упражнении 5.19.

15.31. (Вставка и удаление любого элемента связного списка) Наш шаблон класса связных списков выполняет операции вставки и удаления элементов только в начале и конце списка. Эти возможности удобны в том случае, если мы используем скрытое наследование и композицию для разработки шаблонов класса стеков и класса очередей с помощью незначительных добавлений к повторно используемому шаблону класса связных списков. Действительно, связные списки являются наиболее общими структурами, которые мы используем в программах. Модифицируйте шаблон класса связных списков, который мы создали в этой главе, для выполнения операций вставки и удаления любого элемента списка.

15.32. (Списки и очереди без указателей на конец) Наша реализация связного списка (рис. 15.3) использует два указателя на начало и конец: `firstPtr` и `lastPtr`. Указатель `lastPtr` используется функциями-элементами `insertAtBack` и `removeFromBack` класса `List`. Функция `insertAtBack` соответствует функции `enqueue` класса `Queue`. Перепишите класс `List` так, чтобы он не использовал `lastPtr`. В этом случае любые операции в конце списка должны начинаться с про-мотра всего списка с его начала до конца. Влияет ли это на реали-зацию класса `Queue` (рис. 15.12)?

15.33. Используйте вариант программы стека с композицией (рис. 15.11) для создания законченной работающей программы. Модифицируйте эту программу, введя в нее встраиваемые функции-элементы. Сравните эти два подхода. Обобщите достоинства и недостатки встраиваемых функций-элементов.

15.34. (*Сортировка и поиск на бинарном дереве*) Одной из проблем сортировки бинарного дерева является то, что последовательность, в которой вставляются данные, влияет на его структуру: для того же самого набора данных различная последовательность их появления может кардинально изменять форму дерева. Производительность обработки бинарного дерева алгоритмами сортировки и поиска чувствительна к структуре дерева. Какую структуру будет иметь бинарное дерево, если данные вставлялись в него в порядке возрастания? А если они вставлялись в порядке убывания? Какую структуру должно иметь дерево, чтобы достигнуть максимальной производительности поиска.

15.35. (*Индексированные списки*) Мы рассматривали связные списки, которые должны просматриваться последовательно. Для больших списков это может приводить к плохой производительности. Обычный способ улучшения производительности поиска заключается в создании и поддержке индексных указателей списка. Индексный указатель — это множество указателей на места размещения в списке различных ключей. Например, приложение, которое осуществляет поиск в большом списке имен, может повысить производительность, создав индексный указатель с 26 элементами, по одному на каждую букву английского алфавита. Тогда, например, поиск последнего имени, начинающегося с 'Y' будет начинаться с поиска в индексном указателе желательного элемента 'Y', а затем можно проводить поиск в списке, начиная с той позиции, на которую указывает соответствующий индекс. Поиск будет много быстрее, чем если бы пришлось просматривать весь список с самого начала. Используйте класс `List` на рис. 15.3 как основу класса индексированных списков `IndexedList`. Напишите программу, которая осуществляет операции с индексированными списками. Не забудьте включить функции-элементы `insertInIndexedList` (вставить), `searchIndexedList` (поиск) и `deleteFromIndexedList` (удалить).

16

Биты, символы, строки и структуры



Ц е л и

- Научиться создавать и использовать структуры.
- Научиться передавать структуры в функции вызовом по ссылке и вызовом по значению.
- Научиться манипулировать данными с помощью полуразрядных операций и создавать битовые поля для компактного хранения данных.
- Научиться использовать функции библиотеки обработки символов (**ctype**), библиотеки утилит общего назначения (**stdlib**) и библиотеки обработки строк (**string**).
- Оценить мощность библиотек функций как средства повторного использования программного обеспечения.

План

- 16.1. Введение
- 16.2. Описание структур
- 16.3. Инициализация структур
- 16.4. Использование структур в функциях
- 16.5. Создание синонимов
- 16.6. Пример: эффективное моделирование тасования и раздачи карт
- 16.7. Поразрядные операции
- 16.8. Битовые поля
- 16.9. Библиотека обработки символов
- 16.10. Функции преобразования строк
- 16.11. Функции поиска из библиотеки обработки строк
- 16.12. Функции работы с памятью из библиотеки обработки строк
- 16.13. Другие функции библиотеки обработки строк

Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Замечания по мобильности • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

16.1. Введение

В этой главе мы сначала приведем дополнительные сведения о структурах, а затем обсудим работу с битами, символами и строками.

Структуры могут содержать переменные разных типов данных в отличие от массивов, которые включают только элементы одного типа. Этот факт и значительная часть того, что мы скажем о структурах в следующих нескольких разделах, применимо в равной степени и к классам. Вообще говоря,

единственное реальное различие между структурами и классами в языке C++ состоит в том, что к элементам структуры по умолчанию доступ является открытый, а к элементам класса — закрытым. Обычно структуры используются для определения записей данных, которые должны быть сохранены в файлах (см. главу 14, «Обработка файлов и ввод-вывод потоков строк»). Указатели и структуры облегчают создание более сложных структур данных, таких как связные списки, очереди, стеки и деревья (см. главу 15 «Структуры данных»). Мы обсудим, как объявлять структуры, инициализировать их и передавать функциям. Затем мы представим быструю программную модель тасования и раздачи карт.

16.2. Описание структур

Рассмотрим следующее описание структуры:

```
struct Card {  
    char *face;  
    char *suit;  
};
```

Ключевое слово **struct** открывает описание структуры **Card**. Идентификатор **Card** является именем *структурь* и используется в C++ для объявления переменных типа *структурь* (в языке С именем типа приведенной выше структуры было бы **struct Card**). Данные (иногда и функции, как в классах), объявленные в фигурных скобках описания структуры, являются *элементами структуры*. Элементы одной структуры должны иметь уникальные имена, но две разные структуры могут включать элементы с одинаковыми именами, не конфликтующими друг с другом. Каждое объявление структуры должно завершаться точкой с запятой.

Типичная ошибка программирования 16.1

Забывают поставить точку с запятой после окончания объявления структуры.

Описание структуры **Card** включает два элемента типа **char ***, а именно: **face** и **suit**. Элементами структуры могут быть переменные базовых типов данных (например, **int**, **float** и т.д.) или составные данные, например, массивы, а также другие структуры. Как мы видели в главе 4, все элементы массива должны быть одного типа. Однако данные-элементы структуры могут быть разного типа. Например, структура **Employee** (служащие) может включать как элементы строки символов для имени и фамилии, элемент типа **int** для возраста служащего, элемент типа **char**, содержащий символы 'М' или 'Ж' для обозначения пола служащего, элемент типа **float** для обозначения почасового оклада и т.д.

Структура не может включать саму себя. Например, в описании структуры **Card** не может быть объявлена переменная типа структуры **Card**. Тем не менее может быть включен указатель на структуру типа **Card**. Структура, содержащая элемент, который является указателем на тот же самый тип структуры, называется *структурой с самоадресацией*. Структуры с самоадресацией использовались в главе 15 для построения разных типов структур со связными данными.

Приведенное объявление структуры не предусматривает выделения какой-либо области памяти; более уместно сказать, что объявление структуры создает новый тип данных, который может использоваться в дальнейшем при объявлении переменных типа структуры. Переменные типа структуры объявляются аналогично переменным других типов. Объявление

```
Card a, deck[52], *c;
```

объявляет а — переменную типа структуры Card, deck — массив с 52 элементами типа Card и с — указатель на структуру типа Card. Переменные заданного типа структуры могут также объявляться с помощью списка разделенных запятыми имен переменных, расположенного между закрывающейся фигурной скобкой и точкой с запятой, завершающей объявление структуры. Например, предыдущее объявление можно включить в описание структуры Card следующим образом :

```
struct Card {
    char *face;
    char *suit;
} a, deck[52], *c;
```

Имя структуры является необязательным. Если объявление структуры не содержит ее имени, то переменные структурного типа могут быть объявлены только в описании структуры, а не с помощью их отдельного объявления.

Хороший стиль программирования 16.1

При создании типа структуры следует предусмотреть и создание имени структуры. Имя структуры удобно использовать в дальнейшем для объявления новых переменных типа данной структуры.

Допустимыми операциями, которые могут быть выполнены со структурами, являются только следующие: операция присваивания одной структуры другой структуре того же типа, операция адреса (&) структуры, операции доступа к элементам структуры (см. главу 6 «Классы и абстрагирование данных») и операция sizeof для определения размера структуры. Как и при использовании классов, большинство операций может быть перегружено, чтобы работать с объектами типа структур.

Типичная ошибка программирования 16.2

Присваивание структуры одного типа структуре другого типа.

Элементы структуры не обязательно сохраняются в последовательных байтах памяти. Иногда при сохранении структур в памяти возникают «дырки», потому что компьютеры могут хранить данные определенных типов только в областях памяти с определенными границами, таких, например, как машинное полуслово, машинное слово или двойное машинное слово. Машинное слово (или просто слово) — стандартная единица (элемент) памяти, используемая для хранения данных в компьютере; обычно это 2 или 4 байта.

Чтобы понять проблемы, возникающие при выравнивании, рассмотрим следующее описание, в котором объявляются переменные типа структуры (фактически объекты) sample1 и sample2:

```
struct Example {
    char c;
    int i;
} sample1, sample2;
```

В компьютере, в котором слово равно 2 байтам, может требоваться, чтобы каждый из элементов структуры **Example** был выровнен относительно границ слова, например, относительно начала слова (это зависит от компьютера). На рис. 16.1 показан пример выравнивания области памяти для объекта типа **Example**, которому присвоен символ 'a' и целое число 97 (показаны значения битов). Если эти элементы хранятся, начиная с границы слова, то в памяти для объектов типа **Example** возникает дырка размером в один байт (байт 1 на рисунке). Значение в этой дырке в один байт не определено. Даже если значения элементов структур **sample1** и **sample2** действительно равны, то не обязательно при сравнении они окажутся равными друг другу, поскольку маловероятно, что неопределенные дырки в один байт содержат одинаковые значения.

Типичная ошибка программирования 16.3

Сравнение структур является синтаксической ошибкой из-за разных требований по выравниванию в разных системах.

Байты	0	1	2	3
	01100001		00000000	01100001

Рис. 16.1. Возможный вариант выравнивания памяти для переменной типа **Example**

Замечание по мобильности 16.1

Поскольку размер данных-элементов определенного типа является машинно-зависимым и выравнивание памяти тоже является машинно- зависимым, то и представление структуры также машинно-зависимо.

16.3. Инициализация структур

Структуры могут быть инициализированы с помощью списка инициализаторов (начальных значений), как массивы. Для инициализации структуры вслед за именем в объявлении переменной ставится знак равенства, а затем в фигурных скобках записывается список инициализаторов, разделенных запятой. Например, объявление

```
Card a = ("Tree", "Hearts");
```

создает переменную **a**, которая является структурой типа **Card** (описанной ранее), и задает элементу **face** начальное значение "Tree", а элементу **suit** начальное значение "Hearts". Если инициализаторов в списке меньше, чем число элементов в структуре, то оставшимся элементам автоматически присваиваются нулевые начальные значения (или значение **NULL**, если элемент является указателем). Переменные типа структуры, объявленные вне описания функции (т.е. внешние) получают начальные значения элементов, равные 0

или `NULL`, если только они не инициализированы явным образом во внешнем объявлении. Переменные типа структуры могут быть также инициализированы операторами присваивания путем присваивания переменной значения другой структуры того же типа (тем самым задаются значения сразу всем элементам) или путем присваивания значений отдельным данным-элементам структуры.

16.4. Использование структур в функциях

Существуют два способа передачи в функции информации о структурах. Вы можете либо передать всю структуру в целом, либо передать отдельные элементы структуры. По умолчанию данные (за исключением отдельных элементов массивов) передаются вызовом по значению. Структуры и их элементы можно также передавать вызовом по ссылке, используя передачу ссылок или указателей.

Чтобы передать структуры вызовом по ссылке, следует передать в функцию адрес переменной структуры. Массивы структуры (аналогично всем другим массивам) автоматически передаются вызовом по ссылке.

В главе 4 мы установили, что массив может быть передан вызовом по значению с помощью структуры. Чтобы передать массив вызовом по значению, создайте структуру (или класс) с массивом в качестве элемента. Поскольку структуры передаются вызовом по значению, то и этот массив будет передан вызовом по значению.

Типичная ошибка программирования 16.4

Ошибочно предполагается, что структуры подобно массивам автоматически передаются вызовом по ссылке и предпринимается попытка изменить в вызываемой программе значения структуры в вызывающей функции.

Совет по повышению эффективности 16.1

Передача структур (особенно больших структур) вызовом по ссылке является более эффективной, чем передача структур вызовом по значению, при которой необходимо копировать всю структуру.

16.5. Создание синонимов

Синонимы (или псевдонимы) для определенных ранее типов данных создаются с помощью ключевое слово `typedef`. Имена типов структур часто переопределяются с помощью `typedef` для применения более коротких или более удобных для чтения имен типов. Например, оператор

```
typedef Card* CardPtr;
```

определяет новое имя типа `CardPtr` как синоним типа `Card *`.

Хороший стиль программирования 16.2

Записывайте имена в предложении `typedef` заглавными буквами, чтобы подчеркнуть, что эти имена являются синонимами других имен типов.

Создание нового имени с помощью `typedef` не приводит к появлению нового типа данных; `typedef` просто создает новое имя уже описанного типа, которое затем может быть использовано в программе в качестве псевдонима существующего имени типа.

С помощью `typedef` могут быть созданы синонимы и для встроенных типов данных. Например, в программе, в которой для представления целого числа требуется 4 байта, можно использовать в одной системе (на одних компьютерах) тип `int`, а в другой — тип `long int`, который соответствует целым с удвоенным объемом памяти. В программах, требующих обеспечить переносимость, можно использовать `typedef` для создания, например, псевдонима `Integer` для представления целых чисел, занимающих 4 байта. Тип `Integer` может быть псевдонимом типа `int` для систем с 4-х байтовыми целыми или псевдонимом типа `long int` в системах с 2-х байтовым представлением целых чисел, в которых значения типа `long int` занимают 4 байта. А затем при написании мобильной программы программист просто объявляет, что все целые переменные, занимающие 4 байта, имеют тип `Integer`.

Замечание по мобильности 16.2

Использование `typedef` позволяет создавать более мобильные программы.

16.6. Пример: эффективное моделирование тасования и раздачи карт

Программа, приведенная на рис. 16.2, основана на моделировании процессов тасования и раздачи карт, рассмотренных в главе 5, «Указатели и строки». Программа представляет колоду карт как массив структур и использует высокоеффективные алгоритмы тасования и раздачи карт. Вывод результатов работы программы показан на рис. 16.3.

В этой программе функция `fillDeck` инициализирует массив структур `Card`, в котором строки символов представляют карты каждой масти упорядоченно от туза до короля. Массив структур `Card` передается функции `shuffle`, которая реализует быстродействующий алгоритм тасования. Функция `shuffle` принимает в качестве аргумента массив из 52 структур `Card`. Функция организует цикл по всем 52 картам (индексы массива принимают значения от 0 до 51). Для каждой карты случайнym образом выбирается число от 0 до 51, определяющее карту, с которой данная карта будет меняться местами. Затем текущая структура `Card` и случайно выбранная структура `Card` меняются местами в массиве. Всего выполняется 52 перестановки за один проход массива, после чего массив структур `Card` оказывается полностью перетасованным. В этом алгоритме не может возникнуть неопределенная отсрочка, которая могла появляться в алгоритме тасования, приведенном в главе 5. Поскольку структуры `Card` менялись местами в массиве, то быстродействующий алгоритм раздачи карт, реализованный функцией `deal`, требует всего одного прохода массива для раздачи уже перетасованной колоды.

Типичная ошибка программирования 16.5

Забывают включить индекс массива при ссылке на отдельные структуры в массиве структур.

```
// Программа тасования и раздачи карт,
// использующая структуры
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

struct Card {
    char *face;
    char *suit;
};

void fillDeck(Card *, char *[], char *[]);
void shuffle(Card *);
void deal(Card *);

main()
{
    Card deck[52];
    char *face[ ] = { "Туз", "Двойка", "Тройка", "Четверка",
                      "Пятерка", "Шестерка", "Семерка",
                      "Восьмерка", "Девятка", "Десятка",
                      "Валет", "Дама", "Король" };
    char *suit[ ] = { "черви", "бубны", "трефы", "пики" };

    srand(time(NULL));      //рандомизация
    fillDeck(deck, face, suit);
    shuffle(deck);
    deal(deck);
    return 0;
}
void fillDeck(Card *wDeck, char *wFace[ ], char *wSuit[ ])
{
    for (int i = 0; i < 52; i++) {
        wDeck[i].face = wFace[i % 13];
        wDeck[i].suit = wSuit[i / 13];
    }
}
void shuffle(Card *wDeck)
{
    for (int i = 0; i <= 51; i++) {
        int j = rand() % 52;
        Card temp = wDeck[i];
        wDeck[i] = wDeck[j];
        wDeck[j] = temp;
    }
}
void deal(Card *wdeck)
{
for (int i = 0; i < 52; i++)
    cout << setiosflags(ios::right) << setw(5) << wdeck[i].face
        << " масти " << setiosflags(ios::left) << setw(8)
        << wdeck[i].suit << ((i + 1) % 2 ? '\t' : '\n' );
}
```

Рис. 16.2. Высокоэффективная программа моделирования тасования и раздачи карт

Восьмерка масти бубны	Туз масти черви
Восьмерка масти трефы	Пятерка масти пики
Семерка масти черви	Двойка масти бубны
Туз масти трефы	Десятка масти бубны
Двойка масти пики	Шестерка масти бубны
Семерка масти пики	Двойка масти трефы
Валет масти треф	Десятка масти пик
Король масти черви	Валет масти бубен
Тройка масти черви	Тройка масти бубен
Тройка масти треф	Девятка масти треф
Десятка масти черви	Двойка масти черви
Десятка масти треф	Семерка масти бубен
Шестерка масти треф	Дама масти пик
Шестерка масти черви	Тройка масти пик
Девятка масти бубен	Туз масти бубен
Валет масти пик	Пятерка масти треф
Король масти бубен	Семерка масти треф
Девятка масти пик	Четверка масти черви
Шестерка масти пик	Восьмерка масти пик
Дама масти бубен	Пятерка масти бубен
Туз масти пик	Девятка масти черви
Король масти треф	Пятерка масти черви
Король масти пик	Четверка масти бубен
Дама масти черви	Восьмерка масти черви
Четверка масти пик	Валет масти черви
Четверка масти треф	Дама масти треф

Рис. 16.3. Выходные данные быстродействующей программной модели тасования и раздачи карт

16.7 Поразрядные операции

Язык C++ предоставляет программистам расширенные возможности для выполнения поразрядных операций, которые необходимы тем, кто хочет спуститься на уровень битов и байтов. Для разработки операционных систем, или программного обеспечения для тестирования аппаратных средств компьютеров, или программного обеспечения, поддерживающего работу в сети, и для многих других видов программного обеспечения необходимо, чтобы программист взаимодействовал непосредственно с аппаратными средствами компьютеров. В этом и нескольких следующих разделах мы обсудим возможности языка C++, позволяющие выполнять поразрядные операции. Мы познакомимся со всеми поразрядными операциями и обсудим, каким образом можно сокращать затраты памяти, используя битовые поля.

Все данные реализуются во внутреннем машинном представлении как последовательности битов. Каждый бит может принимать либо значение 0, либо значение 1. В большинстве систем последовательность из 8 битов образует один байт, который является стандартной единицей для хранения переменной типа `char`. Другие типы данных хранятся в большем числе байтов. Поразрядные (побитовые) операции используются для манипуляций с битами целочисленных операндов (`char`, `short`, `int` и `long`, причем обоих видов: `signed` и `unsigned`). Но обычно поразрядные операции применяются к целым без знака (`unsigned`).

Замечание по мобильности 16.3

Поразрядные манипуляции с данными являются машинно-зависимыми.

Отметим, что обсуждение поразрядных операций в этом разделе сопровождается показом двоичного представления целых operandов. Для детального изучения двоичной системы счислений (с основанием 2) смотрите приложение Г «Системы счисления». Отметим также, что программы разделов 16.7 и 16.8 были проверены на РС-совместимых компьютерах с использованием Borland C++. Эта система использует 16-ти битовые (двухбайтовые) целые. Из-за машинно-зависимой природы поразрядных операций эти программы могут не работать в вашей системе.

Существуют следующие поразрядные операции: *поразрядное И (&)*, *поразрядное ИЛИ (|)*, *поразрядное исключающее ИЛИ (^)*, *сдвиг влево (<<)*, *сдвиг вправо (>>)*; *поразрядное НЕ (дополнение, отрицание) (~)*. (Обратите внимание, что мы уже использовали операции &, << и >> для других целей. Это классический пример перегрузки операций). Операции поразрядное И, поразрядное ИЛИ и поразрядное исключающее ИЛИ выполняют поразрядное сравнение двух своих operandов. В результате проведения операции поразрядное И bit устанавливается в 1, если соответствующий bit в обоих operandах равен 1. В результате операции поразрядное ИЛИ bit устанавливается в 1, если хотя бы в одном из operandов он равен 1. В результате выполнения операции поразрядное исключающее ИЛИ bit устанавливается в 1, если соответствующий bit равен 1 в одном и только в одном операнде. Операция сдвига влево сдвигает биты своего левого операнда влево на количество битов, заданное правым operandом. Операция сдвига вправо сдвигает биты левого операнда вправо на количество битов, заданное правым operandом. Операция поразрядное НЕ устанавливает в своем операнде все биты со значениями 0 в 1, а все биты со значениями 1 в 0. Детальное обсуждение каждой поразрядной операции будет проведено ниже на конкретных примерах. Краткое описание поразрядных операций приводится на рис. 16.4.

Операция		Описание
&	поразрядное И	Биты результата устанавливаются в 1, если соответствующие биты обоих operandов равны 1.
	поразрядное ИЛИ	Биты результата устанавливаются в 1, если соответствующий bit по крайней мере одного операнда равен 1.
^	поразрядное исключающее ИЛИ	Биты результата устанавливаются в 1, если соответствующий bit одного и только одного operandана равен 1.
<<	сдвиг влево	Сдвигает биты первого operandана влево на количество битов, заданное вторым operandом. При выполнении операции сдвига правые освобождающиеся биты заполняются 0.
>>	сдвиг вправо	Сдвигает биты первого operandана вправо на количество битов, заданное вторым operandом. Метод заполнения левых освобождающихся битов является машинно- зависимым.
~	поразрядное НЕ (дополнение до единицы)	Все биты со значениями 0 устанавливаются в 1, а все биты со значениями 1 устанавливаются в 0.

Рис. 16.4. Поразрядные операции

При использовании поразрядных операций для большей наглядности полезно печатать двоичное представление значений. Программа, приведенная на рис. 16.5, печатает целое значение типа `unsigned` в двоичном представлении группами по восемь битов каждое. Функция `displayBits` применяет операцию поразрядное И к переменным `value` и `displayMask`. Часто операция поразрядное И используется с операндом, называемый *маской* — целым значением, в котором определенные биты установлены в 1. Маски используются для того, чтобы выделить в анализируемом значении указанные биты и сделать невидимыми остальные. В функции `displayBits` маске `displayMask` присвоено `1 << 15` (`10000000 00000000`). Операция сдвига влево смещает значение 1 влево и заполняет правые освободившиеся биты нулями.

```
// Печать двоичного представления целого без знака
#include <iostream.h>
#include <iomanip.h>

main()
{
    unsigned x;
    void displayBits (unsigned);

    cout << "Введите целое без знака: ";
    cin >> x;
    displayBits(x);
    return 0;
}

void displayBits(unsigned value)
{
    unsigned c, displayMask = 1 << 15;
    cout << setw(7) << value << " = ";

    for (c = 1; c <= 16; c++) {
        cout << (value & displayMask ? '1': '0');
        value <<= 1;
        if (c % 8 == 0)
            cout << ' ';
    }

    cout << endl;
}
```

```
Введите целое без знака: 65000
65000 = 11111101 11101000
```

Рис. 16.5. Печать двоичного представления целого без знака (часть 2 из 2)

Оператор

```
cout << (value & displayMask ? '1' : '0');
```

определяет, надо ли печатать 1 или 0 для текущего самого левого бита переменной `value`. Предположим, что переменная `value` содержит 65000 (`11111101 11101000`). Если переменные `value` и `displayMask` соединяются

операцией поразрядное И (`&`), то все биты за исключением бита самого старшего разряда переменной `value` маскируются (скрыты), поскольку результат операции И, примененной к любому биту и биту, содержащему 0, дает 0. Если бит в самом старшем разряде `value` установлен в 1, то результат операции `value & displayMask` равен 1 и печатается 1; в противном случае определяется и печатается 0. Затем переменная `value` сдвигается влево на один бит с помощью выражения `value <<= 1` (это равносильно выражению `value = value << 1`). Эти шаги повторяются для каждого бита в переменной `value` типа `unsigned`. На рис. 16.6 приведены результаты действия операции поразрядное И на два бита.

Бит 1	Бит 2	Бит 1 & Бит 2
0	0	0
1	0	0
0	1	0
1	1	1

Рис. 16.6. Результат объединения двух битов операцией поразрядное И

Типичная ошибка программирования 16.6

Использование операции логическое И (`&&`) вместо операции поразрядное И (`&`) и наоборот.

Программа на рис. 16.7 показывает использование операций поразрядное И, поразрядное ИЛИ, поразрядное исключающее ИЛИ и поразрядное НЕ. Программа использует функцию `displayBits` для печати целого значения типа `unsigned`. Вывод программы показан на рис. 16.8.

```
// Использование операций поразрядное И, поразрядное ИЛИ,
// поразрядное исключающее ИЛИ и поразрядное НЕ.
#include <iostream.h>
#include <iomanip.h>

void displayBits(unsigned);

main()
{
    unsigned number1, number2, mask, setBits;
    number1 = 65535;
    mask = 1;
    cout << "Результат объединения значений" << endl;
    displayBits(number1);
    displayBits(mask);
    cout << "операцией поразрядное И & равен"
        << endl;
    displayBits(number1 & mask);
```

Рис. 16.7. Использование операций поразрядное И, поразрядное ИЛИ, поразрядное исключающее ИЛИ и поразрядное НЕ (часть 1 из 2)

```
number1 = 15;
setBits = 241;
cout << endl << "Результат объединения значений" << endl;
displayBits(number1);
displayBits(setBits);
cout << "операцией поразрядное ИЛИ | равен" << endl;
displayBits(number1 | setBits);

number1 = 139;
number2 = 199;
cout << endl << "Результат объединения значений" << endl;
displayBits(number1);
displayBits(number2);
cout << "операцией поразрядное исключающее ИЛИ ^ равен"
    << endl;
displayBits(number1 ^ number2);

number1 = 21845;
cout << endl << "Результат применения операции поразрядное НЕ "
    << "~ к значению" << endl;
displayBits(number1);
cout << "равен" << endl;
displayBits(~number1);

return 0;
}

void displayBits(unsigned value)
{
    unsigned c, displayMask = 1 << 15;
    cout << setw(7) << value << " = ";

    for (c = 1; c <= 16; c++) {
        cout << (value & displayMask ? '1' : '0');
        value <<= 1;
        if (c % 8 == 0)
            cout << ' ';
    }

    cout << endl;
}
```

Рис. 16.7. Использование операций поразрядное И, поразрядное ИЛИ, поразрядное исключающее ИЛИ и поразрядное НЕ (часть 2 из 2)

В программе на рис. 16.7 переменной **mask** присваивается значение **1** (00000000 00000001), а переменной **number1** присваивается значение **65535** (11111111 11111111). Затем значения переменных **mask** и **number1** соединяются операцией поразрядное И (&) в выражении **mask & number1** и в результате получается значение 00000000 00000001. Все биты, за исключением бита самого младшего разряда в переменной **number1** маскированы (скрыты) применением операции поразрядное И с маской **mask**.

Операция поразрядное ИЛИ используется для установки в операнде указанных битов в 1. В программе на рис. 17.7 переменной **number1** присваивается значение **15** (00000000 00001111), а переменной **setBits** присваивается

значение 241 (00000000 11110001). Когда значения переменных `number1` и `setBits` объединяются операцией поразрядное ИЛИ в выражении `number1 | setBits`, результат равен 255 (00000000 11111111). На рис. 16.9 показаны результаты выполнения операции поразрядное ИЛИ для двух битов.

```

Результат объединения значений
65535 = 11111111 11111111
1 = 00000000 00000001
операцией поразрядное И & равен
1 = 00000000 00000001

Результат объединения значений
15 = 00000000 00001111
241 = 00000000 11110001
операцией поразрядное ИЛИ | равен
255 = 00000000 11111111

Результат объединения значений
139 = 00000000 10001011
199 = 00000000 11000111
операцией поразрядное исключающее ИЛИ ^ равен
76 = 00000000 01001100

Результат применения операции поразрядное НЕ ~ к значению
21845 = 01010101 01010101
равен
43690 = 10101010 10101010

```

Рис. 16.8. Выходные данные для программы, приведенной на рис. 16.7

Типичная ошибка программирования 16.7

Использование операции логическое ИЛИ (`||`) вместо операции поразрядное ИЛИ (`|`) и наоборот.

Бит 1	Бит 2	Бит 1 Бит 2
0	0	0
1	0	1
0	1	1
1	1	1

Рис. 16.9. Результаты применения к двум битам операции поразрядное ИЛИ

Операция поразрядное исключающее ИЛИ (`^`) устанавливает бит результата в 1, если в одном и только в одном из двух операндов соответствующий бит равен 1. В программе на рис. 16.7 переменным `number1` и `number2` присваиваются соответственно значения 139 (00000000 10001011) и 199 (00000000 11000111). Если эти переменные соединяются операцией поразрядное исключающее ИЛИ в выражении `number1 ^ number2`, то результат равен 00000000 01001100. На рис. 16.10 показаны результаты выполнения операции поразрядное исключающее ИЛИ для двух битов.

Операция *поразрядное НЕ* (*поразрядное дополнение*) (`~`) устанавливает биты со значением **1** в **0**, а биты со значением **0** в **1**; это называется также *дополнением значения до единицы*. В программе на рис. 16.7 переменной `number1` присваивается значение **21845** (01010101 01010101). Затем вычисляется выражение `~number1`, результат которого равен (10101010 10101010).

Бит 1	Бит 2	Бит 1 \wedge Бит 2
0	0	0
1	0	1
0	1	1
1	1	0

Рис. 18.10. Результаты применения к двум битам операции поразрядное исключающее ИЛИ

Программа, приведенная на рис. 16.11, демонстрирует применение операций сдвига влево (`<<`) и вправо (`>>`). Функция `displayBits` используется для печати целых значений типа `unsigned`.

```
// Использование операций поразрядного сдвига
#include <iostream.h>
#include <iomanip.h>
void displayBits (unsigned);
main()
{
    unsigned number1 = 960;

    cout << "Результат сдвига влево" << endl;
    displayBits(number1);
    cout << "на 8 битов с использованием операции сдвига влево <<:"
        << endl;
    displayBits(number1 << 8);
    cout << endl << "Результат сдвига вправо" << endl;
    displayBits(number1);
    cout << "на 8 битов с использованием операции сдвига вправо >>:"
        << endl;
    displayBits(number1 >> 8);
    return 0;
}

void displayBits(unsigned value)
{
    unsigned c, displayMask = 1 << 15;

    cout << setw(7) << value << " = ";

    for (c = 1; c <= 16; c++) {
        cout << (value & displayMask ? '1' : '0');
        value <<= 1;
        if (c % 8 == 0)
            cout << ' ';
    }
    cout << endl;
}
```

Рис. 16.11. Использование операций поразрядного сдвига (часть 1 из 2)

```

Результат сдвига влево
960 = 00000011 11000000
на 8 битов с использованием операции сдвига влево <<:
49152 = 11000000 00000000

Результат сдвига вправо
960 = 00000011 11000000
на 8 битов с использованием операции сдвига вправо >>:
3 = 00000000 00000011

```

Рис. 16.11. Использование операций поразрядного сдвига (часть 2 из 2)

Операция сдвига влево (`<<`) сдвигает биты своего левого операнда влево на количество битов, заданное правым операндом. Биты, расположенные справа, освобождаются и заполняются нулями; левые разряды, сдвинутые за пределы операнда, теряются. В программе, приведенной на рис. 16.11, переменной `number1` присваивается значение **960** (00000011 11000000). Результатом операции сдвига влево значения переменной `number1` на 8 битов с помощью выражения `number1 << 8` является значение **49152** (11000000 00000000).

Операция сдвига вправо (`>>`) сдвигает биты своего левого операнда вправо на заданное правым операндом количество битов. Выполнение операции сдвига вправо целого значения типа `unsigned` приводит к освобождению левых битов, которые заполняются нулями. В программе, приведенной на рис. 16.11, результатом операции сдвига вправо значения переменной `number1` на 8 битов с помощью выражения `number1 >> 8` является значение **3** (00000000 00000011).

Типичная ошибка программирования 16.8

Результат сдвига какого-либо значения не определен, если правый операнд является отрицательным или если правый операнд больше числа битов в левом операнде.

Замечание по мобильности 16.4

Результат сдвига вправо целого значения со знаком (типа `signed`) является машинно-зависимым. Некоторые компьютеры заполняют освобожденные биты нулями, а другие компьютеры используют бит знака.

Каждая поразрядная операция (за исключением операции поразрядного НЕ) имеет соответствующие операции присваивания. Эти *операции поразрядного присваивания* показаны на рис. 16.12 и выполняются аналогично арифметическим операциям присваивания, которые рассмотрены в главе 2.

Операции поразрядного присваивания	
<code>&=</code>	Операция присваивания поразрядного И
<code> =</code>	Операция присваивания поразрядного ИЛИ
<code>^=</code>	Операция присваивания поразрядного исключающего ИЛИ
<code><<=</code>	Операция присваивания сдвига влево
<code>>>=</code>	Операция присваивания сдвига вправо

Рис. 16.12. Операции поразрядного присваивания

На рис. 16.13 показаны приоритет и ассоциативность различных операций языка C++, рассмотренных нами к настоящему моменту. Эти операции приведены в порядке убывания их приоритета.

Операции	Ассоциативность	Тип
:: (унарная)	справа налево	разрешение области действия
:: (бинарная)	слева направо	
() [] . ->	слева направо	различный
+ ~ + -- ! (тип) & * ~ sizeof	справа налево	унарные
* / %	слева направо	мультипликативные
+ -	слева направо	аддитивные
<< >>	слева направо	сдвиг
< <= > >=	слева направо	отношение
== !=	слева направо	проверка на равенство
&	слева направо	поразрядное И
^	слева направо	поразрядное НЕ
	слева направо	поразрядное ИЛИ
&&	слева направо	логическое И
	слева направо	логическое ИЛИ
?:	справа налево	условная
= += -= *= /= %= = &= ^= <<= >>=	справа налево	присваивание
,	слева направо	запятая (последование)

Рис. 16.13. Приоритет и ассоциативность операций

16.8. Битовые поля

Язык C++ предоставляет возможность задавать количество битов, в которых хранятся элементы типов `unsigned` или `int` класса или структуры (или элементы объединения — см. главу 18, «Другие темы»). Такие элементы называются **битовыми полями**. Битовые поля позволяют рационально использовать память с помощью хранения данных в минимально требуемом количестве битов. Элементы битовые поля **должны** быть объявлены как тип `int` или `unsigned`.

Совет по повышению эффективности 16.2

Битовые поля способствуют рациональному использованию памяти.

Рассмотрим следующее описание структуры:

```
struct BitCard {
    unsigned face : 4;
    unsigned suit : 2;
    unsigned color : 1;
};
```

Это описание включает три битовых поля типа `unsigned`: `face`, `suit` и `color`, используемых для представления карт колоды, состоящей из 52 карт. При объявлении битового поля вслед за указанием типа элемента `unsigned` или `int` ставится двоеточие (`:`) и пишется целочисленная константа, задающая ширину поля (т.е. число битов, в которых хранится этот элемент). Ширина поля должна быть целочисленной константой в диапазоне между 0 и заданным общим числом битов, используемых для хранения целого значения типа `int` в вашей системе. Приведенные ниже примеры были проверены на компьютере с двухбайтовым (16 битов) представлением целого значения.

Приведенное выше описание структуры показывает, что для хранения элемента `face` выделено 4 бита, для хранения элемента `suit` — 2 бита и для хранения элемента `color` — 1 бит. Количество битов определяется ожидаемым диапазоном значений для каждого элемента структуры. Элемент `face` хранит значения от 0 (Туз) до 12 (Король) в области памяти размером 4 бита (4 бита, выделенные для элемента `face`, могут хранить значения от 0 до 15). Элемент `suit` может хранить значения от 0 до 3 (0 = бубны, 1 = черви, 2 = трефы и 3 = пики). Область памяти размером 2 бита, выделенная для элемента `suit`, может хранить значения от 0 до 3. И, наконец, элемент `color` хранит либо 0 (Красный), либо 1 (Черный). Область памяти размером 1 бит, выделенная для элемента `color`, может хранить либо 0, либо 1.

Программа, приведенная на рис. 16.14 (выходные данные показаны на рис. 16.15), создает массив `deck`, содержащий 52 структуры `struct` типа `bitCard`. Функция `fillDeck` вставляет 52 карты в массив `deck`, а функция `deal` выводит на печать 52 карты. Заметим, что элементы битового поля структур доступны точно так же, как и другие элементы структуры. Элемент `color` включается как средство отображения цвета карты в системе, которая, позволяет отображать цвет.

```
// Пример использования битовых полей для хранения колоды карт
#include <iostream.h>
#include <iomanip.h>

struct BitCard {
    unsigned face : 4;
    unsigned suit: 2;
    unsigned color : 1;
};

void fillDeck(BitCard *);
void deal(BitCard *);

main()
{
    BitCard deck [52];
```

Рис. 16.14. Использование битовых полей для хранения колоды карт (часть 1 из 2)

```

        fillDeck(deck);
        deal(deck);
        return 0;
    }

    void fillDeck(BitCard *wDeck)
    {
        for (int i = 0; i <= 51; i++) {
            wDeck[i].face = i % 13;
            wDeck[i].suit = i / 13;
            wDeck[i].color = i / 26;
        }
    }

    // Выходные данные для карт представлены в виде двух колонок.
    // Карты 0-25 расположены в первой колонке с индексом k1.
    // Карты 26-51 расположены во второй колонке с индексом k2.
    void deal(BitCard *wDeck)
    {
        for (int k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++) {
            cout << "Карта:" << setw(3) << wDeck[k1].face << " Масть:"
                << setw(2) << wDeck[k1].suit << " Цвет:" << setw(2)
                << wDeck[k1].color <<
            cout << "Карта:" << setw(3) << wDeck[k2].face << " Масть:"
                << setw(2) << wDeck[k2].suit << " Цвет" << setw(2)
                << wDeck[k2].color << endl;
        }
    }
}

```

Рис. 16.14. Использование битовых полей для хранения колоды карт (часть 2 из 2)

Карта:	0	Масть:	0	Цвет:	0	Карта:	0	Масть:	2	Цвет:	1
Карта:	1	Масть:	0	Цвет:	0	Карта:	1	Масть:	2	Цвет:	1
Карта:	2	Масть:	0	Цвет:	0	Карта:	2	Масть:	2	Цвет:	1
Карта:	3	Масть:	0	Цвет:	0	Карта:	3	Масть:	2	Цвет:	1
Карта:	4	Масть:	0	Цвет:	0	Карта:	4	Масть:	2	Цвет:	1
Карта:	5	Масть:	0	Цвет:	0	Карта:	5	Масть:	2	Цвет:	1
Карта:	6	Масть:	0	Цвет:	0	Карта:	6	Масть:	2	Цвет:	1
Карта:	7	Масть:	0	Цвет:	0	Карта:	7	Масть:	2	Цвет:	1
Карта:	8	Масть:	0	Цвет:	0	Карта:	8	Масть:	2	Цвет:	1
Карта:	9	Масть:	0	Цвет:	0	Карта:	9	Масть:	2	Цвет:	1
Карта:	10	Масть:	0	Цвет:	0	Карта:	10	Масть:	2	Цвет:	1
Карта:	11	Масть:	0	Цвет:	0	Карта:	11	Масть:	2	Цвет:	1
Карта:	12	Масть:	0	Цвет:	0	Карта:	12	Масть:	2	Цвет:	1
Карта:	0	Масть:	1	Цвет:	0	Карта:	0	Масть:	3	Цвет:	1
Карта:	1	Масть:	1	Цвет:	0	Карта:	1	Масть:	3	Цвет:	1
Карта:	2	Масть:	1	Цвет:	0	Карта:	2	Масть:	3	Цвет:	1
Карта:	3	Масть:	1	Цвет:	0	Карта:	3	Масть:	3	Цвет:	1
Карта:	4	Масть:	1	Цвет:	0	Карта:	4	Масть:	3	Цвет:	1
Карта:	5	Масть:	1	Цвет:	0	Карта:	5	Масть:	3	Цвет:	1
Карта:	6	Масть:	1	Цвет:	0	Карта:	6	Масть:	3	Цвет:	1
Карта:	7	Масть:	1	Цвет:	0	Карта:	7	Масть:	3	Цвет:	1
Карта:	8	Масть:	1	Цвет:	0	Карта:	8	Масть:	3	Цвет:	1
Карта:	9	Масть:	1	Цвет:	0	Карта:	9	Масть:	3	Цвет:	1
Карта:	10	Масть:	1	Цвет:	0	Карта:	10	Масть:	3	Цвет:	1
Карта:	11	Масть:	1	Цвет:	0	Карта:	11	Масть:	3	Цвет:	1
Карта:	12	Масть:	1	Цвет:	0	Карта:	12	Масть:	3	Цвет:	1

Рис. 16.15. Выходные данные, полученные в результате выполнения программы, приведенной на рис. 16.14

Можно задавать *неименованное битовое поле*; в этом случае поле используется в структуре как *заполнение*. Например, описание структуры

```
struct Example {
    unsigned a : 13;
    unsigned : 3;
    unsigned b : 4;
};
```

использует неименованное 3-х битовое поле как заполнение: ничто не может храниться в этих трех битах. Элемент **b** (в нашем компьютере машинное слово занимает 2 байта) хранится в другом элементе памяти.

Неименованное битовое поле нулевой ширины используется для выравнивания следующего битового поля по границе нового элемента памяти. Например, описание структуры

```
struct Example {
    unsigned a : 13;
    unsigned : 0;
    unsigned b : 4;
};
```

использует неименованное поле нулевой ширины, чтобы пропустить оставшиеся биты (все, сколько их есть) в том элементе памяти, в котором хранится элемент **a**, и выровнять элемент **b** по границе следующего элемента памяти.

Замечание по мобильности 16.5

Манипуляции с битовыми полями являются машинно-зависимыми. Например, в некоторых компьютерах битовые поля могут пересекать границы машинного слова, тогда как в других компьютерах это недопустимо.

Типичная ошибка программирования 16.9

Попытка осуществить доступ к отдельным битам битового поля как если бы они были элементами массива. Битовые поля не являются «массивами битов».

Типичная ошибка программирования 16.10

Попытка получить адрес битового поля (операция **&** не может применяться к битовым полям, поскольку они не имеют адресов).

Совет по повышению эффективности 16.3

Хотя битовые поля сокращают требования к памяти, их использование может привести к тому, что компилятор будет генерировать машинный код, который выполняется с низкой скоростью. Это происходит вследствие того, что приходится использовать дополнительные операции машинного языка для получения доступа к отдельным частям адресуемых элементов памяти. Это является одним из множества примеров необходимости компромисса между требованиями эффективности по памяти и по времени выполнения программы.

16.9. Библиотека обработки символов

Большинство данных вводится в компьютеры как символы, включая буквы, цифры и различные специальные символы. В этом разделе мы обсудим возможности языка C++ по исследованию отдельных символов и манипули-

рованию ими. Позже в этой главе мы продолжим рассмотрение операций с символьными строками, начатое в главе 5.

Библиотека обработки символов включает несколько функций, которые выполняют полезные операции по обработке символьных данных. Каждая из этих функций принимает символ типа `int` или `EOF` в качестве аргумента. Часто операции с символами выполняются как операции с целыми значениями. Вспомним, что `EOF` обычно имеет значение `-1` и архитектура некоторых аппаратных средств не позволяет сохранять отрицательные значения в переменных типа `char`. Поэтому функции обработки символов манипулируют с символами как с целыми. На рис. 16.16 представлены функции библиотеки обработки символов. При применении функций из библиотеки обработки символов необходимо убедиться, что в программу включен заголовочный файл `<ctype.h>`.

Прототип	Описание функции
<code>int isdigit(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является цифрой, в противном случае возвращает <code>false</code> (0).
<code>int isalpha(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является буквой, в противном случае возвращает <code>false</code> (0).
<code>int isalnum(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является цифрой или буквой, в противном случае возвращает <code>false</code> (0).
<code>int isxdigit(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является шестнадцатеричной цифрой, в противном случае возвращает <code>false</code> (0). (См. приложение Г «Системы счисления» для более детального изучения двоичных, восьмеричных, десятичных и шестнадцатеричных чисел).
<code>int islower(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является строчной буквой (в нижнем регистре), в противном случае возвращает <code>false</code> (0). Функция работает только для латинских букв.
<code>int isupper(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является прописной буквой (в верхнем регистре), в противном случае возвращает <code>false</code> (0). Функция работает только для латинских букв.
<code>int tolower(int c)</code>	Если элемент <code>c</code> является прописной буквой, то <code>tolower</code> возвращает элемент <code>c</code> как строчную букву. В противном случае <code>tolower</code> возвращает аргумент без изменения. Функция работает только для латинских букв.
<code>int toupper(int c)</code>	Если элемент <code>c</code> является строчной буквой, то <code>toupper</code> возвращает элемент <code>c</code> как прописную букву. В противном случае <code>toupper</code> возвращает аргумент без изменения. Функция работает только для латинских букв.
<code>int isspace(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является символом разделителем – символом перехода на новую строку ('\n'), пробелом (' '), символом перехода на новую страницу ('\f'), символом возврата каретки ('\r'), символом горизонтальной табуляции ('\t') или символом вертикальной табуляции ('\v'). В противном случае возвращается <code>false</code> (0).
<code>int iscntrl(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является управляющим символом, в противном случае, возвращает <code>false</code> (0).
<code>int ispunct(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является символом пунктуации, отличным от пробела, цифры или буквы; в противном случае возвращает <code>false</code> (0).
<code>int isprint(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является любым печатным символом, включая пробел (' '); в противном случае возвращает <code>false</code> (0).
<code>int isgraph(int c)</code>	Возвращает <code>true</code> , если элемент <code>c</code> является печатным символом, отличным от пробела; в противном случае возвращает <code>false</code> (0).

Рис. 16.16. Краткое описание функций из библиотеки обработки символов

Программа, приведенная на рис. 16.17, демонстрирует применение функций *isdigit*, *isalpha*, *isalnum* и *isxdigit*. Функция *isdigit* определяет, является ли аргумент цифрой (0–9). Функция *isalpha* определяет, является ли аргумент прописной буквой (A–Z) или строчной буквой (a–z). Функция *isalnum* определяет, является ли аргумент прописной буквой, строчной буквой или цифрой (A–Z, a–z, 0–9).

```
// Использование функций isdigit, isalpha, isalnum и isxdigit

#include <iostream.h>
#include <ctype.h>

main()
{
    cout << "Согласно isdigit:" << endl
        << (isdigit('8') ? "8 является" : "8 не является")
        << " цифрой" << endl
        << (isdigit('#') ? "# является" : "# не является")
        << " цифрой" << endl;

    cout << endl << "Согласно isalpha:" << endl
        << (isalpha('A') ? "A является" : "A не является")
        << " буквой" << endl
        << (isalpha('b') ? "b является" : "b не является")
        << " буквой" << endl
        << (isalpha('&') ? "& является" : "& не является")
        << " буквой" << endl
        << (isalpha('4') ? "4 является" : "4 не является")
        << " буквой" << endl;

    cout << endl << "Согласно isalnum:" << endl
        << (isalnum('A') ? "A является" : "A не является")
        << " цифрой или буквой" << endl
        << (isalnum('8') ? "8 является" : "8 не является")
        << " цифрой или буквой" << endl
        << (isalnum('#') ? "# является" : "# не является")
        << " цифрой или буквой" << endl;

    cout << endl << "Согласно isxdigit:" << endl
        << (isxdigit('F') ? "F является" : "F не является")
        << " шестнадцатеричной цифрой" << endl
        << (isxdigit('J') ? "J является" : "J не является")
        << " шестнадцатеричной цифрой" << endl
        << (isxdigit('7') ? "7 является" : "7 не является")
        << " шестнадцатеричной цифрой" << endl
        << (isxdigit('$') ? "$ является" : "$ не является")
        << " шестнадцатеричной цифрой" << endl
        << (isxdigit('f') ? "f является" : "f не является")
        << " шестнадцатеричной цифрой" << endl;

    return 0;
}
```

Рис. 16.17. Использование функций *isdigit*, *isalpha*, *isalnum* и *isxdigit* (часть 1 из 2)

```
Согласно isdigit:
8 является цифрой
# не является цифрой
```

```
Согласно isalpha:
A является буквой
b является буквой
& не является буквой
4 не является буквой
```

```
Согласно isalnum:
A является цифрой или буквой
8 является цифрой или буквой
# не является цифрой или буквой
```

```
Согласно isxdigit:
F является шестнадцатеричной цифрой
J не является шестнадцатеричной цифрой
7 является шестнадцатеричной цифрой
$ не является шестнадцатеричной цифрой
f является шестнадцатеричной цифрой
```

Рис. 16.17. Использование функций **isdigit**, **isalpha**, **isalnum** и **isxdigit** (часть 2 из 2)

Программа, приведенная на рис. 16.17, использует условную операцию **(?:)** с каждой функцией, которая определяет, какую из двух строк: «является» или «не является» напечатать после определения типа соответствующего символа. Например, выражение

```
isdigit('8') ? "8 является" : "8 не является"
```

определяет, что если '8' является цифрой, т.е. функция **isdigit** возвращает **true** (ненулевое значение), то печатается строка "8 является", а если '8' не является цифрой, т.е. функция **isdigit** возвращает **0**, то печатается строка "8 не является".

Программа, приведенная на рис. 16.18, демонстрирует функции **islower**, **isupper**, **tolower** и **toupper**. Функция **islower** определяет, является ли ее аргумент строчной буквой (a–z). Функция **isupper** определяет, является ли ее аргумент прописной буквой (A–Z). Функция **tolower** преобразует прописную букву в строчную и возвращает прописную букву. Если аргумент не является прописной буквой, то функция **tolower** возвращает аргумент без изменения. Все эти функции работают только для латинских букв.

```
// Использование функций islower, isupper, tolower, toupper
#include <iostream.h>
#include <ctype.h>

main()
{
    cout << "Согласно islower:" << endl
        << (islower('p') ? "p является" : "p не является")
        << " буквой в нижнем регистре" << endl
        << (islower('P') ? "P является" : "P не является")
        << " буквой в нижнем регистре" << endl
        << (islower('5') ? "5 является" : "5 не является")
```

Рис. 16.18. Использование функций **islower**, **isupper**, **tolower** и **toupper** (часть 1 из 2)

```

    << " буквой в нижнем регистре" << endl
    << (islower('!') ? "! является" : "! не является")
    << " буквой в нижнем регистре" << endl;
cout << endl << "Согласно isupper:" << endl
    << (isupper('D') ? "D является" : "D не является")
    << " буквой в верхнем регистре" << endl
    << (isupper('d') ? "d является" : "d не является")
    << " буквой в верхнем регистре" << endl
    << (isupper('8') ? "8 является" : "8 не является")
    << " буквой в верхнем регистре" << endl
    << (isupper('$') ? "$ является" : "$ не является")
    << " буквой в верхнем регистре" << endl;

cout << endl << "и после перевода в верхний регистр равно "
    << (char) toupper('u') << endl
    << "7 после перевода в верхний регистр равно "
    << (char) toupper('7') << endl
    << "$ после перевода в верхний регистр равно "
    << (char) toupper('$') << endl
    << "L после перевода в нижний регистр равно "
    << (char) tolower('L') << endl;
return 0;
}

```

Согласно islower:

р является буквой в нижнем регистре
 Р не является буквой в нижнем регистре
 5 не является буквой в нижнем регистре
 ! не является буквой в нижнем регистре

Согласно isupper:

D является буквой в верхнем регистре
 d не является буквой в верхнем регистре
 8 не является буквой в верхнем регистре
 \$ не является буквой в верхнем регистре

и после перевода в верхний регистр равно U
 7 после перевода в верхний регистр равно 7
 \$ после перевода в верхний регистр равно \$
 L после перевода в нижний регистр равно l

Рис. 16.18. Использование функций `islower`, `isupper`, `tolower` и `toupper` (часть 2 из 2)

Рис. 16.19 демонстрирует функции `isspace`, `iscntrl`, `ispunct`, `isprint` и `isgraph`. Функция `isspace` определяет, является ли аргумент символом разделителем, таким, как пробел (' '), символ перевода страницы ('\f'), символ перехода на новую строку ('\n'), символ возврата каретки `return` ('\r'), символ горизонтальной табуляции ('\t') или символ вертикальной табуляции ('\v'). Функция `iscntrl` определяет, является ли ее аргумент управляющим символом, таким, как символ горизонтальной или вертикальной табуляции, символ перевода страницы, символ звукового сигнала ('\a'), символ `backspace` — возврат на один символ ('\b'), символ возврата каретки или перехода на новую строку. Функция `ispunct` определяет, является ли ее аргумент печатным символом, отличным от пробела, цифры или буквы, таким, как

\$, #, (,), [,], {, }, ;, :, % и т.д. Функция **isprint** определяет, является ли ее аргумент печатным символом, который может быть отображен на экране (включая символ пробела). Функция **isgraph** осуществляет ту же проверку, что и **isprint**, исключая пробел.

```
// Использование функций isspace, iscntrl, ispunct, isprint, isgraph
#include <iostream.h>
#include <ctype.h>

main()
{
    cout << "Согласно isspace:" << endl
        << "Новая строка "
        << (isspace('\n') ? "является" : "не является")
        << " символом разделителем" << endl
        << "Горизонтальная табуляция "
        << (isspace('\t') ? "является" : "не является")
        << " символом разделителем" << endl
        << (isspace('%') ? "% является" : "% не является")
        << " символом разделителем" << endl;

    cout << endl << "Согласно iscntrl:" << endl
        << "Новая строка "
        << (iscntrl('\n') ? "является" : "не является")
        << " управляемым символом" << endl
        << (iscntrl('$') ? "$ является" : "$ не является")
        << " управляемым символом" << endl;

    cout << endl << "Согласно ispunct:" << endl
        << (ispunct(';) ? ";" является" : ";" не является")
        << " символом пунктуации" << endl
        << (ispunct('Y') ? "Y является" : "Y не является")
        << " символом пунктуации" << endl
        << (ispunct('#') ? "#" является" : "# не является")
        << " символом пунктуации" << endl;

    cout << endl << "Согласно isprint:" << endl
        << (isprint('$') ? "$ является" : "$ не является")
        << " печатным символом" << endl
        << "Звуковой сигнал "
        << (isprint('\a') ? "является" : "не является")
        << " печатным символом" << endl;

    cout << endl << "Согласно isgraph:" << endl
        << (isgraph('Q') ? "Q является" : "Q не является")
        << " печатным символом, отличным от пробела" << endl
        << "Пробел " << (isgraph(' ') ? "является" :
        "не является")
        << " печатным символом, отличным от пробела" << endl;

    return 0;
}
```

Рис. 16.19. Использование функций **isspace**, **iscntrl**, **ispunct**, **isprint** и **isgraph** (часть 1 из 2)

Согласно `isspace`:

Новая строка является символом разделителем
Горизонтальная табуляция является символом разделителем
% не является символом разделителем

Согласно `iscntrl`:

Новая строка является управляемым символом
\$ не является управляемым символом

Согласно `ispunct`:

; является символом пунктуации
Y не является символом пунктуации
является символом пунктуации

Согласно `isprint`:

\$ является печатным символом
Звуковой сигнал не является печатным символом

Согласно `isgraph`:

Q является печатным символом, отличным от пробела
Пробел не является печатным символом, отличным от пробела

Рис. 16.19. Использование функций `isspace`, `iscntrl`, `ispunct`, `isprint` и `isgraph` (часть 2 из 2)

16.10. Функции преобразования строк

В главе 5 мы обсудили несколько наиболее популярных функций языка C++ для операций с символьными строками. В нескольких следующих разделах мы рассмотрим другие такие функции, включая функции преобразования строк в численные значения, функции поиска в строках и функции копирования, сравнения и поиска блоков памяти.

В этом разделе рассматриваются функции преобразования строк из библиотеки *утилит общего назначения* (*stdlib*). Эти функции преобразуют строки цифр в целые значения или в значения с плавающей запятой. На рис. 16.20 приведены краткие сведения по этим функциям. Обратите внимание на использование `const` для объявления переменной `nPtr` в заголовках функций (читается справа налево как «`nPtr` является указателем на символьную константу»); `const` объявляет, что значение аргумента не будет модифицировано. При использовании библиотеки *утилит общего назначения* вы должны убедиться, что в программу включен заголовочный файл `<stdlib.h>`.

Функция `atof` (рис. 16.21) преобразует свой аргумент — строку, которая представляет собой число с плавающей запятой, в значение типа `double`. Эта функция возвращает значение типа `double`. Если преобразование невозможно, например, если первый символ строки не является цифрой, то поведение функции `atof` не определено.

Функция `atoi` преобразует свой аргумент — строку цифр, которая представляет собой целое, в значение типа `int`. Если преобразование невозможно, то поведение функции `atoi` не определено.

Прототип функции	Описание функции
double atof(const char *nPtr)	Преобразует строку nPtr в число типа double .
int atoi(const char *nPtr)	Преобразует строку nPtr в число типа int .
long atoi(const char *nPtr)	Преобразует строку nPtr в число типа long int .
double strtod(const *nPtr, char **endPtr)	Преобразует строку nPtr в число типа double .
long strtol(const char *nPtr, char **endPtr, int base)	Преобразует строку nPtr в число типа long .
unsigned long strtoul(const char *nPtr, char **endPtr, int base)	Преобразует строку nPtr в число типа unsigned long .

Рис. 16.20. Функции преобразования строк из библиотеки утилит общего назначения

```
// Использование atof
#include <iostream.h>
#include <stdlib.h>

main ()
{
    double d = atof("99.0");

    cout << "Строка \"99.0\" преобразуется"
        << " в значение типа double, равное "
        << d << "\nПреобразованное значение, деленное на 2, равно "
        << (d/2.0) << endl;
    return 0;
}
```

Строка "99.0" преобразуется в значение типа double, равное 99
 Преобразованное значение, деленное на 2, равно 49.5

Рис. 16.21. Использование функции **atof**

```
//Использование atoi
#include <iostream.h>
#include <stdlib.h>

main ()
{
    int i = atoi("2593");
    cout << "Строка \"2593\" преобразуется"
        << " в значение типа int, равное " << i
        << "\nПреобразованное значение минус 593 равно "
        << (i - 593)
        << endl;
    return 0;
}
```

Строка "2593" преобразуется в значение типа int, равное 2593

Преобразованное значение минус 593 равно 2000

Рис. 16.22. Использование функции **atoi**

Функция **atol** (рис. 16.23) преобразует свой аргумент — строку цифр, представляющую собой длинное целое, в значение типа **long**. Функция возвращает значение типа **long**. Если преобразование невозможно, поведение функции **atol** не определено. Если значения типа **Int** и типа **long** хранятся в области памяти размером 4 байта, то функции **atoi** и **atol** работают одинаково.

```
//Использование atol
#include <iostream.h>
#include <stdlib.h>
main ()
{
    long l = atol("1000000");
    cout << "Строка \"1000000\" преобразуется"
        << " в значение типа long, равное "
        << l << "\nПреобразованное значение, деленное на 2, равно "
        << (l / 2) << endl;
    return 0;
}
```

Строка "1000000" преобразуется в значение типа long, равное 1000000
Преобразованное значение, деленное на 2, равно 500000

Рис. 16.23. Использование функции **atol**

Функция **strtod** (рис. 16.24) преобразует последовательность символов, представляющую собой значение с плавающей точкой, в значение типа **double**. Эта функция принимает два аргумента: строку типа **char *** и указатель на строку. Переданная строка содержит последовательность символов, которые должны быть преобразованы к типу **double**. Второму аргументу присваивается позиция первого символа после преобразованного фрагмента строки. Оператор

```
d = strtod(string, &stringPtr);
```

из программы, приведенной на рис. 16.24, определяет, что переменной **d** присваивается значение типа **double**, преобразованное из переменной **string**, а **&stringPtr** присваивается позиция первого символа в **string** после преобразованного значения (51.2).

```
//Использование strtod
#include <iostream.h>
#include <stdlib.h>

main ()
{
    double d;
    char *string = "51.2% принято", *stringPtr;

    d = strtod(string, &stringPtr);
    cout << "Строка \""
        << string
        << "\" преобразуется в значение \n"
        << "типа double " << d << " и строку \""
        << stringPtr << "\"" << endl;
    return 0;
}
```

Рис. 16.24. Использование функции **strtod** (часть 1 из 2)

 Стока "51.2%" принято" преобразуется в значение типа double 51.2 и строку "% принято"

Рис. 16.24. Использование функции `strtod` (часть 2 из 2)

Функция `strtol` (рис. 16.25) преобразует в значение типа `long` последовательность символов, представляющую собой целое значение. Функция принимает три аргумента: строку типа `char *`, указатель на строку и целое значение. Стока состоит из символьной последовательности, которая должна быть преобразована. Второму аргументу присваивается позиция первого символа после преобразованного фрагмента строки. Целое значение задает основание системы счисления, используемое при преобразовании. Оператор

```
x = strtol(string, &remainderPtr, 0);
```

в программе, приведенной в рис. 16.25, определяет, что переменной `x` присваивается значение типа `long`, преобразованное из переменной `string`. Второму аргументу `&remainderPtr` присваивается остаток строки `string` после преобразования первой лексемы. Использование для второго аргумента значения `NULL` приводит к тому, что остаток строки игнорируется. Третий аргумент `0` показывает, что преобразовываемое значение может быть представлено в восьмеричном (основание 8), десятичном (основание 10) или шестнадцатеричном (основание 16) форматах. Основание может принимать значение `0` или любое значение между 2 и 36. Подробные сведения о восьмеричной, шестнадцатеричной и двоичной системах счисления вы можете получить в приложении Г, «Системы счисления». В представлении целых чисел по основаниям от 11 до 36 используются символы `A-Z`, которые отражают значения от 10 до 35. Например, шестнадцатеричное значение может включать цифры `0-9` и символы `A-F`. При основании 11 число может включать цифры `0-9` и символ `A`. При основании 24 число может включать цифры `0-9` и символы `A-N`. При основании 36 целое может включать цифры `0-9` и символы `A-Z`.

```
//Использование strtol
#include <iostream.h>
#include <stdlib.h>

main ()
{
    long x;
    char *string = "-1234567abc", *remainderPtr;

    x = strtol(string, &remainderPtr, 0);
    cout << "Первоначальная строка: \"" << string
        << "\nПреобразованное значение равно " << x
        << "\nОстаток первоначальной строки: \""
        << remainderPtr
        << "\nПреобразованное значение плюс 567 равно "
        << (x + 567) << endl;
    return 0;
}
```

Рис. 16.25. Использование функции `strtol` (часть 1 из 2)

```
Первоначальная строка: "-1234567abc"
Преобразованное значение равно -1234567
Остаток первоначальной строки: "abc"
Преобразованное значение плюс 567 равно-1234000
```

Рис. 16.25. Использование функции `strtol` (часть 2 из 2)

Функция `strtoul` (рис. 16.26) преобразует заданную последовательность символов, представляющую целое типа `unsigned long`, в соответствующее значение типа `unsigned long`. Эта функция работает идентично функции `strtol`. Оператор

```
x = strtoul (string, &remainderPtr, 0);
```

в программе на рис. 16.26 определяет, что переменной `x` присваивается значение типа `unsigned long`, преобразованное из переменной `string`. Второму аргументу `&remainderPtr` присваивается остаток строки `string` после преобразования. Третий аргумент `0` показывает, что преобразовываемое значение может быть представлено в восьмеричном, десятичном или шестнадцатеричном форматах.

```
//Использование strtoul
#include <iostream.h>
#include <stdlib.h>

main ()
{
    long x;
    char *string = "1234567abc", *remainderPtr;

    x = strtoul(string, &remainderPtr, 0);
    cout << "Первоначальная строка: \""
        << string
        << "\"\nПреобразованное значение равно " << x
        << "\nОстаток первоначальной строки: \""
        << remainderPtr
        << "\"\nПреобразованное значение минус 567 равно "
        << (x - 567) << endl;
    return 0;
}

Первоначальная строка: "1234567abc"
Преобразованное значение равно 1234567
Остаток первоначальной строки: "abc"
Преобразованное значение минус 567 равно 1234000
```

Рис. 16.26. Использование функции `strtoul`

16.11. Функции поиска из библиотеки обработки строк

В этом разделе описаны функции из библиотеки обработки строк, используемые для поиска символов и подстрок в строках. Эти функции собраны в таблице на рис. 16.27. Обратите внимание, что функции `strcspn` и `strspn` возвращают значение типа `size_t`. Тип `size_t` определен в стандарте как целый тип значения, возвращаемого операцией `sizeof`.

Замечание по мобильности 16.6

Тип **size_t** является системно-зависимым синонимом или типа **unsigned long**, или типа **unsigned int**.

Функция **strchr** ищет первое вхождение заданного символа в строку. Если символ найден, функция **strchr** возвращает указатель на этот символ в строке; в противном случае возвращается **NULL**. Программа на рис. 16.28 использует **strchr** для поиска первого вхождения букв 'о' и 'а' в строке «Это тест».

Функция **strcspn** (рис. 16.29) определяет длину начальной части строки, являющейся ее первым параметром, которая не содержит ни одного символа, входящего в строку, являющуюся ее вторым параметром. Функция возвращает длину найденной начальной части.

Прототип функции	Описание функции
<code>char *strchr(const char *s, int c)</code>	Определяет позицию первого вхождения символа c в строку s . Если c найден, возвращается указатель на c в s . В противном случае возвращается указатель NULL .
<code>size_t strcspn(const char *s1, const char *s2)</code>	Определяет и возвращает длину начальной части строки s1 , состоящей из символов, не содержащихся в строке s2 .
<code>size_t strspn(const char *s1, const char *s2)</code>	Определяет и возвращает длину начальной части строки s1 , состоящей только из символов, содержащихся в строке s2 .
<code>char *strupbrk(const char *s1, const char *s2)</code>	Определяет позицию первого вхождения в строку s1 любого из символов строки s2 . Если символ из строки s2 найден в строке s1 , то возвращается указатель на этот символ в s1 . В противном случае возвращается указатель NULL .
<code>char *strchr(const char *s, int c)</code>	Определяет позицию последнего вхождения символа c в строку s . Если c найден, возвращается указатель на c в s . В противном случае возвращается указатель NULL .
<code>char *strstr(const char *s1, const char *s2)</code>	Определяет позицию первого вхождения в строку s1 подстроки s2 . Если подстрока найдена, возвращается указатель на нее в s1 . В противном случае возвращается указатель NULL .

Рис. 16.27. Функции поиска из библиотеки обработки строк

```
//Использование strchr
#include <iostream.h>
#include <string.h>

main ()
{
    char *string = "Это тест";
    char character1 = 'o', character2 = 'a';

    if (strchr(string, character1) != NULL)
        cout << "Символ '" << character1 << "' найден в \""
            << string << "\"." << endl;
    else
        cout << "Символ '" << character1 << "' не найден в \""
            << string << "\"." << endl;

    if (strchr(string, character2) != NULL)
        cout << "Символ '" << character2 << "' найден в \""
            << string << "\"." << endl;
    else
        cout << "Символ '" << character2 << "' не найден в \""
            << string << "\"." << endl;
    return 0;
}
```

Символ 'о' найден в "Это тест".
 Символ 'а' не найден в "Это тест".

Рис. 16.28. Использование функции `strchr`

```
//Использование strcspn
#include <iostream.h>
#include <string.h>

main ()
{
    char *string1 = "Значение равно 3.14159";
    char *string2 = "1234567890";

    cout << "string1 = " << string1 << "\nstring2 = " << string2
        << "\n\nДлина начальной части string1,"
        << "\nне содержащей символов из string2, равна "
        << strcspn(string1, string2) << endl;
    return 0;
}
```

string1 = Значение равно 3.14159
 string2 = 1234567890

Длина начальной части string1,
 не содержащей символов из string2, равна 15

Рис. 16.29. Использование функции `strcspn`

Функция **strpbrk** определяет позицию первого вхождения в строку, являющуюся ее первым параметром, любого из символов строки, являющейся ее вторым параметром. Если символ из второй строки найден в первой строке, функция **strpbrk** возвращает указатель на этот символ. В противном случае возвращается указатель **NULL**. Программа на рис. 16.30 определяет первое вхождение в **string1** любого символа из **string2**.

Функция **strrchr** определяет последнее вхождение заданного символа в строку. Если этот символ найден, функция **strrchr** возвращает указатель на этот символ в строке; в противном случае возвращается **NULL**. Программа на рис. 16.31 определяет последнее вхождение символа 'з' в строку "В зоопарке есть много животных, включая зебру".

```
//Использование strpbrk
#include <iostream.h>
#include <string.h>

main ()
{
    char *string1 = "Это тест";
    char *string2 = "берегись";

    cout << "Среди символов строки \""
        << string2 << "\"\n"
        << *strpbrk(string1, string2) << " - "
        << "первый, встречающийся в строке\n\""
        << string1 << '\"' << endl;
    return 0;
}

Среди символов строки "берегись"
'e' - первый, встречающийся в строке
"Это тест"
```

Рис. 16.30. Использование функции **strpbrk**

```
//Использование strrchr
#include <iostream.h>
#include <string.h>

main ()
{
    char *string1 = "В зоопарке есть много животных, включая
зебру";
    int c = 'з';

    cout << "Остаток string1, начинающийся с последнего\n"
        << "вхождения символа '" << (char) c
        << "'": \""
        << strrchr(string1, c) << '\"' << endl;
    return 0;
}

Остаток string1, начинающийся с последнего
вхождения символа 'з': "зебру"
```

Рис. 16.31. Использование функции **strrchr**

Функция *strspn* (рис. 16.32) определяет длину начальной части строки, являющейся ее первым параметром, которая состоит только из символов строки, являющейся ее вторым параметром. Функция возвращает длину этой начальной части строки.

Функция *strstr* определяет позицию первого вхождения в строку, являющуюся ее первым параметром, подстроки, являющейся ее вторым параметром. Если подстрока найдена, возвращается указатель на нее в строке первого параметра. В противном случае возвращается указатель NULL. Программа на рис. 16.33 использует *strstr* для поиска подстроки "бра" в строке "абракадабра".

```
//Использование strspn
#include <iostream.h>
#include <string.h>

main ()
{
    char *string1 = "Значение равно 3.14159";
    char *string2 = "аонЗечирв ";

    cout << "string1 = " << string1 << "\nstring2 = " << string2
        << "\n\nДлина начальной части string1, \n"
        << "состоящей только из символов string2 ="
        << strspn(string1, string2) << endl;
    return 0;
}

string1 = Значение равно 3.14159
string2 = аонЗечирв

Длина начальной части string1,
состоящей только из символов string2 =15
```

Рис. 16.32. Использование функции *strspn*

```
//Использование strstr
#include <iostream.h>
#include <string.h>

main ()
{
    char *string1 = "абракадабра";
    char *string2 = "бра";

    cout << "string1 = " << string1 << "\nstring2 = " << string2
        << "\n\nОстаток string1, начинающийся\n"
        << "с первого вхождения string2: "
        << strstr(string1, string2) << endl;
    return 0;
}

string1 = абракадабра
string2 = бра

Остаток string1, начинающийся
с первого вхождения string2: бракадабра
```

Рис. 16.33. Использование функции *strstr*

16.12. Функции работы с памятью из библиотеки обработки строк

Библиотека обработки строк представлена в этом разделе функциями, осуществляющими копирование, сравнение и поиск блоков памяти. Эти функции трактуют блоки памяти как массивы символов. Они могут манипулировать любыми блоками данных. В таблице на рис. 16.34 приведены функции работы с областями памяти из библиотеки обработки строк. При рассмотрении этих функций под «объектом» подразумевается блок данных.

Прототип функции	Описание функции
<code>void *memcpy(void *s1, const void *s2, size_t n)</code>	Копирует n символов из объекта, на который указывает s2 , в объект, на который указывает s1 . Возвращается указатель на объект результата.
<code>void *memmove(void *s1, const void *s2, size_t n)</code>	Копирует n символов из объекта, на который указывает s2 , в объект, на который указывает s1 . Копирование осуществляется так, как если бы символы сначала копировались из объекта, на который указывает s2 , во временный массив, а затем копировались из этого временного массива в объект, на который указывает s1 . Возвращается указатель на объект результата.
<code>int *memcmp(const void *s1, const void *s2, size_t n)</code>	Сравнивает первые n символов объектов, на которые указывают s1 и s2 . Функция возвращает 0, отрицательное значение или положительное значение, если соответственно s1 равна, меньше или больше s2 .
<code>void *memchr(const void *s, int c, size_t n)</code>	Определяет первое вхождение символа c (преобразованного к unsigned char) в первые n символов объекта, на который указывает s . Если символ c найден, возвращается указатель на этот символ в объекте. В противном случае возвращается указатель NULL .
<code>void *memset(void *s, int c, size_t n)</code>	Копирует символ c (преобразованный к unsigned char) в первые n символов объекта, на который указывает s . Возвращается указатель на объект результата.

Рис. 16.34. Функции работы с памятью из библиотеки обработки строк

Параметры указатели в этих функциях объявляются типа **void ***. В главе 5 мы увидели, что указатели на данные любого типа могут быть непосредственно присвоены указателям типа **void ***. Поэтому рассматриваемые функции могут принимать указатели на любые типы данных. Помните, что указатели типа **void *** сами не могут непосредственно присваиваться указателям какого-либо типа данных. Поскольку указатель типа **void *** не может быть разименован, каждая функция получает в качестве параметра размер, определяющий число символов (байтов), которые она должна обработать. Для упрощения примеры этого раздела оперируют с массивами символов (блоками символов).

Функция *memcpу* копирует заданное число символов из объекта, на который указывает ее второй параметр, в объект, на который указывает ее первый параметр. Функция может принимать указатель на объект любого типа. Результат работы функции не определен, если эти два объекта перекрываются в памяти, например, если они являются частями одного и того же объекта. Программа на рис. 16.35 использует *memcpу* для копирования строки, хранящейся в массиве *s2*, в массив *s1*.

```
//Использование memcpу
#include <iostream.h>
#include <string.h>

main ()
{
    char s1[22], s2[ ] = "Копируется эта строка";

    memcpу(s1, s2, 22);
    cout << "После того, как s2 скопирована в s1 функцией memcpу,\n"
        << "s1 содержит \"\" << s1 << '\"' << endl;
    return 0;
}
```

После того, как s2 скопирована в s1 функцией *memcpу*,
s1 содержит "Копируется эта строка"

Рис. 16.35. Использование функции *memcpу*

Функция *memmove* подобно *memcpу* копирует заданное число символов из объекта, на который указывает ее второй параметр, в объект, на который указывает ее первый параметр. Но копирование осуществляется так, как если бы байты сначала копировались из объекта, на который указывает второй параметр, во временный массив символов, а затем копировались из этого временного массива в объект, на который указывает первый параметр. Это позволяет, например, копировать символы из одной части строки в другую часть той же строки.

Типичная ошибка программирования 16.11

Все функции копирующие строки, кроме *memmove*, дают неопределенный результат при копировании одной части строки в другую часть той же строки.

Программа на рис. 16.36 использует функцию *memmove* для копирования последних 10 байтов массива *x* в первые 10 байтов того же массива *x*.

Функция *memcp* (рис. 16.37) сравнивает указанное число символов своего первого параметра с соответствующими символами своего второго параметра. Функция возвращает 0, отрицательное значение или положительное значение, если соответственно первый параметр равен, меньше или больше второго параметра.

```
//Использование memmove
#include <iostream.h>
#include <string.h>

main ()
{
    char x[ ] = "Home Sweet Home";

    cout << "Строка в массиве x до применения memmove: "
        << x << endl;
    cout << "Строка в массиве x после применения memmove: "
        << (char *) memmove(x, &x[5], 10) << endl;
    return 0;
}
```

Строка в массиве x до применения memmove: Home Sweet Home
 Страна в массиве x после применения memmove: Sweet Home Home

Рис. 16.36. Использование функции **memmove**

```
//Использование memcmp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

main ()
{
    char s1[ ] = "ABCDEFG", s2[ ] = "ABCDXYZ";

    cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
        << "\nmemcmp(s1, s2, 4) = " << setw(3) << memcmp(s1, s2, 4)
        << "\nmemcmp(s1, s2, 7) = " << setw(3) << memcmp(s1, s2, 7)
        << "\nmemcmp(s2, s1, 4) = " << setw(3) << memcmp(s2, s1, 4)
        << endl;
    return 0;
}

s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) = 0
memcmp(s1, s2, 7) = -19
memcmp(s2, s1, 4) = 0
```

Рис. 16.37. Использование функции **memcmp**

Функция **memchr** определяет первое вхождение байта, представленного как **unsigned char**, в указанное число байтов объекта. Если указанный байт найден, возвращается указатель на этот байт объекта; в противном случае возвращается указатель **NULL**. Программа на рис. 16.38 ищет символ (байт) 'р' в строке "Это строка".

```
//Использование memchr
#include <iostream.h>
#include <string.h>

main ()
{
    char *s = "Это строка";
    cout << "Оставшаяся часть строки после найденного символа 'р': "
\"
        << (char *) memchr(s, 'р', 11) << '\n' << endl;
    return 0;
}
```

Оставшаяся часть строки после найденного символа 'р': "рока"

Рис. 16.38. Использование функции **memchr**

Функция *memset* копирует значение байта, являющегося ее вторым аргументом, в заданное число байтов объекта, на который указывает ее первый аргумент. Программа на рис. 16.39 использует *memset* для копирования 'б' в первые 7 байтов строки *string1*.

```
//Использование memset
#include <iostream.h>
#include <string.h>

main ()
{
    char string1[15] = "ББББББББББББББ";
    cout << "string1 = " << string1 << endl;
    cout << "string1 после выполнения memset = "
        << (char *) memset(string1, 'б', 7) << endl;
    return 0;
}
```

string1 = ББББББББББББББ

string1 после выполнения memset = бббббббббббббб

Рис. 16.39. Использование функции **memset**

16.13. Другие функции библиотеки обработки строк

Нам осталось рассмотреть функцию *strerror* из библиотеки обработки строк. Эта функция представлена на рис. 16.40.

Функция принимает номер ошибки и находит в системе строку сообщения об этой ошибке. Возвращается указатель на эту строку. Программа на рис. 16.41 демонстрирует применение *strerror*.

Замечание по мобильности 16.7

Сообщения, генерируемые *strerror*, являются системно-зависимыми.

Прототип функции	Описание функции
char *strerror(int errornum)	Находит системно зависимое сообщение об ошибке с номером errornum . Возвращается указатель на эту строку.

Рис. 16.40. Еще одна функция работы с строкой из библиотеки обработки строк

```
//Использование strerror
#include <iostream.h>
#include <string.h>

main ()
{
    char string1[15] = "BBBBBBBBBBBBBB";
    cout << strerror(2) << endl;
    return 0;
}
```

No such file or directory

Рис. 16.41. Использование функции **strerror**

Резюме

- Структуры являются набором (иногда такой набор называют агрегатом) связанных данных, объединенных одним именем.
- Структуры могут содержать переменные разных типов данных.
- Каждое описание структуры начинается с ключевого слова **struct**. Внутри фигурных скобок в описании структуры записываются ее элементы.
- Элементы одной структуры должны иметь уникальные имена.
- Объявление структуры создает новый тип данных, который может использоваться в дальнейшем при объявлении переменных типа структуры.
- Структуры могут быть инициализированы с помощью списка инициализаторов (начальных значений): вслед за именем в объявлении переменной ставится знак равенства, а затем в фигурных скобках записывается список инициализаторов, разделенных запятой. Если инициализаторов в списке меньше, чем число элементов в структуре, то оставшимся элементам автоматически присваиваются нулевые начальные значения (или значение **NULL**, если элемент является указателем).
- Переменную типа структуры можно присваивать другим переменным структурам того же типа.
- Переменную типа структуры можно инициализировать переменной структурой того же типа.
- Структуры в целом и отдельные их элементы передаются в функции вызовом по значению. Массивы элементы, конечно, передаются вызовом по ссылке.

- Чтобы передать структуру в функцию вызовом по значению, надо передать ее адрес.
- Создание нового имени с помощью `typedef` не приводит к появлению нового типа данных; `typedef` просто создает новое имя уже описанного типа.
- Операция поразрядное И (`&`) имеет два целых операнда. Бит результата устанавливается в 1, если соответствующие биты обоих operandов установлены в 1.
- Маски используются для того, чтобы выделить в анализируемом значении указанные биты и сделать невидимыми остальные.
- Операция поразрядное ИЛИ (`|`) имеет два целых операнда. Бит результата устанавливается в 1, если соответствующие биты хотя бы в одном из operandов установлены в 1.
- Каждая из поразрядных операций (за исключением поразрядного НЕ) имеет соответствующую операцию присваивания.
- Операция поразрядное исключающее ИЛИ (`^`) имеет два целых операнда. Бит результата устанавливается в 1, если в одном и только в одном из operandов соответствующий бит установлен в 1.
- Операция сдвига влево (`<<`) сдвигает биты первого операнда влево на количество битов, заданное вторым operandом. Правые освобождающиеся биты заполняются 0.
- Операция сдвига вправо (`>>`) сдвигает биты своего левого операнда вправо на заданное правым operandом количество битов. Выполнение операции сдвига вправо целого значения типа `unsigned` приводит к освобождению левых битов, которые заполняются нулями. Освобождающиеся левые биты в целом числе со знаком могут заполняться или 0, или 1 в зависимости от используемой компьютерной системы.
- Операция поразрядное НЕ (поразрядное дополнение, отрицание) (`~`) имеет один operand и устанавливает в нем биты со значением 1 в 0, а биты со значением 0 в 1; это называется также дополнением значения до единицы.
- Битовые поля позволяют рационально использовать память за счет хранения данных в минимально требуемом количестве битов.
- Битовые поля должны быть объявлены как тип `int` или `unsigned`.
- При объявлении битового поля вслед за указанием типа элемента `unsigned` или `int` ставится двоеточие и пишется целочисленная константа, задающая ширину поля.
- Ширина поля должна быть целочисленной константой в диапазоне между 0 и заданным общим числом битов, используемых для хранения целого значения типа `int` в вашей системе.
- Если битовое поле определено без имени, то оно используется в структуре как заполнение.
- Неименованное битовое поле нулевой ширины используется для выравнивания следующего битового поля по границе нового машинного слова.

- Функция `islower` определяет, является ли ее аргумент строчной буквой (a–z).
- Функция `isupper` определяет, является ли ее аргумент прописной буквой (A–Z).
- Функция `isdigit` определяет, является ли аргумент цифрой (0–9).
- Функция `isalpha` определяет, является ли аргумент прописной буквой (A–Z) или строчной буквой (a–z).
- Функция `isalnum` определяет, является ли аргумент прописной буквой, строчной буквой или цифрой (A–Z, a–z, 0–9).
- Функция `isxdigit` определяет, является ли аргумент шестнадцатеричной цифрой (A–F, a–f, 0–9).
- Функция `toupper` преобразует строчную букву (в нижнем регистре) в прописную (в верхнем регистре).
- Функция `tolower` преобразует прописную букву (в верхнем регистре) в строчную (в нижнем регистре).
- Функция `isspace` определяет, является ли аргумент одним из следующих символов разделителей: ' ' пробел, '\f', '\n', '\r', '\t' или '\v'.
- Функция `iscntrl` определяет, является ли ее аргумент одним из следующих управляющих символов: '\t', '\v', '\f', '\a', '\b', '\r' или '\n'.
- Функция `ispunct` определяет, является ли ее аргумент печатным символом, отличным от пробела, цифры или буквы.
- Функция `isprint` определяет, является ли ее аргумент печатным символом, включая символ пробела.
- Функция `isgraph` определяет, является ли ее аргумент печатным символом, отличным от пробела.
- Функция `atof` преобразует свой аргумент — строку символов, которая представляет собой число с плавающей запятой, в значение типа `double`.
- Функция `atoi` преобразует свой аргумент — строку цифр, которая представляет собой целое число, в значение типа `int`.
- Функция `atol` преобразует свой аргумент — строку цифр, представляющую собой длинное целое число, в значение типа `long`.
- Функция `strtod` преобразует последовательность символов, представляющую собой число с плавающей точкой, в значение типа `double`. Эта функция принимает два аргумента: строку типа `char *` и указатель на строку `char *`. Переданная строка содержит последовательность символов, которые должны быть преобразованы к типу `double`. Второму аргументу присваивается позиция первого символа после преобразованного фрагмента строки.
- Функция `strtol` преобразует в значение типа `long` последовательность символов, представляющую собой целое число. Функция принимает три аргумента: строку типа `char *`, указатель на `char *` и целое значение. Стока состоит из символьной последовательности, которая должна быть преобразована. Второму аргументу присваивается позиция первого символа после преобразованного фрагмента строки. Целое зна-

чение задает основание системы счисления, используемое при преобразовании.

- Функция `strtoul` преобразует заданную последовательность символов, представляющую целое число типа `unsigned long`, в соответствующее значение типа `unsigned long`. Функция принимает три аргумента: строку типа `char *`, указатель на `char *` и целое значение. Стока состоит из символьной последовательности, которая должна быть преобразована. Второму аргументу присваивается позиция первого символа после преобразованного фрагмента строки. Целое значение задает основание системы счисления, используемое при преобразовании.
- Функция `strchr` ищет первое вхождение заданного символа в строку. Если символ найден, функция `strchr` возвращает указатель на этот символ в строке; в противном случае возвращается `NULL`.
- Функция `strcspn` определяет длину начальной части строки, являющейся ее первым параметром, которая не содержит ни одного символа, входящего в строку, являющуюся ее вторым параметром. Функция возвращает длину найденной начальной части.
- Функция `strupr` определяет позицию первого вхождения в строку, являющейся ее первым параметром, любого из символов строки, являющейся ее вторым параметром. Если символ из второй строки найден в первой строке, функция `strupr` возвращает указатель на этот символ. В противном случае возвращается указатель `NULL`.
- Функция `strrchr` определяет последнее вхождение заданного символа в строку. Если этот символ найден, функция `strrchr` возвращает указатель на этот символ в строке; в противном случае возвращается `NULL`.
- Функция `strspn` определяет длину начальной части строки, являющейся ее первым параметром, которая состоит только из символов строки, являющейся ее вторым параметром. Функция возвращает длину этой начальной части строки.
- Функция `strstr` определяет позицию первого вхождения в строку, являющуюся ее первым параметром, подстроки, являющейся ее вторым параметром. Если подстрока найдена, возвращается указатель на нее в строке первого параметра. В противном случае возвращается указатель `NULL`.
- Функция `memcp` копирует заданное число символов из объекта, на который указывает ее второй параметр, в объект, на который указывает ее первый параметр. Функция может принимать указатель на объект любого типа. Принимаемые функцией `memcp` указатели типа `void` преобразуются в ней в указатели типа `char`. Функция манипулирует с байтами аргумента как с символами.
- Функция `memmove` копирует заданное число байтов из объекта, на который указывает ее второй параметр, в объект, на который указывает ее первый параметр. Копирование осуществляется так, как если бы байты сначала копировались из объекта, на который указывает второй параметр, во временный массив символов, а затем копировались из этого временного массива в объект, на который указывает первый параметр.

- Функция `memchr` сравнивает указанное число символов своего первого параметра с соответствующими символами своего второго параметра.
- Функция `memchr` определяет первое вхождение байта, представленного как `unsigned char`, в указанное число байтов объекта. Если указанный байт найден, возвращается указатель на этот байт объекта; в противном случае возвращается указатель `NULL`.
- Функция `memset` копирует значение байта, являющегося ее вторым аргументом и трактуемого как `unsigned char`, в заданное число байтов объекта, на который указывает ее первый аргумент.
- Функция `strerror` принимает номер ошибки и находит в системе строку сообщения об этой ошибке. Возвращается указатель на эту строку.

Терминология

<code>&</code> операция поразрядного И	<code>strpbrk</code>
<code>&=</code> операция присваивания поразрядного И	<code>strchr</code>
<code><<</code> операция сдвига влево	<code>strcspn</code>
<code><<=</code> операция присваивания сдвига влево	<code>strerror</code>
<code>>></code> операция сдвига вправо	<code>string.h</code>
<code>>>=</code> операция присваивания сдвига вправо	<code>strrchr</code>
<code>^</code> операция поразрядного исключающего ИЛИ	<code>strspn</code>
<code>^=</code> операция присваивания поразрядного исключающего ИЛИ	<code>strrstr</code>
<code> </code> операция поразрядного ИЛИ	<code>strtod</code>
<code>=</code> операция присваивания поразрядного ИЛИ	<code>strtol</code>
<code>-</code> операция поразрядного НЕ (дополнения, отрицания)	<code>strtoul</code>
<code>ASCII</code>	<code>struct</code>
<code>atof</code>	<code>tolower</code>
<code>atoi</code>	<code>toupper</code>
<code>atol</code>	<code>typedef</code>
<code>ctype.h</code>	библиотека утилит общего назначения
<code>isalnum</code>	битовое поле
<code>isalpha</code>	битовое поле нулевой ширины
<code>iscntrl</code>	буква
<code>isdigit</code>	дополнение
<code>isgraph</code>	дополнение до единицы
<code>islower</code>	запись
<code>isprint</code>	заполнение
<code>ispunct</code>	инициализация структур
<code>isspace</code>	код символа
<code>isupper</code>	компромисс память — время
<code>isxdigit</code>	маска
<code>memchr</code>	маскирование битов
<code>memcmp</code>	массив структур
<code>memcpy</code>	набор символов
<code>memmove</code>	неименованное битовое поле
<code>memset</code>	обработка строк
<code>stdlib.h</code>	обработка текстов
	ограничитель
	печатный символ
	подстрока (цепочка поиска)
	присваивание структуры
	самоадресуемая структура
	сдвиг

сдвиг влево	тип структуры
сдвиг вправо	указатель на структуру
символ разделитель	управляющий символ
символьная константа	функции преобразования строк
строка	шестнадцатеричная цифра
строковая константа	ширина битового поля

Типичные ошибки программирования

- 16.1. Забывают поставить точку с запятой после окончания объявления структуры.
- 16.2. Присваивание структуры одного типа структуре другого типа.
- 16.3. Сравнение структур является синтаксической ошибкой из-за разных требований по выравниванию в разных системах.
- 16.4. Ошибочно предполагается, что структуры подобно массивам автоматически передаются вызовом по ссылке и предпринимается попытка изменить в вызываемой программе значения структуры в вызывающей функции.
- 16.5. Забывают включить индекс массива при ссылке на отдельные структуры в массиве структур.
- 16.6. Использование операции логическое И (`&&`) вместо операции по-разрядное И (`&`) и наоборот.
- 16.7. Использование операции логическое ИЛИ (`||`) вместо операции по-разрядное ИЛИ (`|`) и наоборот.
- 16.8. Результат сдвига какого-либо значения не определен, если правый операнд является отрицательным или если правый operand больше числа битов в левом операнде.
- 16.9. Попытка осуществить доступ к отдельным битам битового поля как если бы они были элементами массива. Битовые поля не являются «массивами битов».
- 16.10. Попытка получить адрес битового поля (операция `&` не может применяться к битовым полям, поскольку они не имеют адресов).
- 16.11. Все функции копирующие строки, кроме `memmove`, дают неопределенный результат при копировании одной части строки в другую часть той же строки.

Хороший стиль программирования

- 16.1. При создании типа структуры следует предусмотреть и создание имени структуры. Имя структуры удобно использовать в дальнейшем для объявления новых переменных типа данной структуры.
- 16.2. Записывайте имена в предложении `typedef` заглавными буквами, чтобы подчеркнуть, что эти имена являются синонимами других имен типов.

Советы по повышению эффективности

- 16.1. Передача структур (особенно больших структур) вызовом по ссылке является более эффективной, чем передача структур вызовом по значению, при которой необходимо копировать всю структуру.
- 16.2. Битовые поля способствуют рациональному использованию памяти.
- 16.3. Хотя битовые поля сокращают требования к памяти, их использование может привести к тому, что компилятор будет генерировать машинный код, который выполняется с низкой скоростью. Это происходит вследствие того, что приходится использовать дополнительные операции машинного языка для получения доступа к отдельным частям адресуемых элементов памяти. Это является одним из множества примеров необходимости компромисса между требованиями эффективности по памяти и по времени выполнения программы.

Замечания по мобильности

- 16.1. Поскольку размер данных-элементов определенного типа является машинно-зависимым и выравнивание памяти тоже является машинно-зависимым, то и представление структуры также машинно-зависимо.
- 16.2. Использование `typedef` позволяет создавать более мобильные программы.
- 16.3. Поразрядные манипуляции с данными являются машинно- зависимыми.
- 16.4. Результат сдвига вправо целого значения со знаком (типа `signed`) является машинно- зависимым. Некоторые компьютеры заполняют освобожденные биты нулями, а другие компьютеры используют бит знака.
- 16.5. Манипуляции с битовыми полями являются машинно- зависимыми. Например, в некоторых компьютерах битовые поля могут пересекать границы машинного слова, тогда как в других компьютерах это недопустимо.
- 16.6. Тип `size_t` является системно- зависимым синонимом или типа `unsigned long`, или типа `unsigned int`.
- 16.7. Сообщения, генерируемые `strerror`, являются системно- зависимыми.

Упражнения для самопроверки

- 16.1. Заполнить пробелы в следующих утверждениях:

- a) _____ является набором связанных переменных, объединенных одним именем.
- b) Биты в результате вычисления выражения, использующего операцию _____, устанавливаются в 1, если соответствующие биты в обоих операндах установлены в 1. В противном случае биты устанавливаются в 0.

- c) Переменные, объявленные в описании структуры, называются ее _____.
- d) Биты в результате вычисления выражения, использующего операцию _____, устанавливаются в 1, если по крайней мере в одном операнде соответствующие биты установлены в 1. В противном случае биты устанавливаются в 0.
- e) Объявление структуры начинается с ключевого слова _____.
- f) Для создания синонима ранее определенного типа данных используется ключевое слово _____.
- g) Биты в результате вычисления выражения, использующего операцию _____, устанавливаются в 1, если в одном и только в одном операнде соответствующие биты установлены в 1. В противном случае биты устанавливаются в 0.
- h) Операция поразрядное И (&) часто используется, чтобы _____ биты, т.е. выделить определенные биты и сделать невидимыми остальные.
- i) На имя типа структуры можно ссылаться как на ее _____.
- j) Доступ к элементам структуры осуществляется записью _____ или записью _____.
- k) Операции _____ и _____ используются для сдвига значений битов соответственно влево или вправо.

16.2. Укажите, что из нижеследующего верно, а что неверно. Если неверно, то объясните, почему.

- a) Структуры могут содержать данные только одного типа.
- b) Элементы разных структур должны иметь уникальные имена.
- c) Ключевое слово `typedef` используется для определения нового типа данных.
- d) Структуры всегда передаются в функции вызовом по ссылке.

16.3. Напишите один или группу операторов, выполняющих следующее:

- a) Опишите структуру с именем `Part`, содержащую переменную `partNumber` типа `int` и массив `partName` типа `char`, элементы которого могут содержать до 25 символов.
- b) Определите `PartPtr` как синоним типа `Part *`.
- c) Объявите переменную `a` типа `Part`, массив `b[10]` элементов типа `Part` и переменную `ptr`, являющуюся указателем на тип `Part`.
- d) Прочтите с клавиатуры число и имя в соответствующие данные-элементы переменной `a`.
- e) Присвойте значения данных-элементов переменной `a` элементу 3 массива `b`.
- f) Присвойте адрес массива `b` переменной `ptr`.
- g) Напечатайте значения данных-элементов элемента 3 массива `b`, используя переменную `ptr` и операцию доступа к элементу структуры.

16.4. Найдите ошибку в каждом из следующих пунктов:

- a) Предположите, что **struct Card** была определена как содержащая два указателя на тип **char** с именами **face** и **suit**. Была также объявлена переменная **c** типа **Card** и переменная **cPtr** как указатель на **Card**. Переменной **cPtr** присвоен в качестве начального значения адрес переменной **b**.

```
cout << *cPtr.face << endl;
```

- b) Предположите, что **struct Card** была определена как содержащая два указателя на тип **char** с именами **face** и **suit**. Был также объявлен массив **hearts[13]** типа **Card**. Следующий оператор должен печатать значение элемента **face** в элементе 10 этого массива.

```
cout << hearts.face << endl;
```

- c) struct Person {
 char lastName[15];
 char firstName[15];
 int age;
}

- d) Предположите, что объявлены переменная **p** типа **Person** и переменная **c** типа **Card**.

```
p = c;
```

16.5. Напишите по одному оператору, выполняющему каждый из перечисленных ниже пунктов. Предполагайте, что переменные **c** (в ней хранится символ), **x**, **y** и **z** имеют тип **int**, переменные **d**, **e** и **f** имеют тип **float**, переменная **ptr** имеет тип **char ***, массивы **s1[100]** и **s2[100]** имеют тип **char**.

- a) Преобразуйте символ, хранящийся в **c**, в символ верхнего регистра. Результат присвойте переменной **C**.
- b) Определите, является ли значение переменной **c** цифрой. Используйте условную операцию, как это делалось в программах на рис. 16.17, 16.18 и 16.19, чтобы печатать «является» или «не является» при выдаче результатов на экран.
- c) Преобразуйте строку "1234567" в значение типа **long** и напечатайте его.
- d) Определите, является ли значение переменной **c** управляемым символом. Используйте условную операцию, чтобы печатать «является» или «не является» при выдаче результатов на экран.
- e) Присвойте указателю **ptr** адрес последнего вхождения **c** в строку **s1**.
- f) Преобразуйте строку "8.63562" в значение типа **double** и напечатайте его.
- g) Определите, является ли значение переменной **c** буквой. Используйте условную операцию, чтобы печатать «является» или «не является» при выдаче результатов на экран.
- h) Присвойте указателю **ptr** адрес последнего вхождения строки **s2** в строку **s1**.

- i) Определите, является ли значение переменной с печатным символом. Используйте условную операцию, чтобы печатать « является» или « не является» при выдаче результатов на экран.
- j) Присвойте указателю `ptr` адрес последнего вхождения в строку `s1` любого символа из строки `s2`.
- k) Присвойте указателю `ptr` адрес первого вхождения `c` в строку `s1`.
- l) Преобразуйте строку "-21" в значение типа `int` и напечатайте его.

Ответы на упражнения для самопроверки

- 16.1.** a) структура. b) поразрядное И (&). c) элементами (членами). d) поразрядное ИЛИ (|). e) struct. f) typedef. g) поразрядное исключающее ИЛИ (^). h) замаскировать. i) тэг (ярлык). j) элемент структуры (), указатель на структуру (->). k) сдвига влево (<<), сдвига вправо (>>).
- 16.2.** a) Неверно. Структуры могут содержать данные различных типов.
- b) Неверно. Элементы разных структур могут иметь совпадающие имена, но элементы одной структуры должны иметь уникальные имена.
- c) Неверно. `typedef` используется для определения псевдонима (синонима) ранее созданного типа данных.
- d) Неверно. Структуры всегда передаются в функции вызовом по значению.
- 16.3.** a)

```
struct Part {
    int partNumber;
    char partName[26];
};
```
- b) `typedef Part * PartPrt;`
- c) `Part a, b[10], *ptr;`
- d) `cin >> a.partNumber >> a.partName;`
- e) `b[3] = a;`
- f) `ptr = b;`
- g) `cout << (ptr + 3)->partNumber << ' '
 << (ptr + 3)->partName << endl;`
- 16.4.** a) Ошибка: пропущены скобки, в которые должно быть заключено `* cPtr` и без которых последовательность вычислений выражения будет неверной.
- b) Ошибка: пропущен индекс массива. Правильное выражение: `hearts[10].face`.
- c) Ошибка: в конце описания структуры должна быть точка с запятой.
- d) Ошибка: переменные разных типов структур нельзя присваивать друг другу.

- 16.5. a) `c = toupper(c);`
- b) `cout << '\' << c << '\''
 << (isdigit(c) ? " является" : " не является"
 << " цифрой" << endl;`
- c) `cout << atol("1234567") << endl;`
- d) `cout << '\' << c << '\''
 << (iscntrl(c) ? " является" : " не является"
 << " управляющим символом" << endl;`
- e) `ptr = strrchr(s1, c);`
- f) `cout << atof("1234567") << endl;`
- g) `cout << '\' << c << '\''
 << (isalpha(c) ? " является" : " не является"
 << " буквой" << endl;`
- h) `ptr = strstr(s1, s2);`
- i) `cout << '\' << c << '\''
 << (isprint(c) ? " является" : " не является"
 << " печатным символом" << endl;`
- j) `ptr = strpbrk(s1, s2);`
- k) `ptr = strchr(s1, c);`
- l) `cout << atoi("-21") << endl;`

Упражнения

16.6. Напишите определения следующих структур и объединений:

- a) Структура **Inventory**, содержащая массив символов `partName[30]`, целое `partNumber`, элемент с плавающей запятой `price`, целое `stock` и целое `reorder`.
- b) Структура **Address** (адрес), содержащая массивы символов `streetAddress[25]` (улица), `city[20]` (город), `state[3]` (штат) и `zipCode[6]` (почтовый код).
- c) Структура **Student**, содержащая массивы `firstName[15]`, `lastName[15]` и переменную `homeAddress` типа struct **Address** из пункта (b).
- d) Структура **Test**, содержащая 16 битовых полей шириной в 1 бит каждое. Именами этих полей являются буквы от а до р.

16.7. Имея следующие описания структур и объявления переменных

```
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 доступа к следующим элементам структур.

- a) Элемент `lastName` структуры `customerRecord`.
- b) Элемент `lastName` структуры, на которую указывает `customerPtr`.
- c) Элемент структуры `customerRecord`.
- d) Элемент `firstName` структуры, на которую указывает `customerPtr`.
- e) Элемент `customerNumber` структуры `customerRecord`.
- f) Элемент `customerNumber` структуры, на которую указывает `customerPtr`.
- g) Элемент `phoneNumber` элемента `personal` структуры `customerRecord`.
- h) Элемент `phoneNumber` элемента `personal` структуры, на которую указывает `customerPtr`.
- i) Элемент `address` элемента `personal` структуры `customerRecord`.
- j) Элемент `address` элемента `personal` структуры, на которую указывает `customerPtr`.
- k) Элемент `city` элемента `personal` структуры `customerRecord`.
- l) Элемент `city` элемента `personal` структуры, на которую указывает `customerPtr`.
- m) Элемент `state` элемента `personal` структуры `customerRecord`.
- n) Элемент `state` элемента `personal` структуры, на которую указывает `customerPtr`.
- o) Элемент `zipCode` элемента `personal` структуры `customerRecord`.
- p) Элемент `zipCode` элемента `personal` структуры, на которую указывает `customerPtr`.

- 16.8. Модифицируйте программу на рис. 16.14, введя в нее эффективное тасование карт (как показано на рис. 16.2). Напечатайте результатирующее состояние колоды в две колонки, как на рис. 16.3. Перед каждой картой напечатайте ее цвет.
- 16.9. Напишите программу, сдвигающую на 4 бита вправо значение целого числа. Программа должна печатать двоичное представление целого до и после операции сдвига. Ваша система заполняет освободившиеся биты нулями, или единицами?
- 16.10. Если ваш компьютер использует 4-х байтовые целые, модифицируйте программу на рис. 16.5, чтобы она работала с 4-х байтовыми целыми.

16.11. Сдвиг влево целого без знака на 1 бит эквивалентен умножению значения на 2. Напишите функцию `power2`, которая принимает два целых параметра `number` и `pow` и вычисляет $\text{number} \ll \text{pow}$

Используйте для расчета операцию сдвига. Программа должна печатать значения как целые и в двоичном представлении.

16.12. Операция сдвига влево может использоваться для упаковки двух значений символов в одной двухбайтовой целой переменной без знака. Напишите программу, которая вводит два символа с клавиатуры и передает их функции `packCharacters`. Для упаковки двух символов в целую переменную типа `unsigned` присвойте переменной первый символ, сдвиньте ее на 8 позиций влево и примените к ней и второму символу операцию поразрядное ИЛИ. Программа должна выводить символы в двоичном представлении до и после их упаковки в переменную типа `unsigned`, чтобы убедиться, что упаковка проведена корректно.

16.13. Используя операцию сдвига вправо, поразрядное И и маску, напишите функцию `unpackCharacters`, которая принимает целую переменную типа `unsigned` из упражнения 16.12 и распаковывает из нее два символа. Чтобы распаковать два символа из двухбайтовой переменной типа `unsigned`, скомбинируйте эту переменную с маской 65280 (11111111 00000000) и сдвиньте результат вправо на 8 битов. Присвойте результат переменной типа `char`. Затем скомбинируйте переменную типа `unsigned` с маской 255 (00000000 11111111). Присвойте результат другой переменной типа `char`. Программа должна печатать переменную типа `unsigned` до распаковки, а затем печатать символы в двоичном представлении, чтобы убедиться, что распаковка проведена корректно.

16.14. Если ваша система использует 4-х байтовые целые, перепишите программу упражнения 16.12, чтобы она упаковывала 4 символа.

16.15. Если ваша система использует 4-х байтовые целые, перепишите функцию `unpackCharacters` из упражнения 16.14, чтобы она распаковывала 4 символа. Создавайте маски, необходимые для распаковки 4-х символов, применяя сдвиг влево на 8 битов значения 255 переменной маски 0, 1, 2 и 3 раза (в зависимости от того, какой бит вы хотите распаковать).

16.16. Напишите программу, которая изменяет на обратную последовательность битов в целом значении без знака. Программа должна запрашивать ввод значения у пользователя и вызывать функцию `reverseBits` для печати битов в обратной последовательности. Печатайте двоичное представление значения до и после изменения последовательности битов, чтобы убедиться, что преобразование произведено правильно.

16.17. Модифицируйте функцию `displayBits`, приведенную на рис. 16.5, чтобы она было переносима между системами, использующими для хранения целых 2 и 4 байта. Подсказка: используйте операцию `sizeof` для определения размера целого на конкретном компьютере.

16.18. Напишите программу, которая вводит символ с клавиатуры и проверяет этот символ каждой из функций библиотеки обработки символов. Программа должна печатать значение, возвращаемое каждой функцией.

16.19. Следующая программа использует функцию **multiple** для определения, кратно ли введенное с клавиатуры целое некоторой целой переменной **x**. Изучите функцию **multiple** и определите значение **x**.

```
// Эта программа определяет, кратно ли значение величине X
#include <iostream.h>
```

```
int multiple(int);

main()
{
    int y;

    cout << "Введите целое между 1 и 32000: ";
    cin >> y;

    if (multiple(y))
        cout << y << " кратно X" << endl;
    else
        cout << y << " не кратно X" << endl;

    return 0;
}

int multiple(int num)
{
    int mask = 1, mult = 1;

    for (int i = 0; i < 10; i++, mask <= 1)
        if ((num & mask) != 0) {
            mult = 0;
            break;
        }
    return mult;
}
```

16.20. Что делает следующая программа?

```
#include <iostream.h>
int mystery(unsigned);
main()
{
    unsigned x;

    cout << "Введите целое: ";
    cin >> x;
    cout << "Результат равен " << mystery(x) << endl;
    return 0;
}

int mystery(unsigned bits)
{
    unsigned mask = 1 << 15, total = 0;
```

```
for (int i = 0; i < 16; i++, bits <= 1)
    if ((bits & mask) == mask)
        ++total;

return total % 2 == 0 ? 1 : 0;
}
```

- 16.21. Напишите программу, которая вводит строку текста функцией-элементом `getline` класса `istream` (см. главу 11) в массив символов `s[100]`. Затем выводит ее в верхнем регистре и в нижнем регистре.
- 16.22. Напишите программу, которая вводит 4 строки, представляющие собой целые числа, преобразует эти строки в целые значения и печатает их сумму.
- 16.23. Напишите программу, которая вводит 4 строки, представляющие собой числа с плавающей точкой, преобразует эти строки в значения типа `double` и печатает их сумму.
- 16.24. Напишите программу, которая вводит с клавиатуры строку текста и подстроку (ключ поиска). Используя функцию `strstr`, найдите первое вхождение подстроки в тексте и присвойте этот адрес переменной `searchPtr` типа `char *`. Если подстрока найдена, напечатайте остальной текст строки, начиная с найденной подстроки. Затем снова используйте `strstr` для определения следующего вхождения подстроки в строку текста. Если второе вхождение найдено, напечатайте остальной текст строки, начиная с этого второго вхождения. Подсказка: второй вызов `strstr` должен содержать `searchPtr + 1` в качестве первого аргумента.
- 16.25. Напишите программу, основанную на программе упражнения 16.24, которая вводит несколько строк текста и одну подстроку (ключ поиска). Используя функцию `strstr` определите общее число вхождений подстроки в эти строки. Напечатайте результат.
- 16.26. Напишите программу, которая вводит несколько строк текста и один символ (ключ поиска). Используя функцию `strchr` определите общее число вхождений этого символа в строки текста.
- 16.27. Напишите программу, основанную на программе упражнения 16.26, которая вводит несколько строк текста и использует функцию `strchr` для определения общего числа вхождений каждой буквы алфавита в этот текст. Буквы в верхнем и нижнем регистрах подсчитываются совместно. Сохраняйте число вхождений каждой буквы в массиве и напечатайте эти значения в табулированном формате после того, как все результаты будут определены.
- 16.28. Таблица в приложении В содержит числовые коды, представляющие все символы набора символов ASCII. Изучите эту таблицу и затем ответьте, верны ли следующие утверждения:
- Буква «A» имеет код меньший, чем буква «B».
 - Цифра «9» имеет код меньший, чем цифра «0».
 - Обычно используемые символы сложения, вычитания, умножения и деления имеет код меньший, чем любая цифра.
 - Цифры имеют код меньший, чем буквы.

- е) Если сортирующая программа сортирует строки в порядке возрастания, то эта программа помещает символ правой скобки перед символом левой скобки.
- 16.29. Напишите программу, которая читает ряд последовательных строк и печатает только те из них, которые начинаются с буквы «b».
- 16.30 Напишите программу, которая читает ряд последовательных строк и печатает только те из них, которые кончаются буквами «ED».
- 16.31. Напишите программу, которая вводит коды ASCII и печатает соответствующие им символы. Модифицируйте эту программу так, чтобы она генерировала все возможные трехразрядные коды в диапазоне от 000 до 255 и пыталась напечатать соответствующие им символы. Что случается при выполнении этой программы.
- 16.32. Используя таблицу символов ASCII из приложения В напишите ваши собственные версии функций обработки символов из таблицы на рис. 16.16.
- 16.33. Напишите ваши собственные версии функций преобразования строк в числа из таблицы на рис. 16.20.
- 16.34. Напишите ваши собственные версии функций поиска в строках (таблица на рис. 16.27).
- 16.35. Напишите ваши собственные версии функций работы с блоками памяти из таблицы на рис. 16.34.
- 16.36. (*Проект: контроль орфографии*) Многие популярные пакеты программ обработки текстов имеют встроенные блоки контроля орфографии. Мы использовали средства контроля орфографии в Microsoft Word 5.0 при подготовке этой книги и обнаружили, что независимо от того, насколько тщательно мы писали главу, Word всегда находил больше орфографических ошибок, чем мы могли обнаружить вручную.

В данном проекте мы просим вас разработать свою собственную утилиту контроля орфографии. Мы предложим ряд упрощений, чтобы помочь вам начать этот проект. В дальнейшем вы можете решить добавить в проект дополнительные возможности. Вы можете решить также, что полезно использовать компьютеризованные словари в качестве источников правильного написания слов.

Почему мы пишем так много слов с орфографическими ошибками? В некоторых случаях это связано с тем, что мы просто не знаем правильное правописание и гадаем, как написать то или иное слово. В некоторых случаях мы переставляем две буквы (например, «умложение» вместо «умолчание»). Иногда мы случайно сдваиваем буквы (например, «удоббно» вместо «удобно»). Иногда мы случайно нажимаем соседнюю клавишу (например, «лень» вместо «день») и т.д.

Спроектируйте и реализуйте программу контроля орфографии на C++. Ваша программа должна обрабатывать массив `wordList` (словарь) символьных строк. Вы можете или ввести эти строки сами, или взять их из компьютерного словаря.

Ваша программа просит пользователя ввести слово. Затем программа проглядывает слова в массиве `wordList`. Если введенное слово в массиве есть, ваша программа должна напечатать «Слово написано правильно».

Если слово не найдено в словаре, ваша программа должна напечатать «Слово написано неправильно». Тогда программа должна попытаться найти другое слово в `wordList`, которое может быть подразумевалось пользователем, написавшим его. Например, вы можете попытаться переставить всеми возможными способами соседние буквы, чтобы обнаружить, что слово «умолчание» имеется в словаре. Конечно, это означает, что ваша программа должна перевернуть все возможные перестановки соседних букв: «муолчание», «уомлчание», «умлочание», «умочлание» и т.д. Когда вы найдете новое слово, которое имеется в `wordList`, напечатайте сообщение типа: «Может быть Вы подразумевали «умолчание»?».

Чтобы улучшить вашу программу проверки орфографии, вы можете добавить в нее реализацию других тестов, таких, как замещение сдвоенных букв одной буквой, и любых других.

17

Препроцессор



Ц е л и

- Научиться использовать директиву препроцессора **#include**, помогающую при разработке больших программ.
- Научиться использовать директиву **#define** для определения макросов и макросов с параметрами.
- Понять, что такое условная компиляция.
- Научиться выводить сообщения об ошибках при условной компиляции.
- Научиться использовать макрос **assert** для проверки значений выражений.

План

- 17.1. Введение
- 17.2. Директива препроцессора `#Include`
- 17.3. Директива препроцессора `#define`: символические константы
- 17.4. Директива препроцессора `#define`: макросы
- 17.5. Условная компиляция
- 17.6. Директивы препроцессора `#error` и `#pragma`
- 17.7. Операции `#` и `##`
- 17.8. Нумерация строк
- 17.9. Предопределенные символические константы
- 17.10. Макрос `assert`

Резюме • Терминология • Типичные ошибки программирования • Хороший стиль программирования • Советы по повышению эффективности • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

17.1. Введение

В этой главе вы познакомитесь с *препроцессором*. Обработка программы препроцессором происходит перед ее компиляцией. На этом этапе предварительной обработки вы можете выполнить следующие действия: включить в компилируемый файл другие файлы, определить *символические константы* и *макросы*, задать режим *условной компиляции* программного кода и *условного выполнения директив препроцессора*. Все директивы препроцессора начинаются с символа `#` и до начала директивы в строке могут находиться только символы пробела.

17.2. Директива препроцессора #include

Директива препроцессора `#include` уже использовалась в тексте этой книги. Директива `#include` применяется для включения копии указанного в директиве файла в то место, где находится эта директива. Существуют две формы директивы `#include`:

```
#include <имя_файла>
#include "имя_файла"
```

Различие между ними заключается в методе поиска препроцессором включаемого файла. Если имя файла заключено в угловые скобки (`< и >`), как это делается для включения заголовочных файлов *стандартной библиотеки*, то последовательность поиска препроцессором заданного файла в каталогах определяется используемой системой программирования; обычно просматривается заранее установленный список каталогов. Если имя файла заключено в кавычки, препроцессор сначала ищет файл в том каталоге, где находится компилируемый файл, а затем продолжает поиск тем же способом, что и при угловых скобках. Такой формой обычно пользуются для включения заголовочных файлов, определяемых программистом.

Директива `#include` обычно используется для включения стандартных заголовочных файлов, таких, как `iostream.h` и `iomanip.h`. Директива `#include` используется также при работе с программами, состоящими из нескольких исходных файлов, которые должны компилироваться вместе. Заголовочные файлы содержат объявления и определения, общие для различных программных файлов, и поэтому часто создаются и включаются в файлы программ. В качестве таких объявлений и определений выступают классы, структуры, объединения, перечислимые типы и прототипы функций.

17.3. Директива препроцессора #define: символические константы

Директива препроцессора `#define` создает символические константы, обозначаемые идентификаторами, и макросы — операции, обозначаемые символьными строками. Формат директивы препроцессора `#define` следующий

```
#define идентификатор замещающий_текст
```

После появления этой строки в файле все встретившиеся далее в тексте программы имена, совпавшие с элементом директивы *идентификатор*, будут автоматически заменены на указанный в директиве *замещающий_текст* прежде, чем начнется компиляция программы. Например, после задания директивы

```
#define PI 3.14159
```

все последующие вхождения в текст программы символической константы `PI` будут заменены на численную константу `3.14159`. Символические константы дают возможность программисту присвоить константе имя и использовать его далее в программе. Если возникнет необходимость изменить значение константы во всей программе, для этого достаточно будет внести только одно изменение в директиву препроцессора `#define` и перекомпилировать программу; значение

константы будет изменено по всей программе автоматически. Отметьте: *все, что находится справа от имени символьической константы, является замещающим ее текстом*. Например, после выполнения директивы `#define PI =3.14159` препроцессор заменит все имена `PI` на текст `=3.14159`. Многие логические и синтаксические ошибки возникают по причине непонимания этого правила. Переопределение значения символьической константы является также, обычным источником ошибок. Заметим, что в C++ отдается предпочтение использованию именованных переменных типа `const`, а не символьических констант. Константные переменные являются данными определенного типа и их имена видны отладчику. А если используется символьическая константа, то после того, как символьическая константа была заменена на соответствующий текст, только этот текст и будет виден отладчику. Недостатком переменных типа `const` является то, что им требуется память в объеме, соответствующем их типу, для хранения своего значения, тогда как для символьических констант не требуется никакой дополнительной памяти.

Хороший стиль программирования 17.1

Использование ясных по смыслу имен для символьических констант улучшает понимание текста программы, делает текст самодокументируемым.

17.4. Директива препроцессора `#define`: макросы

Макрос, определяемый директивой препроцессора `#define`, это символьическое имя некоторых операций. Как и в случае символьических констант, идентификатор макроса заменяется на замещающий текст до начала компиляции программы. Макросы могут быть определены с параметрами или без них. Макросы без параметров обрабатываются подобно символьическим константам. Если макрос имеет параметры, то сначала в замещающий текст подставляются значения параметров, а затем уже этот расширенный макрос подставляется в текст вместо идентификатора макроса и списка его параметров.

Рассмотрим следующий макрос с одним параметром для расчета площади круга

```
#define CIRCLE_AREA(x) ( PI * (x) * (x) )
```

Везде в файле, где появится идентификатор `CIRCLE_AREA(x)`, значение аргумента `x` будет использовано для замены `x` в замещающем тексте, символьическая константа `PI` будет заменена ее значением (определенным выше) и этот расширенный текст макроса будет использован для замещения. Например, оператор с макросом в тексте программы

```
area = CIRCLE_AREA(4);
```

примет вид:

```
area = ( 3.14159 * (4) * (4) );
```

Поскольку это выражение состоит только из констант, его значение будет вычислено во время компиляции и полученный результат будет присвоен переменной `area` во время выполнения программы. Круглые скобки вокруг каждого включения параметра `x` в тексте макроса и вокруг всего выражения применяются для того, чтобы обеспечить соответствующий порядок вычис-

лений в том случае, когда аргументом макроса является выражение. Например, при раскрытии макроса в операторе

```
area = CIRCLE_AREA(c + 2);
```

оператор примет следующий вид:

```
area = ( 3.14159 * (c + 2) * (c + 2) );
```

и вычисления будут выполняться правильно, потому что скобки обеспечили правильную последовательность вычислений. Если в определении макроса опустить все круглые скобки, то после расширения макроса рассматриваемый оператор будет иметь следующий вид:

```
area = 3.14159 * c + 2 * c + 2;
```

С учетом приоритетов операций, вычисление значения выражения будет сделано в следующем порядке

```
area = (3.14159 * c) + (2 * c) + 2;
```

и будет получен неверный результат.

Типичная ошибка программирования 17.1

Если параметры макроса в выражении для подстановки не заключены в круглые скобки, то при подстановке могут возникнуть ошибки последовательности вычислений.

Вычисления, выполняемые макросом **CIRCLE_AREA**, можно выполнять при помощи функции. Функция **circleArea**

```
double circleArea(double x) { return 3.14159 * x * x; }
```

выполняет те же вычисления, что и макрос **CIRCLE_AREA**. Недостатком использования этой функции является то, что на ее вызов должны быть затрачены некоторые ресурсы. Преимущества использования макроса **CIRCLE_AREA** состоят в том, что вычисления, выполняемые макросом, непосредственно помещаются в текст программы и не приводят к дополнительным накладным расходам, связанным с вызовом функции; при этом текст программы остается легко читаемым, потому что имя **CIRCLE_AREA** говорит само за себя. Недостаток этого макроса в том, что значение аргумента вычисляется дважды. И конечно, макрос будет расширяться при каждом своем появлении в тексте программы. Если макрос большой, то это приводит к увеличению размера программы. Таким образом, нужно искать компромисс между быстрым действием и размером программы (если у вас маленький диск). Заметим, что предпочтительнее использовать встраиваемые **inline** функции (см. главу 3), которые обеспечивают эффективность макросов и преимущества функций при разработке программного обеспечения.

Совет по повышению эффективности 17.1

Макросы могут использоваться для замены вызовов функций на встраиваемый в программу код. Это устраниет накладные расходы по вызову функции. Но предпочтительнее все же использовать встраиваемые функции.

Рассмотрим следующее определение макроса с двумя параметрами, вычисляющего площадь прямоугольника:

```
#define RECTANGLE_AREA(x, y) ( (x) * (y) )
```

Везде в программе, где встречается идентификатор **RECTANGLE_AREA(x, y)**, параметры **x** и **y** в замещающем тексте макроса будут заменяться значениями соответствующих аргументов и будет выполняться расширение макроса. Например, макрос в операторе

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

будет расширен до следующего:

```
rectArea = ( (a + 4, * (b + 7) );
```

Значение полученного выражения будет вычисляться и результат будет присваиваться переменной **rectArea**.

Выражением для подстановки макроса или символьической константы является любой текст в строке директивы **#define**, следующий за идентификатором. Если текст для подстановки макроса или символьической константы длиннее, чем остаток строки, то в конце строки можно поставить символ обратного слэша (\), указывающий на то, что замещающий текст продолжается на следующей строке.

Определения символьических констант и макросов могут быть аннулированы при помощи директивы **#undef**. Директива **#undef** отменяет определение символьической константы или макроса. Область действия символьической константы или макроса начинается с места их определения и заканчивается явным их аннулированием директивой **#undef** или концом файла. После аннулирования соответствующий идентификатор может быть снова использован в директиве **#define**.

Функции в стандартной библиотеке иногда определяются как макросы на основе других библиотечных функций. Например, в заголовочном файле **<stdio.h>** обычно определяется следующий макрос

```
#define getchar() getc(stdin)
```

Макроопределение **getchar** не использует функцию **getc** для получения символа из стандартного потока ввода. Функция **putchar** в **<stdio.h>** и функции обработки символов в **<cctype.h>** часто также вводятся как макросы. Обратите внимание, что выражения с побочными эффектами (например, изменяющие значения переменных) не должны передаваться макросу в качестве аргументов, потому что значения аргументов макроса могут вычисляться в его теле неоднократно.

17.5. Условная компиляция

Условная компиляция дает возможность программисту управлять выполнением директив препроцессора и компиляцией программного кода. Каждая условная директива препроцессора вычисляет значение целочисленного константного выражения. Операции преобразования типов, операция **sizeof** и константы перечислимого типа не могут участвовать в выражениях, вычисляемых в директивах препроцессора.

Условная директива препроцессора во многом похожа на оператор `if`. Рассмотрим следующий фрагмент кода:

```
#if !defined(NULL)
    #define NULL 0
#endif
```

Эти директивы определяют, не была ли определена ранее константа `NULL`. Выражение `defined(NULL)` дает значение 1, если `NULL` определена, и 0 в противном случае. Если результат равен 0, то выражение `!defined(NULL)` дает значение 1 и в следующей строке производится определение константы `NULL`. В противном случае директива `#define` пропускается. Каждая директива `#if` должна заканчиваться своим `#endif`. Директивы `#ifdef` и `#ifndef` являются сокращением выражений `#if defined(имя)` и `#if !defined(имя)`. Можно использовать сложные конструкции условных директив препроцессора при помощи директив `#elif` (эквивалент `else if` в структуре `if`) и `#else` (эквивалент `else` в структуре `if`).

При разработке программы программисты часто находят удобным для себя временно «закомментировать» большие фрагменты кода и не компилировать их. Если в коде используются комментарии в стиле C, то знаки комментария `/*` и `*/` не помогут решить эту задачу. В таком случае программист может использовать следующую конструкцию директив препроцессора

```
#if 0
    Фрагмент кода, который не нужно компилировать
#endif
```

Для того, чтобы этот фрагмент включить в процесс компиляции, достаточно заменить 0 в приведенной конструкции на 1.

Условная компиляция обычно используется как средство отладки. Многие системы программирования на C++ предоставляют разработчику *отладчики* программ. Однако, сначала нужно изучить этот отладчик и научиться его использовать, что часто вызывает затруднения у студентов и начинающих программистов. Вместо отладчика можно использовать операторы вывода значений переменных, что позволяет контролировать процесс выполнения программы. Эти операторы «обкладываются» условными директивами препроцессора и компилируются только пока процесс отладки программы не завершен. Например, в следующем фрагменте

```
#ifdef DEBUG
    cerr << "Переменная x = " << x << endl;
#endif
```

оператор вывода в поток `cerr` будет компилироваться только в случае, если символическая константа `DEBUG` была определена (директивой `#define DEBUG`) до директивы `#ifdef DEBUG`. После завершения процесса отладки директива `#define DEBUG` может быть просто удалена из исходного файла и операторы вывода, нужные только для целей отладки, будут игнорироваться во время компиляции. В больших программах, возможно, потребуется определять несколько различных символьических констант, которые могут управлять условной компиляцией различных частей исходного файла.

Типичная ошибка программирования 17.2

К ошибкам приводит добавление условно компилируемых операторов вывода, нужных для целей отладки, в таких местах программы, где по синтаксису C++ ожидается только один оператор. В этих случаях условно компилируемый оператор нужно включать в составной оператор. Тогда при компиляции программы с отладочными операторами поток управления программы не будет изменяться.

17.6. Директивы препроцессора `#error` и `#pragma`

Директива препроцессора `#error` имеет следующий синтаксис
`#error` лексемы

Директива печатает сообщение об ошибке, зависящее от используемой системы и содержащее заданные в директиве лексемы. Лексемы представляют собой группы символов, отделяемые друг от друга пробелами. Например, директива

```
#error 1 - Ошибка выхода из диапазона
```

содержит 6 лексем. В Borland C++, например, когда обрабатывается директива `#error`, лексемы директивы выводятся в качестве сообщения об ошибке, предварительная обработка завершается и программа не компилируется.

Директива `#pragma`

`#pragma` лексемы

вызывает действия, зависящие от используемой системы. Директива `#pragma`, не распознанная системой, игнорируется. Borland C++, например, распознает несколько указаний компилятору — прагм, которые дают возможность программисту полнее использовать возможности системы программирования Borland C++. За подробной информацией по директивам `#error` и `#pragma` обращайтесь к документации по вашей системе программирования на C++.

17.7. Операции `#` и `##`

Операции препроцессора `#` и `##` доступны в C++ и С ANSI. Операция `#` преобразует подставляемую лексему в строку символов, взятую в кавычки. Рассмотрим следующее макроопределение:

```
#define HELLO(x) cout << "Hello, " #x << endl
```

Тогда, если в программе встретится макрос `HELLO(John)`, он будет расширяться до

```
cout << "Hello, " "John" << endl
```

Строка `"John"` заменила параметр `#x` в замещающем тексте. Строки, разделенные символами пробела, сцепляются (склеиваются) во время предварительной обработки, так что вышеприведенный оператор эквивалентен оператору

```
cout << "Hello, John" << endl
```

Обратите внимание, что операция `#` должна использоваться только в макросе с параметрами, потому что операция `#` применяется к параметру макроса.

Операция `##` выполняет конкатенацию (сплеливание, склеивание) двух лексем. Рассмотрим следующее макроопределение:

```
#define TOKENCONCAT(x, y) x ## y
```

Когда в программе встречается макрос `TOKENCONCAT`, его аргументы склеиваются и полученное выражение используется для замещения идентификатора макроса. Например, `TOKENCONCAT (0, K)` будет замещаться в тексте программы на `0K`. Операция `##` должна иметь два операнда.

17.8. Нумерация строк

Директива препроцессора `#line` задает целочисленное константное начальное значение номера строки для нумерации следующих за директивой строк исходного текста программы. Директива

```
#line 100
```

задает начальное значение номера строки, равное 100, и все последующие строки исходного текста программы будут нумероваться, начиная с этого номера. В директиву `#line` может быть включено имя файла. Директива

```
#line 100 "file1.c"
```

изменит нумерацию последующих строк программы, которая начнется со значения 100, и компилятор во всех своих сообщениях будет ссылаться на файл с именем `"file1.c"`. Директива обычно используется для того, чтобы сделать сообщения о синтаксических ошибках и предупреждения компилятора более удобными для понимания. Номера строк не добавляются в исходный файл.

17.9. Предопределенные символические константы

Имеются пять *предопределенных символьических констант* (см. рис. 17.1). Идентификаторы каждой из предопределенных символьических констант начинаются и заканчиваются *двумя* символами подчеркивания. Эти идентификаторы и идентификатор `defined` (описанный в разделе 17.5) не могут использоваться в директивах `#define` или `#undef`.

Символическая константа	Назначение
<code>__LINE__</code>	Номер текущей строки исходного текста программы (целочисленная константа).
<code>__FILE__</code>	Предполагаемое имя исходного файла (строка).
<code>__DATE__</code>	Дата компиляции исходного файла (строка в формате "Ммм dd yyyy", например, "Jan 19 1994").
<code>__TIME__</code>	Время компиляции исходного файла (символьная строка формата "hh:mm:ss")
<code>__STDC__</code>	Целочисленная константа 1. Используется для указания, что данная реализация удовлетворяет стандартам ANSI.

Рис. 17.1. Предопределенные символические константы

17.10. Макрос assert

Макрос `assert`, определенный в заголовочном файле `<assert.h>`, выполняет проверку значения выражения. Если значение выражения 0 (ложь), то макрос `assert` выводит сообщение об ошибке и вызывает функцию `abort` (из библиотеки утилит общего назначения `<stdlib.h>`), которая завершает выполнение программы. Этот макрос удобно использовать при отладке для проверки того, что переменная имеет правильное значение. Например, предположим, что переменная `x` в программе не должна принимать значение большее, чем 10. В этом случае макрос `assert` можно использовать для проверки значения `x` и вывода сообщения об ошибке, если значение `x` вышло из допустимого диапазона. Оператор может выглядеть следующим образом:

```
assert(x <= 10);
```

Если при выполнении этого оператора `x` будет иметь значение, большее, чем 10, то программа выдаст сообщение об ошибке, содержащее номер строки и имя файла, после чего завершит свою работу. Теперь программист для того, чтобы найти ошибку, может сконцентрировать свое внимание на области кода, выдавшей сообщение. После того, как в тексте программы объявляется символьическая константа `NDEBUG`, все последующие вызовы макроса `assert` будут игнорироваться. Таким образом, когда все они будут больше не нужны (т.е. когда отладка закончена), в начале программы достаточно добавить строку

```
#define NDEBUG
```

вместо того, чтобы удалять каждый макрос `assert` вручную.

Резюме

- Все директивы препроцессора начинаются с символа `#`.
- В строке программы перед директивой препроцессора могут находиться только символы пробела.
- Директива `#include` включает в текст программы копию указанного файла. Если имя файла заключено в кавычки, препроцессор начинает поиск включаемого файла с того каталога, в котором находится компилируемый файл. Если имя файла заключено в угловые скобки (`<` и `>`), то последовательность поиска препроцессором заголовочного файла в каталогах определяется используемой системой программирования.
- Директива препроцессора `#define` используется для определения символьической константы или макроса.
- Символическая константа — это имя, присвоенное константе.
- Макрос — это некоторая операция, определенная директивой препроцессора `#define`. Макросы могут иметь, или не иметь параметры.
- Замещающий текст макроса или символьической константы — это весь текст, следующий за идентификатором в строке директивы `#define`. Если текст для подстановки макроса или символьической константы длиннее, чем остаток строки, то в конце строки можно поставить символ обратного слэша (`\`), указывающий на то, что замещающий текст продолжается на следующей строке.

- Символические константы и макросы могут быть аннулированы с помощью директивы препроцессора `#undef`. Директива `#undef` отменяет определение символической константы или макроса.
- Область действия символической константы или макроса начинается с места их определения и заканчивается явным их аннулированием директивой `#undef` или концом файла.
- Условная компиляция дает возможность программисту управлять выполнением директив препроцессора и компиляцией программного кода.
- Условные директивы препроцессора вычисляют значения целочисленных константных выражений. Операции преобразования типов, операция `sizeof` и константы перечислимого типа не могут участвовать в выражениях, вычисляемых в директивах препроцессора.
- Каждая директива `#if` должна заканчиваться директивой `#endif`.
- Директивы `#ifdef` и `#ifndef` являются сокращением выражений `#if defined(имя)` и `#if !defined(имя)`.
- С помощью директив `#elif` (эквивалент `else if` в структуре `if`) и `#else` (эквивалент `else` в структуре `if`) можно создавать сложные конструкции условных директив препроцессора.
- Директива `#error` печатает сообщение об ошибке, зависящее от используемой системы и содержащее заданные в директиве лексемы, после чего завершает предварительную обработку и компиляцию.
- Директива `#pragma` вызывает действия, зависящие от используемой системы. Директива `#pragma`, не распознанная системой, игнорируется.
- Операция `#` преобразует подставляемую лексему в строку символов, взятую в кавычки. Операция `#` должна использоваться только в макросе с параметрами, потому что операция `#` применяется к параметру макроса.
- Операция `##` выполняет конкатенацию (сплление, склеивание) двух лексем. Операция `##` должна иметь два операнда.
- Директива препроцессора `#line` задает целочисленное константное начальное значение номера строки для нумерации следующих за директивой строк исходного текста программы.
- Имеется пять предопределенных символьических констант. Константа `_LINE_` обозначает номер текущей строки исходного текста (целое число). Константа `_FILE_` — это предполагаемое имя файла (строка). Константа `_DATE_` обозначает дату компиляции исходного файла (строка). Константа `_TIME_` обозначает время компиляции исходного файла (строка). Константа `_STDC_` 1 предназначена для указания, что данная реализация удовлетворяет стандартам ANSI. Обратите внимание, что каждая из предопределенных символьических констант начинается и заканчивается двумя символами подчеркивания.
- Макрос `assert`, определенный в заголовочном файле `assert.h`, проверяет значение выражения. Если значение выражения равно 0 (ложь), то макрос `assert` выводит сообщение об ошибке и вызывает функцию `abort`, завершающую выполнение программы.

Терминология

#define	stdlib.h
#elif	аргумент
#else	директива препроцессора
#endif	заголовочные файлы
#error	стандартной библиотеки
#if	заголовочный файл
#ifdef	замещающий текст
#ifndef	макрос
#include "имя_файла"	макрос с аргументами
#include <имя_файла>	область действия символьической
#line	константы или макроса
#pragma	отладчик
#undef	предопределенные символьические
\ (обратный слэш) символ	константы
продолжения макроса в новой	препроцессор
строке	препроцессорная операция
<u>DATE</u>	преобразования в строку #
<u>FILE</u>	препроцессорная операция склейки
<u>LINE</u>	лексем ##
<u>STDC</u>	расширение макроса
<u>TIME</u>	символьская константа
abort	условная компиляция
assert	условное выполнение директив
assert.h	препроцессора
stdio.h	

Хороший стиль программирования

17.1. Использование ясных по смыслу имен для символьических констант улучшает понимание текста программы, делает текст самодокументируемым.

Советы по повышению эффективности

17.1. Макросы могут использоваться для замены вызовов функций на встраиваемый в программу код. Это устраняет накладные расходы по вызову функции. Но предпочтительнее все же использовать встраиваемые функции.

Упражнения для самопроверки

17.1. Заполните пробелы в следующих утверждениях:

- Каждая директива препроцессора должна начинаться с _____.
- Конструкции условной компиляции могут быть расширены до конструкций с множественным выбором при помощи директив _____ и _____.
- Директива _____ создает макросы и символьические константы.

- d) В строке, содержащей директиву, перед директивой препроцессора могут появляться только символы _____.
- e) Директива _____ аннулирует символьические константы и макросы.
- f) Директивы _____ и _____ являются сокращенными обозначениями директив `#if defined (имя)` и `#if !defined (имя)`.
- g) _____ дает возможность программисту управлять выполнением директив препроцессора и трансляцией программного кода.
- h) Макрос _____ выводит сообщение и завершает выполнение программы, если значение выражения макроса равно 0.
- i) Директива _____ включает один файл в другой.
- j) Операция _____ склеивает два своих аргумента.
- k) Операция _____ преобразует свой операнд в строку символов.
- l) Символ _____ указывает на то, что замещающий текст символьской константы или макроса продолжается в следующей строке.
- m) Директива _____ устанавливает начальный номер для нумерации последующих строк файла исходного текста.

17.2. Напишите программу, выводящую значения предопределенных символьических констант, перечисленных на рис. 17.1.

17.3. Для выполнения приведенных ниже операций напишите соответствующую директиву препроцессора:

- a) Определите символьическую константу YES со значением 1.
- b) Определите символьическую константу NO со значением 0.
- c) Включите заголовочный файл `common.h`. Этот файл находится в том же каталоге, что и компилируемый файл.
- d) Перенумеруйте последующие строки в файле, начиная с номера 3000.
- e) Если символьическая константа TRUE определена, аннулируйте ее, а затем переопределите на 1. Не используйте директиву `#ifdef`.
- f) Если символьическая константа TRUE определена, аннулируйте ее, а затем переопределите на 0. Используйте директиву препроцессора `#ifdef`.
- g) Если символьическая константа ACTIVE не равна 0, определите символьскую константу INACTIVE, равную 0. В противном случае определите INACTIVE как 1.
- h) Определите макрос CUBE_VOLUME, который вычисляет объем куба. Макрос получает один аргумент.

Ответы на упражнения для самопроверки

- 17.1. a) #. b) #elif, #else. c) #define. d) пробела. e) #undef. f) #ifdef, #ifndef. g) Условная компиляция. h) assert. i) #include. j) ##. k) #. l) \. m) #line.
- 17.2. `#include <iostream.h>`
`main ()`

```

{
    cout << "____LINE____ = " << ____LINE____ << endl;
    cout << "____FILE____ = " << ____FILE____ << endl;
    cout << "____DATE____ = " << ____DATE____ << endl;
    cout << "____TIME____ = " << ____TIME____ << endl;
    cout << "____STDC____ = " << ____STDC____ << endl;
    return 0;
}

____LINE____ = 5
____FILE____ = macros.c
____DATE____ = Mar 08 1994
____TIME____ = 10:23:47
____STDC____ = 1

```

17.3. 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 ACTIVE
 #define INACTIVE 0
 #else
 #define INACTIVE 1
 #endif`
- h) `#define CUBE_VOLUME(x) ((x) * (x) * (x))`

Упражнения

17.4. Напишите программу, в которой определите макрос с одним аргументом, при помощи которого можно было бы вычислять объем сферы. Программа должна рассчитать объем сферы, радиус которой изменяется от 1 до 10, и выдать результаты в табличной форме. Формула для объема сферы:

$$(4/3) * \pi * r^3$$

где π равно 3.14159.

17.5. Составьте программу, которая выводит на экран следующий результат:

Сумма x и у равна 13

В программе должен определяться макрос **SUM** с двумя аргументами **x** и **y**, который и производит вывод указанной строки.

- 17.6. Напишите программу, которая использует макрос **MINIMUM2** для определения меньшего из двух чисел. Данные вводите с клавиатуры.
- 17.7. Напишите программу, в которой задайте макрос **MINIMUM3**, определяющий меньшее из трех чисел. В процессе нахождения наименьшего числа макрос **MINIMUM3** должен использовать макрос **MINIMUM2** из упражнения 17.6. Данные должны вводится с клавиатуры.
- 17.8. Напишите программу, которая использует макрос **PRINT** для вывода значения строки.
- 17.9. Напишите программу, в которой для вывода элементов целочисленного массива используется макрос **PRINTARRAY**. Макрос должен получать массив и число элементов массива как аргументы.
- 17.10. Напишите программу, которая использует макрос **SUMARRAY** для вычисления значения суммы элементов целочисленного массива. Макрос должен получать массив и число элементов массива как аргументы.
- 17.11. Выполните упражнения 17.4 и 17.10, используя вместо макросов встроенные функции.

18

Другие темы



Ц е л и

- Научиться переназначать ввод с клавиатуры на ввод из файла.
- Научиться переназначать вывод на экран на вывод в файл.
- Научиться работать с функциями, имеющими списки параметров переменной длины.
- Научиться обрабатывать аргументы командной строки.
- Научиться задавать типы численных констант.
- Научиться обрабатывать непредусмотренные события.
- Познакомиться с динамическим выделением памяти для массивов в стиле С.
- Научиться изменять размер динамически выделяемой памяти в стиле С.

План

- 18.1. Введение
- 18.2. Переназначение ввода-вывода в системах UNIX и DOS
- 18.3. Списки параметров переменной длины
- 18.4. Использование аргументов командной строки
- 18.5. Замечания по компиляции программ, состоящих из нескольких исходных файлов
- 18.6. Завершение программы при помощи функций *exit* и *atexit*
- 18.7. Спецификатор типа *volatile*
- 18.8. Сuffixы целочисленных и вещественных констант
- 18.9. Обработка сигналов
- 18.10. Динамическое выделение памяти: функции *calloc* и *realloc*
- 18.11. Безусловный переход: оператор *goto*
- 18.12. Объединения
- 18.13. Спецификации связывания
- 18.14. Заключительные замечания

Резюме • Терминология • Типичные ошибки программирования • Советы по повышению эффективности • Замечания по мобильности • Замечания по технике программирования • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

18.1. Введение

Эта глава охватывает несколько тем повышенной сложности, обычно не включаемых в курсы для начинающих. Многие из возможностей, обсуждаемых здесь, являются специфическими для конкретных операционных систем, таких, например, как UNIX и DOS.

18.2. Переназначение ввода-вывода в системах UNIX и DOS

Обычно ввод данных в программу производится с клавиатуры (стандартный ввод), а вывод из программы направляется на экран монитора (стандартный вывод). Во многих операционных системах, в частности, в UNIX и DOS, имеется возможность *переназначить* (переадресовать) ввод на ввод из файла, а вывод — на вывод в файл. Обе формы переназначения выполняются без использования процедур обработки файлов стандартной библиотеки.

Имеется несколько способов переназначения ввода-вывода в командной строке UNIX. Рассмотрим программу **sum**, которая принимает вводимые по одному целые числа и суммирует введенные значения; это продолжается до тех пор, пока программа не получит сигнал об окончании файла, после чего она выводит значение суммы. Обычно пользователь вводит целые числа с клавиатуры и заканчивает ввод нажатием комбинации клавиш «конец файла». Используя переназначение ввода, данные можно ввести из файла. Например, если данные находятся в файле **input**, то следующая командная строка

```
$ sum < input
```

вызовет на исполнение программу **sum**, а знак *переназначения ввода* (<) укажет программе, что данные для ввода находятся в файле **input**, а не будут вводиться с клавиатуры. Переназначение ввода в операционной системе DOS выполняется точно так же.

Обратите внимание, что символ \$ — это подсказка командной строки в UNIX (некоторые системы UNIX используют в качестве символа подсказки символ %). Студенты часто путаются и не понимают, что переназначение является возможностью операционной системы, а не очередной особенностью C++.

Второй метод переназначения ввода — это *конвейерная обработка*. Знак *конвейера* () переадресует вывод одной программы на ввод другой программы. Предположим, что программа **random** выдает последовательность случайных целых чисел; вывод программы **random** может быть перенаправлен непосредственно в программу **sum** при помощи следующей командной строки UNIX

```
$ random | sum
```

В результате программа **sum** вычислит сумму целых чисел, которые вывела программа **random**. Конвейерная обработка может выполняться в UNIX и DOS.

Вывод программы может быть переназначен в файл. Для этого используется знак *переназначения вывода* (>) (один и тот же знак используется и в UNIX, и в DOS). Например, чтобы переназначить вывод программы **random** в файл **out**, используйте строку

```
$ random > out
```

В заключение отметим, что вывод программы может быть добавлен в конец существующего файла, если воспользоваться знаком *добавления вывода* (>>) (один и тот же знак используется и в UNIX, и в DOS). Например, чтобы добавить вывод из программы **random** к содержимому файла **out**, со-

зданного в результате выполнения предыдущей командной строки, используйте командную строку

```
$ random >> out
```

18.3. Список параметров переменной длины

Можно создать функции, которые при вызове получают неопределенное число аргументов. Многоточие (...) в прототипе функции указывает на то, что функция имеет список параметров переменной длины и любого типа. Обратите внимание на то, что многоточие должно всегда находиться в конце списка параметров и по крайней мере один параметр должен быть задан по имени.

Макросы и определения из заголовочного файла *stdarg.h*, посвященного спискам параметров переменной длины (см. рис. 18.1), обеспечивают возможности, необходимые при построении функций со списками параметров переменной длины. В программе на рис. 18.2 показан пример функции *average*, которая получает переменное число параметров. Первый параметр функции *average* — число значений, которые будут усредняться.

Идентификатор	Назначение
va_list	Тип, предназначенный для хранения информации, необходимой макросам va_start , va_arg и va_end . Для обработки параметров из списка параметров переменной длины необходимо объявлять объект типа va_list .
va_start	Этот макрос должен вызываться перед обработкой списка переменного числа параметров. Макрос инициализирует объект типа va_list для последующего использования его макросами va_arg и va_end .
va_arg	Макрос, расширяющийся до выражения, которое имеет тип и значение следующего параметра в списке параметров переменной длины. Каждый вызов va_arg изменяет значение объекта типа va_list таким образом, что он ссылается на следующий по списку параметр.
va_end	Макрос, который обеспечивает нормальное возвращение из функции, в которой для обработки списка параметров переменной длины использовался макрос va_start .

Рис. 18.1. Тип данных и макросы, определяемые в заголовочном файле **stdarg.h**

Функция *average* использует все определения и макросы файла *stdarg.h*. В функции *average* объект *ap* типа **va_list** используется макросами **va_start**, **va_arg** и **va_end** для обработки списка параметров переменной длины. Функция начинает обработку с вызова **va_start** для инициализации объекта *ap*, используемого далее макросами **va_arg** и **va_end**. В макрос **va_start** передается два аргумента: объект *ap* и идентификатор самого правого параметра в списке параметров перед многоточием — в нашем примере это переменная *i* (**va_start** использует *i*, чтобы определить начало списка аргументов переменной длины). Затем, функция *average* последовательно прибавляет к значению переменной *total* значения параметров из списка параметров переменной длины. Значение, которое будет добавлено к *total*, извлекается из списка параметров при помощи вызова макроса **va_arg**. В макрос **va_arg** передаются два аргумента: объект *ap* типа **va_list** и тип следующего ожидаемого в списке параметров значения, в нашем случае — тип **double**. Макрос

возвращает значение параметра. В завершение своей работы функция **average** вызывает макрос **va_end** с объектом **ap** в качестве аргумента, который обеспечивает нормальное возвращение из **average** в функцию **main**. Полученное среднее значение возвращается функции **main**.

```
// Использование списка параметров переменной длины
#include <iostream.h>
#include <iomanip.h>
#include <stdarg.h>

double average(int, ...);

main()
{
    double w = 37.5, x = 22.5, y = 1.7, z = 10.2;

    cout << setiosflags(ios::fixed | ios::showpoint)
        << setprecision(1) << "w = " << w << endl << "x = " << x
        << endl << "y = " << y << endl << "z = " << z << endl;
    cout << setiosflags(ios::fixed | ios::showpoint)
        << setprecision(3) << endl
        << "Среднее значение w и x равно "
        << average(2, w, x) << endl
        << "Среднее значение w, x и у равно "
        << average(3, w, x, y)
        << endl << "Среднее значение w, x, у и z равно "
        << average(4, w, x, y, z) << endl;

    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
```

```
Среднее значение w и x равно 30.000
Среднее значение w, x и у равно 20.567
Среднее значение w, x, у и z равно 17.975
```

Рис. 18.2. Использование списка параметров переменной длины

Типичная ошибка программирования 18.1

Размещение многоточия в середине списка параметров функции. Многоточие может располагаться только в конце списка параметров.

18.4. Использование аргументов командной строки

Во многих операционных системах, в частности, в DOS и UNIX, имеется возможность передачи аргументов в функцию `main` из командной строки, если в список параметров `main` включить параметры `int argc` и `char *argv[]`. Параметр `argc` получает значение, равное количеству аргументов командной строки. Параметр `argv` — это массив строк, в который помещаются значения аргументов командной строки. Обычно аргументы командной строки используются для задания опций выполнения программы и передачи программе имен файлов.

В программе на рис. 18.3 выполняется посимвольное копирование одного файла в другой. Исполняемый файл этой программы называется `copy`. Командная строка при вызове программы `copy` в системе UNIX выглядит следующим образом

```
$ copy input output
```

Эта командная строка вызывает копирование файла `input` в файл `output`. Если при выполнении программы значение параметра `argc` не будет равно 3 (лексема `copy` учитывается как один из параметров), то программа выдает сообщение об ошибке и завершает свою работу. В противном случае, массив `argv` будет содержать строки `"copy"`, `"input"` и `"output"`. Второй и третий аргументы командной строки используются программой в качестве имен файлов. Файлы открываются при создании объекта `inFile` класса `ifstream` и объекта `outFile` класса `ofstream`. Если оба файла открыты успешно, символычитываются из файла `input` функцией-элементом `get` и записываются в файл `output` функцией-элементом `put` до достижения конца файла `input`. После этого программа завершается. Результирующий файл будет точной копией файла `input`. Нужно отметить, что не все операционные системы поддерживают параметры командной строки так же просто, как UNIX и DOS. ОС Macintosh и VMS, например, требуют специальных установок для обработки аргументов командной строки. Обращайтесь к руководствам по вашей операционной системе за информацией относительно обработки аргументов командной строки.

18.5. Замечания по компиляции программ, состоящих из нескольких исходных файлов

Как ранее уже говорилось в этой книге, можно создавать программы, которые состоят из нескольких исходных файлов (см. главу 6, «Классы и абстрагирование данных»). При создании таких программ имеются некоторые ограничения. Например, описание функции должно полностью содержаться в одном файле, оно не может распространяться на два или большее число файлов.

```
// Использование аргументов командной строки
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
    if (argc != 3)
        cout << "Применяйте формат: "
            << "сoru вхoднoй_файл выxоднoй_файл" << endl;
    else {
        ifstream inFile(argv[1], ios::in);
        if (!inFile)
            cout << argv [1] << " нельзя открыть" << endl;

        ofstream outFile(argv[2], ios::out);
        if (!outFile)
            cout << argv[2] << " нельзя открыть" << endl;

        while ( !inFile.eof() )
            outFile.put( (char) inFile.get() );
    }

    return 0;
}
```

Рис. 18.3. Использование аргументов командной строки

В главе 3 мы ввели понятия класса памяти и области действия. Мы узнали, что переменные, объявленные вне определения какой-нибудь функции, имеют по умолчанию класс памяти *static* и называются глобальными переменными. Глобальная переменная доступна для любой функции, определенной в том же самом файле после того, как эта переменная была объявлена. Глобальные переменные также доступны для функций из других файлов; однако, в этом случае глобальные переменные должны быть объявлены в каждом файле, в котором они используются. Например, если мы определяем глобальную целочисленную переменную *flag* в одном файле и ссылаемся на нее в другом файле, второй файл должен содержать объявление

```
extern int flag;
```

записанное ранее, чем переменная *flag* будет использована в этом файле. В приведенном объявлении, спецификатор класса памяти *extern* указывает компилятору, что переменная *flag* либо будет определена позже в том же самом файле, либо ее определение находится в другом файле. Компилятор потом сообщит компоновщику, что в файле имеется неразрешенная ссылка на переменную *flag* (компилятор не знает, где определена переменная *flag*, и перекладывает эту задачу на редактор связей). Если компоновщик не сможет найти определения для *flag*, он выдаст сообщение об ошибке редактирования связей и не создаст исполняемого файла. Если соответствующее определение глобальной переменной будет найдено, то компоновщик разрешит ссылки на нее, указав ее местонахождение.

Совет по повышению эффективности 18.1

Глобальные переменные улучшают показатели производительности программы, потому что к ним можно обращаться непосредственно из любой функции. При этом исключаются затраты на передачу данных в функцию.

Замечание по технике программирования 18.1

Если эффективность прикладной программы не является критическим параметром, то следует избегать употребления глобальных переменных, потому что они нарушают принцип минимума привилегий и затрудняют поддержку программ.

Так же, как спецификатор `extern` используется для объявления глобальных переменных в других файлах программ, прототипы функций могут быть использованы для расширения области действия функции за пределы файла, в котором она определена. При этом не требуется использовать спецификатор `extern` в прототипе функции. В каждый файл, в котором вызывается функция из другого файла, нужно включить прототип этой функции и откомпилировать эти файлы (см. раздел 17.2). Прототипы функций указывают компилятору, что эти функции либо будут определены позже в этом же файле, либо их определения находятся в другом файле. В этом случае компилятор не пытается разрешать ссылки на такую функцию, эту задачу он оставляет редактору связей. Если компоновщик не сможет найти определение соответствующей функции, он выдаст сообщение об ошибке.

В качестве примера использования прототипов функции для расширения области действия функции, рассмотрим любую программу, содержащую директиву препроцессора `#include <string.h>`. Эта директива включает в файл прототипы функций типа `strcmp` и `strcat`. Все функции в файле могут использовать функции `strcmp` и `strcat` для выполнения своих задач. Функции `strcmp` и `strcat` определены где-то в другом месте и нам не нужно знать, где они определены; мы просто можем использовать их в наших программах. Редактор связей разрешит ссылки на эти функции автоматически. Благодаря этому механизму мы можем использовать функции из стандартной библиотеки.

Замечание по технике программирования 18.2

Программы, состоящие из нескольких исходных файлов, облегчают процесс повторного использования программных кодов и свидетельствуют о хорошей технике программирования. Некоторые функции могут понадобиться в других приложениях. Такие функции следует помещать в отдельные исходные файлы, каждому из которых должен соответствовать заголовочный файл, содержащий прототипы функций. Тогда разработчики других приложений могут использовать эти функции, включая соответствующий заголовочный файл, компилируя и компонуя свое приложение с соответствующим исходным файлом.

Замечание по мобильности 18.1

Некоторые системы не поддерживают имена глобальных переменных и функций длиной более шести символов. Этот факт следует учитывать при написании программ, которые будут переноситься на другие платформы.

Существует возможность ограничить область действия глобальной переменной или функции файлом, в котором они определяются. Спецификатор класса памяти `static`, если его применить к глобальной переменной или функции, ограничит их использование только функциями, определенными в том же самом файле. Это называется *внутренним связыванием*. Глобальные переменные и функции, при объявлении которых не использовался спецификатор `static`, имеют *внешнее связывание*, т. е. к ним можно обращаться из других

файлов, если те файлы содержат соответствующие объявления и прототипы функций.

При объявлении глобальной переменной

```
static float pi = 3.14159;
```

создается переменная `pi` типа `float` со значением `3.14159`, которая будет видна только функциям того файла, в котором она определена.

Спецификатор `static` обычно применяется к вспомогательным функциям (утилитам), которые вызываются только функциями из текущего файла. Если функция не нужна нигде более, кроме файла, в котором она определена, то следуя принципу минимизации привилегий эту функцию нужно объявлять со спецификатором `static`. Если функция описана в файле до того, как она используется, то `static` должен применяться в описании функции. В других случаях этот спецификатор нужно использовать в прототипе функции.

При написании программ, состоящих из большого числа исходных файлов, компиляция программы может сделаться утомительным делом, поскольку даже в случае небольших изменений в одном файле вся программа должна быть перекомпилирована. К счастью, многие системы программирования имеют специальные утилиты, которые перекомпилируют только измененный файл программы. В системах UNIX такая утилита называется `make`. Утилита `make` использует при своей работе файл с именем `makefile`, который содержит инструкции для компиляции и связывания программы. Системы типа Borland C++ и Microsoft Visual C++ для персональных компьютеров, также, имеют утилиту `make` и, кроме того, имеют «проекты». За подробной информацией по утилите `make` обращайтесь к руководству по вашей системе программирования.

18.6. Завершение программы при помощи функций `exit` и `atexit`

Библиотека утилит общего назначения (`stdlib.h`) предоставляет методы завершения выполнения программы, отличные от стандартного выхода из функции `main` с помощью оператора `return`. Функция `exit` вызывает нормальное завершение программы. Функция часто используется для завершения программы в случаях, когда произошла ошибка ввода, или если файл, который программа должна обработать, не может быть открыт. Функция `atexit` регистрирует функцию, которая будет вызываться при нормальном завершении программы, т. е., когда программа завершается по достижении конца функции `main`, или когда вызывается функция `exit`.

Функция `atexit` принимает в качестве параметра указатель на функцию (т. е. имя функции). Функции,ываемые при завершении программы, не должны иметь параметров и не могут возвращать значение. Может быть зарегистрировано до 32 функций, выполняющихся при завершении программы.

Функция `exit` имеет один параметр. Обычно в качестве аргумента используются символические константы `EXIT_SUCCESS` или `EXIT_FAILURE`. Если `exit` вызывается со значением `EXIT_SUCCESS`, то программа возвращает исполняющей системе значение нормального завершения программы, определяемое реализацией системы. Если `exit` вызывается с аргументом `EXIT_FAILURE`, то возвращается определяемое реализацией системы значе-

ние, соответствующее аварийному завершению программы. При выполнении функции `exit` вызываются функции, предварительно зарегистрированные функцией `atexit` в порядке, обратном порядку их регистрации; все потоки, связанные с программой, очищаются и закрываются и управление возвращается среде выполнения. В программе на рис. 18.4 показан пример использования функций `exit` и `atexit`. Программа предлагает пользователю определить, должна ли она завершаться вызовом `exit` или оператором `return` в конце функции `main`. Обратите внимание, что функция `print` выполняется при любом способе завершения программы.

18.7. Спецификатор типа volatile

В главах 4 и 5 мы представили вам спецификатор типа `const`. В C++ имеется еще один спецификатор типа — `volatile`. В расширенном стандарте C++ (Cb94) сказано, что когда используется спецификатор типа `volatile`, способ доступа к объекту такого типа зависит от реализации системы. По утверждению Кернигана и Ричи (Ке88), спецификатор `volatile` используется для подавления различных видов оптимизации.

```
// Использование функций exit и atexit
#include <iostream.h>
#include <stdlib.h>

void print(void);

main()
{
    atexit (print); // регистрация функции print
    cout << "Введите 1 для завершения программы функцией exit"
        << endl
        << "Введите 2 для нормального завершения программы"
        << endl;

    int answer;
    cin >> answer;

    if (answer == 1) {
        cout << endl << "Завершение программы функцией exit"
            << endl;
        exit(EXIT_SUCCESS);
    }

    cout << endl << "Завершение программы по достижении конца main"
        << endl;

    return 0;
}

void print(void)
{
    cout << "Выполнение функции print при завершении программы"
        << endl << "Программа завершена" << endl;
}
```

Рис. 18.4. Использование функций `exit` и `atexit` (часть 1 из 2)

Введите 1 для завершения программы функцией `exit`

Введите 2 для нормального завершения программы

1

Завершение программы функцией `exit`

Выполнение функции `print` при завершении программы

Программа завершена

Введите 1 для завершения программы функцией `exit`

Введите 2 для нормального завершения программы

2

Завершение программы по достижении конца `main`

Выполнение функции `print` при завершении программы

Программа завершена

Рис. 18.4. Использование функций `exit` и `atexit` (часть 2 из 2)

18.8. Суффиксы целочисленных и вещественных констант

Для определения типов целочисленных и вещественных констант в C++ используются суффиксы. Целочисленные суффиксы: `u` или `U` для целого без знака (`unsigned`); `l` или `L` для длинного целого (`long`); `ul` или `UL` для длинного целого без знака (`unsigned long`). Ниже приведены целочисленные константы типов `unsigned`, `long` и `unsigned long` соответственно:

`174u`

`8358L`

`28373ul`

Если целочисленная константа задана без суффикса, то ее тип определяется первым типом, способным разместить значение такого размера (типы просматриваются в следующем порядке: сначала `int`, затем `long int`, затем `unsigned long int`).

Для констант с плавающей запятой имеются следующие суффиксы: `f` или `F` для типа `float`; `l` или `L` для типа `long double`. Следующие константы имеют тип `long double` и `float` соответственно:

`3.14159L`

`1.28f`

Вещественная константа, определенная без суффикса, автоматически имеет тип `double`.

18.9. Обработка сигналов

Сигнал — это некоторое непредвиденное событие (прерывание), которое может вызвать преждевременное завершение программы. Перечислим некоторые из таких непредвиденных событий: *прерывания* (комбинация кла-

виш <ctrl> с в UNIX или DOS), недопустимая команда, ошибочный доступ к памяти (нарушение сегментации), запрос от операционной системы о завершении работы и ошибка операций с вещественными числами (деление на нуль или перемножение слишком больших действительных чисел). Библиотека обработки сигналов содержит функцию `signal`, перехватывающую непредвиденные события. Функция `signal` получает два параметра: целочисленный номер сигнала и указатель на функцию обработки сигнала. Сигналы могут генерироваться функцией `raise`, которая получает целочисленное значение номера сигнала в качестве аргумента. На рис. 18.5 перечислены стандартные сигналы, определяемые в заголовочном файле `signal.h`. В программе на рис. 18.6 демонстрируется использование функций `signal` и `raise`.

В программе на рис. 18.6 функция `signal` используется для перехвата интерактивного сигнала (`SIGINT`). Программа начинается с вызова функции `signal` с аргументами `SIGINT` и указателем на функцию `signal_handler` (не забудьте, что имя функции — это указатель, ссылающийся на начало функции). Когда генерируется сигнал типа `SIGINT`, управление передается функции `signal_handler`, которая выводит сообщение и предлагает пользователю возможность продолжить нормальное выполнение программы. Если пользователь желает продолжить выполнение программы, обработчик сигнала повторно инициализируется вызовом функции `signal` (в некоторых системах требуется повторная инициализация обработчика сигнала) и управление передается в точку программы, в которой сигнал был обнаружен. В этой программе интерактивный сигнал моделируется при помощи вызова функции `raise`. Для этой цели генерируются случайные числа в диапазоне от 1 до 50. Как только будет получено случайное число 25, будет вызвана функция `raise`, генерирующая нужный сигнал. Обычно, интерактивные сигналы возникают за пределами программы. Например, при нажатии во время выполнения программы комбинации клавиш <ctrl> с в UNIX или DOS генерируется интерактивный сигнал, в результате которого выполнение программы завершается. Обработка сигналов может использоваться для перехвата интерактивного сигнала и предотвращения прерывания программы.

Сигнал	Объяснение
<code>SIGABRT</code>	Аварийное завершение программы (например, в результате вызова функции <code>abort</code>).
<code>SIGFPE</code>	Ошибка арифметической операции, например, деление на нуль или операция, вызвавшая переполнение.
<code>SIGILL</code>	Обнаружение недопустимой команды.
<code>SIGINT</code>	Получение интерактивного сигнала.
<code>SIGSEGV</code>	Ошибка обращения к памяти.
<code>SIGTERM</code>	Запрос о завершении работы программы.

Рис. 18.5. Сигналы, определяемые в файле заголовочный `signal.h`

```
// Обработка сигналов

#include <iostream.h>
#include <iomanip.h>

#include <signal.h>
#include <stdlib.h>
#include <time.h>

void signal_handler(int);

main()
{
    signal (SIGINT, signal_handler);
    srand(time(NULL));

    for (int i = 1; i < 101; i++) {
        int x = 1 + rand() % 50;

        if (x == 25)
            raise(SIGINT);

        cout << setw(4) << i;

        if (i % 10 == 0)
            cout << endl;
    }

    return 0;
}

void signal_handler(int signalValue)
{
    cout << endl << "Получен сигнал прерывания (" 
        << signalValue
        << ")." << endl
        << "Хотите продолжать (1 = да или 2 = нет) ?  ";

    int response;
    cin >> response;

    while (response != 1 && response != 2) {
        cout << " (1 = да или 2 = нет) ?  ";
        cin >> response;
    }

    if (response == 1)
        signal(SIGINT, signal_handler);
    else
        exit(EXIT_SUCCESS);
}
```

Рис. 18.6. Использование обработки сигнала (часть 1 из 2)

```

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  84  85  86  87  88
Получен сигнал прерывания (4).
Хотите продолжать (1 = да или 2 = нет)?  1
89  90
91  92  93  94  95  96  97  98  99 100

```

Рис. 18.6. Использование обработки сигнала (часть 2 из 2)

18.10. Динамическое выделение памяти: функции `calloc` и `realloc`

В главе 7, когда мы обсуждали динамическое выделение памяти в стиле C++ с помощью операторов `new` и `delete`, мы сравнивали их с функциями С `malloc` и `free`.

Программисты на C++ должны использовать для выделения памяти `new` и `delete`, а не `malloc` и `free`. Однако, многие, программирующие на C++, сталкиваются с большим числом программ, полученных в наследство от языка С. Вот почему мы дополнительно обсуждаем динамическое выделение памяти в стиле С.

Библиотека утилит общего назначения (`stdlib.h`) содержит еще две других функции динамического выделения памяти: `calloc` и `realloc`. Эти функции могут использоваться для создания и изменения размера *динамических массивов*. Как было показано в главе 5, «Указатели и строки», указатель на массив может индексироваться, подобно тому, как индекс используется с именем массива. Аналогично указателем на непрерывный в памяти объект, созданный при помощи функции `calloc`, можно пользоваться как массивом. Функция `calloc` динамически выделяет память под массив. Прототип функции `calloc` выглядит следующим образом:

```
void *calloc(size_t nmemb, size_t size);
```

Функция получает два параметра: число элементов (`nmemb`) и размер каждого элемента (`size`). Элементам массива присваиваются нулевые начальные значения. Функция возвращает указатель на выделенную память, или нулевой указатель (0), если память не выделена.

Функция `realloc` изменяет размер объекта, память под который была выделена предыдущим обращением к функциям `malloc`, `calloc` или `realloc`. Содержимое объекта не изменяется при условии, что выделяемый объем памяти больше, чем предыдущий размер массива. В противном случае содержимое не изменяется только в пределах нового размера объекта. Прототип функции `realloc` имеет следующий вид

```
void *realloc(void *ptr, size_t size);
```

Функция `realloc` имеет два аргумента: указатель на объект (`ptr`) и новый размер объекта (`size`). Если значение `ptr` ровно `NULL`, то `realloc` работает тождественно функции `malloc`. Если значение `size` равно 0, а значение `ptr` не `NULL`, то занимаемая объектом память освобождается. В случае, когда значение `ptr` не `NULL` и значение `size` больше нуля, функция `realloc` будет пытаться выделить новый блок памяти для объекта. Если память не может быть выделена, то объект, на который указывает `ptr`, не изменяется. Функция `realloc` возвращает или указатель на выделенную область памяти, или `NULL`.

18.11. Безусловный переход: оператор `goto`

На протяжении всей этой книги мы подчеркивали важность использования методов структурного программирования, позволяющих создавать надежное программное обеспечение, которое легко отлаживать, поддерживать и модифицировать. Однако, в некоторых случаях эффективность работы программы бывает важнее, чем строгая приверженность методам структурного программирования. В таких случаях могут использоваться некоторые из неструктурных методов программирования. Например, мы можем использовать оператор `break`, чтобы завершить выполнение структуры повторения прежде, чем условие продолжения цикла примет значение «ложь». Это избавит нас от ненужных повторений цикла, когда задача уже выполнена, а цикл еще не завершил своей работы.

Другой пример неструктурного программирования — *оператор безусловного перехода `goto`*. В результате выполнения оператора `goto` управление передается первому оператору после *метки*, указанной в операторе `goto`. Метка — это идентификатор, за которым следует двоеточие. Метка должна находиться в пределах той же самой функции, что и оператор `goto`, который на нее ссылается. В программе на рис. 18.7 оператор `goto` используется для организации цикла из десяти проходов, в котором выводится значение счетчика `count`. После задания `count` начального значения 1 программа проверяет, не превысило ли значение `count` числа 10 (метка `start` пропускается, поскольку метки не выполняют никаких действий). Если значение `count` больше 10, то управление передается от оператора `goto` первому оператору после метки `end`. В противном случае, значение `count` выводится и увеличивается, а управление передается первому оператору после метки `start`.

В главе 2 мы установили, что для написания любой программы достаточно трех управляющих структур: следования, выбора и повторения. Если придерживаться правил структурного программирования, то в некоторых случаях можно получить глубоко вложенные управляющие структуры, из которых потом будет трудно выбраться. Для быстрого выхода из таких глубоко вложенных структур некоторые программисты используют оператор `goto`, чтобы обойти многочисленные проверки выходов из управляющих структур.

Совет по повышению эффективности 18.2

Оператор `goto` может использоваться для эффективного выхода из глубоко вложенных структур управления.

```
// Применение оператора goto

#include <iostream.h>

main()
{
    int count = 1;

    start:           // метка
        if (count > 10)
            goto end;

        cout << count << " ";
        ++count;
        goto start;

    end:             // метка
    cout << endl;

    return 0;
}
```

1 2 3 4 5 6 7 8 9 10

Рис. 18.7. Применение оператора `goto`

Замечание по технике программирования 18.3

Оператор `goto` должен использоваться только в приложениях, ориентированных на эффективную работу. Оператор `goto` не является инструментом структурного программирования и программы, в которых он используется, труднее отлаживать, поддерживать и модифицировать.

18.12. Объединения

Объединение (union) — это область памяти, в которой в разные моменты времени могут находиться объекты разных типов. В любой момент времени объединение может содержать максимум один объект, потому что элементы объединения совместно используют одну и ту же область памяти. На программиста возлагается обязанность следить за тем, чтобы к данным в объединении обращались по имени элемента соответствующего типа данных.

Типичная ошибка программирования 18.2

Результат обращения не к тому элементу объединения, который был последним размещен в памяти, не определен.

Замечание по мобильности 18.2

Если тип ссылки на элемент объединения не соответствует типу данных, хранящемуся в этот момент в объединении, то результат такой ошибки зависит от реализации системы.

В разные отрезки времени выполнения программы некоторые объекты могут быть не нужны, т.е. программе требуется только часть ее объектов. Вместо того, чтобы впустую растрачивать память на объекты, которые используются не постоянно, можно поместить их в объединение, где они будут делить между собой одну и ту же область памяти. Число байтов памяти, выделяемых для объединения, должно быть не меньше, чем размер самого большого элемента объединения.

Совет по повышению эффективности 18.3

Объединения помогают экономить память.

Замечание по мобильности 18.3

Объем памяти, выделяемый объединению, зависит от реализации системы.

Замечание по мобильности 18.4

Некоторые объединения не могут быть легко перенесены на другие компьютерные платформы. Перенесется ли объединение, или нет часто зависит от соглашений о выравнивании в памяти типов данных элементов объединения.

Объединения объявляются при помощи ключевого слова **union** в таком же формате, как структуры и классы. В следующем объявлении

```
union Number {  
    int x;  
    float y;  
};
```

вводится тип объединения **Number** с элементами **int x** и **float y**. Объединения обычно определяются в программе до функции **main** для того, чтобы в любой функции программы можно было объявить переменные типа этого объединения.

Замечание по технике программирования 18.4

Как и при объявлении структур и классов при помощи ключевых слов **struct** и **class**, объявление объединения с помощью ключевого слова **union** создает новый тип, а не объект. Объявление структуры или объединения вне определения какой-либо функции не создает глобального объекта.

Единственными допустимыми встроенными операциями, которые могут выполняться над объединениями, являются: операция присваивания значения одного объединения другому объединению того же типа, операция вычисления адреса объединения (**&**) и доступ к элементу объединения при помощи операций доступа к элементу структуры (**.** и **->**). Над объединениями

не могут выполняться операции сравнения по тем же самым причинам, по каким они не выполняются над структурами.

Типичная ошибка программирования 18.3

Попытка сравнения объединений приводит к синтаксической ошибке, потому что компилятор не знает, какой элемент каждого объединения активен в настоящий момент и, следовательно, какие элементы объединений нужно выбирать для сравнения.

Можно сказать, что объединение походит на класс, который может иметь конструктор для инициализации любого из своих элементов. Объединение, которое не имеет конструктора, может быть инициализировано значением другого объединения того же самого типа, или выражением типа, соответствующего типу первого элемента объединения, или с помощью инициализатора (заключенного в фигурные скобки значения, соответствующего по типу первому элементу объединения). Объединения могут иметь другие функции-элементы, например, деструкторы, но функции-элементы объединения не могут объявляться виртуальными. По умолчанию элементы объединения имеют открытый уровень доступа.

Типичная ошибка программирования 18.4

Инициализация объединения при его объявлении значением или выражением, тип которого отличается от типа первого элемента объединения.

Объединение не может использоваться в качестве базового класса в иерархии наследования, т. е., из объединений не могут быть получены производные классы. Элементами объединения могут являться только те объекты, которые не имеют конструктора, деструктора, или перегруженной операции присваивания. Ни один из данных-элементов объединения не может быть объявлен со спецификатором класса памяти `static`.

В программе на рис. 18.8 объявляется переменная `value` типа объединения `Number` и выводятся значения обоих элементов этого объединения: как типа `int`, так и типа `float`. Вывод программы зависит от реализации системы. По результатам вывода программы видно, насколько сильно различается внутреннее машинное представление типов данных `float` и `int`.

Анонимное объединение — это объединение, которое не имеет имени типа и при определении которого перед завершающей точкой с запятой не задается имя объекта или указателя. При объявлении такого объединения создается не тип, а объект, не имеющий имени. К элементам анонимного объединения можно обращаться непосредственно по их именам в той области действия, в которой анонимное объединение объявлено, как к любым локальным переменным; при этом не нужно использовать операцию точка (`.`) или стрелка (`->`).

Анонимные объединения имеют некоторые ограничения. Они могут иметь только данные-элементы. Все элементы анонимного объединения должны иметь открытый уровень доступа. Глобальное анонимное объединение, область действия которого — файл, должно быть явно объявлено со спецификатором `static`.

На рис. 18.9 приводится пример использования анонимного объединения.

```
// Пример использования объединения
#include <iostream.h>

union Number {
    int x;
    float y;
};

main()
{
    Number value;

    value.x = 100;
    cout << "Задание значения целого элемента" << endl
        << "и печать обоих элементов." << endl << "int:   "
        << value.x << endl << "float: " << value.y << endl;

    value.y = 100.0;
    cout << "Задание значения элемента с плавающей запятой" << endl
        << "и печать обоих элементов." << endl << "int:   "
        << value.x << endl << "float: " << value.y << endl
        << endl;
    return 0;
}
```

**Задание значения целого элемента
и печать обоих элементов.**

int: 100

float: 3.504168e-16

**Задание значения элемента с плавающей запятой
и печать обоих элементов.**

int: 0

float: 100

Рис. 18.8. Печать значения объединения при обоих типах данных-элементов

```
// Использование анонимного объединения
#include <iostream.h>

main()
{
    // Объявление анонимного объединения
    // Заметьте, что элементы b, d и f разделяют
    // одну и ту же область памяти
    union {
        int b;
        float d;
        char *f;
    };

    // Объявление обычных локальных переменных
    int a = 1;
    float c = 3.3;
    char *e = "Анонимное";
```

Рис. 18.9. Использование анонимного объединения (часть 1 из 2)

```

// Последовательное присваивание значения
// каждому элементу объединения и их вывод.
cout << a << ' ';
b = 2;
cout << b << endl;

cout << c << ' ';
d = 4.4;
cout << d << endl;

cout << e << ' ';
f = "объединение";
cout << f << endl;

return 0;
}

```

1 2
3.3 4.4

Анонимное объединение

Рис. 18.9. Использование анонимного объединения (часть 2 из 2)

18.13. Спецификации связывания

В C++ имеется возможность вызывать из программы, написанной на C++, функции, написанные на С и откомпилированные компилятором С. Как мы уже говорили в разделе 3.20, C++ для обеспечения безопасного по отношению к типам редактирования связей особым образом кодирует имена функций. Язык С имена функций не кодирует. Таким образом, функция, откомпилированная компилятором С, не будет идентифицирована при попытке связать модули на C++ с модулями на С, так как C++ будет искать закодированное имя функции. Но программист на C++ имеет возможность, воспользовавшись *спецификациями связывания*, сообщить компилятору, что данная функция компилировалась на компиляторе С и запретить компилятору C++ кодировать ее имя. Спецификации связывания полезны в том случае, когда имеются большие разработанные ранее библиотеки специализированных функций и пользователь или не имеет доступа к исходным текстам этих библиотек для их перекомпиляции на C++, или у него нет времени на перенесение кода библиотеки в C++.

Для того, чтобы сообщить компилятору, что одна или несколько функций компилировались компилятором С, их прототипы должны быть объявлены следующим образом:

```

extern "C" прототип функции // одна функция

extern "C"      // несколько функций
{
    прототипы функций
}

```

Эти объявления сообщают компилятору, что перечисленные функции компилировались не компилятором C++ и поэтому их имена кодировать не нужно. После компиляции программы на C++ указанные функции могут

быть успешно с ней связаны. Системы программирования на C++ обычно включают в себя перекомпилированные стандартные библиотеки языка C; в этом случае использование спецификаций связывания для вызова таких функций не требуется.

18.14. Заключительные замечания

Мы искренне надеемся, что изучение C++ и объектно-ориентированного программирования с помощью этого курса доставило вам удовольствие. Будущее кажется безоблачным и мы желаем вам успехов в погоне за ним!

- Мы были бы очень признательны Вам за комментарии, критику, сообщения о замеченных ошибках и предложения по улучшению текста книги. Мы упомянем всех внесших свой вклад в следующем издании книги. Пожалуйста, присылайте Ваши сообщения на наш адрес email:

deitel@world.std.com

Желаем удачи!

Резюме

- Во многих операционных системах, в частности, в UNIX и DOS, имеется возможность переназначить (переадресовать) ввод в программу и вывод из программы.
- В UNIX и DOS переназначение ввода производится из командной строки при помощи знака переназначения ввода (<) или знака конвейера (I).
- В UNIX и DOS переназначение вывода производится из командной строки при помощи знака переназначения вывода (>) или знака добавления вывода (>>). Знак переназначения вывода просто помещает вывод программы в файл, а символ добавления вывода дописывает вывод программы в конец файла.
- Макросы и определения заголовочного файла stdarg.h, посвященного списку параметров переменной длины, обеспечивают возможности, необходимые при построении функций со списками параметров переменной длины.
- Знак многоточия (...) в прототипе функции указывает на то, что функция получает список параметров переменной длины.
- Тип va_list предназначен для хранения информации, необходимой макросам va_start, va_arg и va_end. Для обработки параметров из списка параметров переменной длины необходимо объявлять объект этого типа.
- Макрос va_start должен вызываться перед обработкой списка переменного числа параметров. Макрос инициализирует объект типа va_list для последующего использования его макросами va_arg и va_end.
- Макрос va_arg расширяется до выражения, которое имеет тип и значение следующего параметра в списке параметров переменной длины. Каждый вызов va_arg изменяет значение объекта типа va_list таким образом, что он ссылается на следующий по списку параметр.

- Макрос `va_end` обеспечивает нормальное возвращение из функции, в которой для обработки списка параметров переменной длины использовался макрос `va_start`.
- Во многих операционных системах, в частности, в DOS и UNIX, имеется возможность передачи аргументов в функцию `main` из командной строки, если в список параметров `main` включить параметры `int argc` и `char *argv[]`. Параметр `argc` получает значение, равное количеству аргументов командной строки. Параметр `argv` — это массив строк, в который помещаются значения аргументов командной строки.
- Описание функции должно полностью содержаться в одном файле, оно не может распространяться на два или большее число файлов.
- Глобальные переменные должны быть объявлены в каждом файле, в котором они используются.
- Прототипы функций могут расширять область действия функции за пределы файла, в котором она определена (при этом в прототипе функции не требуется использовать спецификатор `extern`). В каждый файл, в котором вызывается функция из другого файла, нужно включить прототип этой функции и откомпилировать эти файлы.
- Спецификатор класса памяти `static`, если его применить к глобальной переменной или функции, ограничит их использование только функциями, определенными в том же самом файле. Это называется внутренним связыванием. Глобальные переменные и функции, при объявлении которых не использовался спецификатор `static`, имеют внешнее связывание, т. е. к ним можно обращаться из других файлов, если те файлы содержат соответствующие объявления и прототипы функций.
- Спецификатор `static` обычно применяется к вспомогательным функциям (утилитам), которые вызываются только функциями из текущего файла. Если функция не нужна нигде более, кроме файла, в котором она определена, то следуя принципу минимизации привилегий эту функцию нужно объявлять со спецификатором `static`.
- При написании программ, состоящих из большого числа исходных файлов, компиляция программы может сделаться утомительным делом, поскольку даже в случае небольших изменений в одном файле вся программа должна быть перекомпилирована. Многие системы программирования имеют специальные утилиты, которые перекомпилируют только измененный файл программы. В системах UNIX такая утилита называется `make`. Утилита `make` использует при своей работе файл с именем `makefile`, который содержит инструкции для компиляции и связывания программы.
- Функция `exit` вызывает нормальное завершение программы.
- Функция `atexit` регистрирует функцию, которая будет вызываться при нормальном завершении программы, т. е., когда программа завершается по достижении конца функции `main`, или когда вызывается функция `exit`.
- Функция `atexit` принимает в качестве параметра указатель на функцию (т. е. имя функции). Функции,ываемые при завершении программы, не должны иметь параметров и не могут возвращать значение.

Может быть зарегистрировано до 32 функций, выполняющихся при завершении программы.

- Функция `exit` имеет один параметр. Обычно в качестве аргумента используются символьические константы `EXIT_SUCCESS` или `EXIT_FAILURE`. Если `exit` вызывается со значением `EXIT_SUCCESS`, то программа возвращает исполняющей системе значение нормального завершения программы, определяемое реализацией системы. Если `exit` вызывается с аргументом `EXIT_FAILURE`, то возвращается определяемое реализацией системы значение, соответствующее аварийному завершению программы.
- При выполнении функции `exit` вызываются функции, предварительно зарегистрированные функцией `atexit` в порядке, обратном порядку их регистрации; все потоки, связанные с программой, очищаются и закрываются и управление возвращается среде выполнения.
- В расширенном стандарте C++ сказано, что при модификации типа спецификатором `volatile` способ доступа к объекту такого типа зависит от реализации. Кернигана и Ричи отмечают, что спецификатор `volatile` используется для подавления различных видов оптимизации.
- Для определения типов целочисленных и вещественных констант в C++ используются суффиксы. Целочисленные суффиксы: `u` или `U` для целого без знака (`unsigned`); `l` или `L` для длинного целого (`long`); `ul` или `UL` для длинного целого без знака (`unsigned long`). Если целочисленная константа задана без суффикса, то ее тип определяется первым типом, способным разместить значение такого размера (типы просматриваются в следующем порядке: сначала `int`, затем `long int`, затем `unsigned long int`). Для констант с плавающей запятой имеются следующие суффиксы: `f` или `F` для типа `float`; `l` или `L` для типа `long double`. Вещественная константа, определенная без суффикса, автоматически имеет тип `double`.
- Библиотека обработки сигналов содержит функцию `signal`, перехватывающую непредвиденные события. Функция `signal` получает два параметра: целочисленный номер сигнала и указатель на функцию обработки сигнала.
- Сигналы могут генерироваться функцией `raise`, которая получает целочисленное значение номера сигнала в качестве аргумента.
- Библиотека утилит общего назначения (`stdlib.h`) содержит функции динамического выделения памяти `calloc` и `realloc`. Эти функции могут использоваться для создания и изменения размера динамических массивов.
- Функция `calloc` имеет два параметра: число элементов (`nmemb`) и размер каждого элемента (`size`). Элементам массива присваиваются нулевые начальные значения. Функция возвращает указатель на выделенную память, или нулевой указатель (0), если память не выделена.
- Функция `realloc` изменяет размер объекта, память под который была выделена предыдущим обращением к функциям `malloc`, `calloc` или `getalloc`. Содержимое объекта не изменяется при условии, что выделяемый объем памяти больше, чем предыдущий размер массива.

- Функция `realloc` имеет два аргумента: указатель на объект (`ptr`) и новый размер объекта (`size`). Если значение `ptr` ровно `NULL`, то `realloc` работает тождественно функции `malloc`. Если значение `size` равно 0, а значение `ptr` не `NULL`, то занимаемая объектом память освобождается. В случае, когда значение `ptr` не `NULL` и значение `size` больше нуля, функция `realloc` будет пытаться выделить новый блок памяти для объекта. Если память не может быть выделена, то объект, на который указывает `ptr`, не изменяется. Функция `realloc` возвращает или указатель на выделенную область памяти, или `NULL`.
- В результате выполнения оператора `goto` происходит изменение в потоке управления программы: управление передается первому оператору после метки, определенной в операторе `goto`.
- Метка — это идентификатор, за которым следует двоеточие. Метка должна находиться в пределах той же самой функции, что и оператор `goto`, который на нее ссылается.
- Объединение — это производный тип данных, элементы которого совместно используют одну и ту же область памяти. Элементы могут быть любого типа.
- Память, выделяемая для объединения, должна быть достаточной для того, чтобы в ней размещался самый большой элемент объединения. В большинстве случаев, объединения содержат два или больше типов данных. Только один элемент и, следовательно, только один тип данных может быть доступен в объединении в каждый данный момент времени.
- Объединения объявляются при помощи ключевого слова `union` в том же формате, что и структуры.
- Объединение может быть инициализировано только значением типа первого элемента объединения.
- Программист на C++ имеет возможность воспользоваться спецификациями связывания и запретить компилятору C++ кодировать имя функции, если эта функция компилировалась компилятором С.
- Для того, чтобы сообщить компилятору, что одна или несколько функций компилировались компилятором С, их прототипы должны быть объявлены следующим образом:

```
extern "C" прототип функции      // одна функция
extern "C"
{
    прототипы функций
}
```

Эти объявления сообщают компилятору, что перечисленные функции компилировались не компилятором C++ и поэтому их имена кодировать не нужно. После компиляции программы на C++ указанные функции могут быть успешно с ней связаны.

- Системы программирования на C++ обычно включают в себя перекомпилированные стандартные библиотеки языка С; в этом случае использование спецификаций связывания для вызова таких функций не требуется.

Терминология

argc	знак переназначения ввода <
argv	знак переназначения вывода >
atexit	конвейер
calloc	недопустимая команда
const	объединение
exit	оператор <i>goto</i>
EXIT_FAILURE	ошибка обращения к памяти (сегментации)
EXIT_SUCCESS	ошибка операции с вещественными числами
make	переназначение ввода-вывода
makefile	перехват
raise	прерывание
realloc	событие
signal	спецификатор класса памяти <i>extern</i>
signal.h	спецификатор класса памяти <i>static</i>
stdarg.h	список параметров переменной длины
union	суффикс константы типа <i>float</i> (<i>f</i> or <i>F</i>)
va_arg	суффикс константы типа <i>long double</i> (<i>l</i> or <i>L</i>)
va_end	суффикс целочисленной константы типа <i>long</i> (<i>l</i> or <i>L</i>)
va_list	суффикс целочисленной константы типа <i>unsigned</i> (<i>u</i> or <i>U</i>)
va_start	суффикс целочисленной константы типа <i>unsigned long</i> (<i>ul</i> or <i>UL</i>)
volatile	знак добавления вывода >>
аргументы командной строки	знак конвейера
библиотека функций обработки сигналов	
внешнее связывание	
внутреннее связывание	
динамические массивы	
знак добавления вывода >>	
знак конвейера	

Типичные ошибки программирования

- 18.1. Размещение многоточия в середине списка параметров функции.
Многоточие может располагаться только в конце списка параметров.
- 18.2. Результат обращения не к тому элементу объединения, который был последним размещен в памяти, не определен.
- 18.3. Попытка сравнения объединений приводит к синтаксической ошибке, потому что компилятор не знает, какой элемент каждого объединения активен в настоящий момент и, следовательно, какие элементы объединений нужно выбирать для сравнения.
- 18.4 Инициализация объединения при его объявлении значением или выражением, тип которого отличается от типа первого элемента объединения.

Советы по повышению эффективности

- 18.1. Глобальные переменные улучшают показатели производительности программы, потому что к ним можно обращаться непосредственно из любой функции. При этом исключаются затраты на передачу данных в функцию.

18.2. Глобальные переменные улучшают показатели производительности программы, потому что к ним можно обращаться непосредственно из любой функции. При этом исключаются затраты на передачу данных в функцию.

18.3. Объединения помогают экономить память.

Замечания по мобильности

18.1. Некоторые системы не поддерживают имена глобальных переменных и функций длиной более шести символов. Этот факт следует учитывать при написании программ, которые будут переноситься на другие платформы.

18.2. Если тип ссылки на элемент объединения не соответствует типу данных, хранящемуся в этот момент в объединении, то результат такой ошибки зависит от реализации системы.

18.3. Объем памяти, выделяемый объединению, зависит от реализации системы.

18.4. Некоторые объединения не могут быть легко перенесены на другие компьютерные платформы. Перенесется ли объединение, или нет часто зависит от соглашений о выравнивании в памяти типов данных элементов объединения.

Замечания по технике программирования

18.1. Если эффективность прикладной программы не является критическим параметром, то следует избегать употребления глобальных переменных, потому что они нарушают принцип минимума привилегий и затрудняют поддержку программ.

18.2. Программы, состоящие из нескольких исходных файлов, облегчают процесс повторного использования программных кодов и свидетельствуют о хорошей технике программирования. Некоторые функции могут понадобиться в других приложениях. Такие функции следует помещать в отдельные исходные файлы, каждому из которых должен соответствовать заголовочный файлы, содержащий прототипы функций. Тогда разработчики других приложений могут использовать эти функции, включая соответствующий заголовочный файл, компилируя и компонуя свое приложение с соответствующим исходным файлом.

18.3. Оператор `goto` должен использоваться только в приложениях, ориентированных на эффективную работу. Оператор `goto` не является инструментом структурного программирования и программы, в которых он используется, труднее отлаживать, поддерживать и модифицировать.

18.4. Как и при объявлении структур и классов при помощи ключевых слов `struct` и `class`, объявление объединения с помощью ключевого слова `union` создает новый тип, а не объект. Объявление структуры или объединения вне определения какой-либо функции не создает глобального объекта.

Упражнения для самопроверки

18.1. Заполните пробелы в следующих утверждениях:

- а) Знак _____ переназначает ввод данных с клавиатуры на ввод данных из файла.
- б) Знак _____ переназначает вывод на экран на вывод в файл.
- с) Знак _____ используется для добавления вывода программы в конец файла.
- д) Знак _____ используется для направления вывода одной программы на ввод другой программы.
- е) Если в списке параметров функции поставлено _____, то это значит, что функция может принимать переменное число параметров.
- ф) Макрос _____ должен вызываться перед обработкой списка с переменным числом параметров.
- г) Макрос _____ используется для доступа к значению очередного параметра в списке параметров переменной длины.
- х) Макрос _____ обеспечивает нормальное возвращение из функции, в которой для обработки списка параметров переменной длины использовался макрос `va_start`.
- и) Значение параметра _____ функции `main` равно числу аргументов командной строки.
- ж) Параметр _____ функции `main` хранит аргументы командной строки в виде символьных строк.
- к) Утилита UNIX _____ использует файл, называемый _____, который содержит инструкции по компиляции и связыванию программы, состоящей из нескольких исходных файлов. Утилита выполняет компиляцию файла только в том случае, если файл был изменен с момента, когда он в последний раз компилировался.
- л) Функция _____ приводит к завершению выполнения программы.
- м) Функция _____ регистрирует функцию, которая будет вызываться при нормальном завершении программы.
- п) Спецификатор типа _____ означает, что объект не должен изменяться после того, как он был инициализирован.
- о) К целочисленной или вещественной константе может быть добавлен _____, определяющий точный тип константы.
- р) Функция _____ может использоваться для перехвата не-предвиденных событий.
- q) Функция _____ используется для генерации сигнала из самой программы.
- г) Функция _____ выполняет динамическое выделение памяти для массива и присваивает элементам массива нулевые значения.
- с) Функция _____ изменяет размер динамически выделенного блока памяти.

- t) _____ является классом, содержащим совокупность переменных, которые занимают одну и ту же область памяти, но в разные моменты времени.
- и) Ключевое слово _____ используется для определения объединения.

Ответы на упражнения для самопроверки

- 18.1. а) переназначение ввода (<). б) переназначение вывода (>). в) добавление вывода(>>). д) конвейер (|). е) многоточие (...). ф) va_start. г) va_arg. х) va_end. и) argc. ж) argv. к) make, makefile. л) exit. м) atexit. н) const. о) суффикс. р) signal. ю) raise. р) calloc. с) realloc. т) Объединение. у) union.

Упражнения

- 18.2. Напишите программу, которая вычисляет произведение ряда целых чисел, передающихся функции `product` через список параметров переменной длины. Проверьте вашу функцию, вызвав ее несколько раз с различным числом аргументов.
- 18.3. Напишите программу, которая печатает аргументы, полученные из командной строки.
- 18.4. Составьте программу сортировки целочисленного массива в порядке возрастания или в порядке убывания. Программа должна принимать аргументы командной строки: значение `-a` должно соответствовать сортировке по возрастанию, а `-d` — сортировке по убыванию. (Причение: это стандартный способ передачи опций программе в UNIX.)
- 18.5. Узнайте в руководствах по вашей системе, какие сигналы поддерживаются библиотекой обработки сигналов (`signal.h`). Напишите программу, обрабатывающую сигналы `SIGABRT` и `SIGINT`. Программа должна проверять перехват этих сигналов при вызове функции `abort`, генерирующем сигнал `SIGABRT`, и при нажатии комбинации клавиш `<ctrl>` с, генерирующем сигнал `SIGINT`.
- 18.6. Сделайте программу, которая динамически выделяет память под массив целых чисел. Размер массива должен вводиться с клавиатуры. Элементам массива должны присваиваться значения, также вводимые с клавиатуры. Выведите значения массива, а после этого перераспределите память под этот массив, уменьшив ее размер вдвое. Выведите значения оставшейся части массива, чтобы убедиться в том, что они соответствуют значениям первой половины исходного массива.
- 18.7. Напишите программу, которая получает два имени файла через аргументы командной строки, читает по одному символу из первого файла и записывает их в обратной последовательности во второй файл.

- 18.8. Напишите программу, использующую оператор `goto` для организации вложенных циклов, которая выводила бы квадрат из звездочек, показанный ниже:

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

Программа должна использовать только следующие три оператора вывода:

```
cout << '*';  
cout << ' ';  
cout << endl;
```

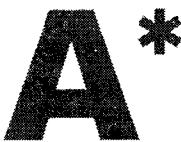
- 18.9. Объявите объединение `Data`, содержащее элементы с типа `char`, `s` типа `short`, `l` типа `long`, `f` типа `float` и `d` типа `double`.
- 18.10. Создайте объединение `Integer` с элементами с типа `char`, `s` типа `short`, `i` типа `int` и `l` типа `long`. Напишите программу, в которую вводятся значения типов `char`, `short`, `int` и `long` и сохраняются в элементах переменных типа объединение `Integer`. Каждую переменную типа объединение выведите как тип `char`, `short`, `int` и `long`. Всегда ли значения выводятся правильно?
- 18.11. Создайте объединение `FloatingPoint` с элементами `f` типа `float`, `d` типа `double` и `l` типа `long double`. Напишите программу, в которую вводятся значения типы `float`, `double` и `long double` и сохраняются в элементах переменных типа объединение `FloatingPoint`. Каждую переменную типа объединение выведите как тип `float`, `double` и `long double`. Всегда ли значения выводятся правильно?

- 18.12. Задано объединение

```
union A {  
    float y;  
    char *z;  
};
```

Какие из следующих операторов, выполняющих инициализацию этого объединения, являются правильными

- a) `A p = B;` // тип объекта `B` совпадает с типом `A`
- b) `A q = x;` // `x` – вещественная величина
- c) `A r = 3.14159;`
- d) `A s = { 79.63 };`
- e) `A t = { "Hi There!" };`
- f) `A u = { 3.14159, "Pi" };`



Стандартная библиотека

A.1. Ошибки <errno.h>

EDOM

ERANGE

Символические целочисленные отличные друг от друга ненулевые константы, удобные для использования в директивах препроцессора #if.

errno

Величина типа **int**, которой некоторые библиотечные функции могут присваивать положительный код ошибки. Значение **errno** обнуляется при запуске программы; однако в библиотеке не существует функции, которая присваивала бы **errno** нулевое значение. Поэтому программа, которая использует **errno** для проверки ошибок, должна сама присваивать нулевое значение этой величине перед вызовом библиотечной функции и проверять ее перед следующим обращением к библиотечной функции. Библиотечная функция может сохранять значение **errno** на входе в функцию, затем присваивать ему нулевое значение и восстанавливать первоначальное значение, если значение **errno** перед возвратом из функции все еще равно нулю. **errno** может получить ненулевое значение в результате вызова библиотечной функции независимо от того, произошла ошибка или нет; все это происходит потому, что использование **errno** не документировано в стандарте описания функций.

A.2. Общие определения <stddef.h>

NULL

Константа нулевого указателя, зависящая от используемой системы.

* Подтверждение прав: 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, элемент_структурь)

Этот макрос, расширяемый до целочисленного константного выражения типа `size_t`, представляет величину смещения в байтах элемента структуры (обозначенного параметром `элемент_структурь`) относительно начала структуры (обозначенной параметром `тип`). Параметр `элемент_структурь` должен задаваться таким образом, чтобы при условии

```
static тип t;
```

выражение `&(t.элемент_структурь)` вычислялось бы до адресной константы. (В случае, если аргумент `элемент_структурь` является битовым полем, то результат непредсказуем.)

ptrdiff_t

Целый со знаком тип результата вычитания двух указателей.

size_t

Целый без знака тип результата операции `sizeof`.

wchar_t

Тип целочисленных данных, диапазон значений которых обеспечивает представление кодов всех членов наибольшего расширенного набора символов среди поддерживаемых локализаций (наборов установок, характеризующих ту или иную «культурную среду» — форматы времени, денежные единицы и т.п.). Значение кода нулевого символа должно быть равно нулю, а каждый символ из основного набора символов должен иметь значение кода, равное значению при использовании этого символа в целочисленной символьной константе.

A.3. Диагностика <assert.h>

```
void assert(int выражение);
```

Макрос `assert` используется в программах для диагностики. Если при расширении макроса значение параметра `выражение` ложно, то `assert` выдает в стандартный файл ошибок в формате, определяемом системой программирования, диагностическое сообщение об ошибке, включая текстовое значение параметра, имя файла с исходным текстом и номер строки исходного текста (последние значения предоставляются макросами препроцессора `_FILE_` и `_LINE_` соответственно). Результирующее сообщение может иметь вид

```
Assertion failed: выражение, file xyz, line nnn
```

После этого макрос `assert` производит вызов функции `abort`. Если в исходном файле, который включает заголовок `assert.h`, появляется директива препроцессора

```
#define NDEBUG
```

все последующие макросы `assert` игнорируются.

A.4. Обработка символов <ctype.h>

Функции, представленные в этом разделе, возвращают ненулевое значение (`true`) в том и только в том случае, если значение аргумента с удовлетворяет условиям, указанным в описании функции.

```
int isalnum (int c);
```

Возвращает `true` для любого символа, для которого функции `isalpha` или `isdigit` возвращают `true`.

```
int isalpha (int c);
```

Возвращает `true` для любого символа, для которого функции `isupper` или `islower` возвращают `true`.

```
int iscntrl (int c);
```

Проверяет, является ли аргумент управляющим символом.

```
int isdigit (int c);
```

Проверяет, является ли аргумент десятичной цифрой.

```
int isgraph (int c);
```

Проверяет, является ли аргумент печатаемым символом, исключая символ пробела (' ').

```
int islower (int c);
```

Проверяет, является ли аргумент символом в нижнем регистре.

```
int isprint (int c);
```

Проверяет, является ли аргумент печатаемым символом, включая символ пробела (' ').

```
int ispunct (int c);
```

Проверяет, является ли аргумент печатаемым символом, исключая символ пробела (' ') и символы, для которых функция `isalnum` возвращает `true`.

```
int isspace (int c);
```

Проверяет, является ли аргумент стандартным символом разделителем. Стандартными символами разделителями являются: пробел (' '), прогон страницы ('\f'), новая строка ('\n'), возврат каретки ('\r'), символы горизонтальной табуляции ('\t') и вертикальной табуляции ('\v').

```
int isupper (int c);
```

Проверяет, является ли аргумент символом в верхнем регистре.

```
int isxdigit (int c);
```

Проверяет, является ли аргумент шестнадцатеричной цифрой.

```
int tolower (int c);
```

Преобразует символ верхнего регистра в соответствующий символ нижнего регистра. Если аргументом является символ, для которого `isupper` возвращает `true`, и имеется соответствующий символ, для которого `islower` возвращает `true`, функция `tolower` возвратит этот соответствующий символ; в противном случае, аргумент будет возвращен неизмененным (Замечание: к сожалению, для символов кириллицы функция не работает).

```
int toupper (int c);
```

Преобразует символ нижнего регистра в соответствующий символ верхнего регистра. Если аргументом является символ, для которого `islower` возвращает `true`, и имеется соответствующий символ, для которого `isupper` возвращает `true`, функция `toupper` возвратит этот соответствующий символ; в противном

случае, аргумент будет возвращен неизмененным (Замечание: к сожалению, для символов кириллицы функция не работает).

A.5. Локализации (характеристики той или иной «культурной среды» — форматы времени, денежные единицы и т.п.) <locale.h>

`LC_ALL`
`LC_COLLATE`
`LC_CTYPE`
`LC_MONETARY`
`LC_NUMERIC`
`LC_TIME`

Это символические целочисленные константы, необходимые для использования в качестве первого аргумента функции `setlocale`.

`NULL`

Константа нулевого указателя, зависящая от используемой системы.

`struct lconv`

Элементы этой структуры определяют форматирование численных значений. Структура должна содержать в любом порядке по крайней мере элементы, перечисленные ниже. В локализации "C" элементы структуры 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 соответствующую часть локализации программы, определяемую параметрами `category` и `locale`, и может использоваться для того, чтобы изменять или запрашивать состояние текущей локализации целиком или ее частей. Значение `LC_ALL` для параметра `category` обозначает локализацию целиком; другие значения параметра `category` задают только часть локализации. `LC_COLLATE` определяет поведение функций `strcoll` и `strxfrm`. `LC_CTYPE` воздействует на функции обработки символов и многобайтовые функции. `LC_MONETARY` относится к возвращаемой функцией `lconv` части локализации, имеющей дело с информацией о форматировании монетарных данных. `LC_NUMERIC` касается части локализации, информация

о которой возвращается функцией `localeconv` и которая определяет символ десятичной точки в функциях форматированного ввода-вывода, функции преобразования строк и форматирование именонетарных данных. `LC_TIME` определяет поведение функции `strftime`.

Локализация с именем "C" определяет минимальные установки окружения, необходимые для компиляции программ на С; значение "" подразумевает локализацию по умолчанию данной версии системы программирования. Функции `setlocale` можно передавать и другие имена, определяемые в используемой версии компилятора. При запуске программы выполняются действия, эквивалентные вызову функции

```
setlocale (LC_ALL, "C");
```

Если выбор указанной параметром `locale` локализации оказался успешным, функция `setlocale` возвращает указатель на строку, указанную в `category`, связанную с новой локализацией. В случае, если установка локализации оказывается неудачной, функция `setlocale` возвращает нулевой указатель и локализация программы не изменяется.

Если в качестве аргумента `locale` передается нулевой указатель, то функция `setlocale` возвращает указатель на строку, связанную с указанной значением аргумента `category` текущей локализации программы; локализация программы при этом не изменяется.

Если указатель на строку, возвращаемый `setlocale`, использовать как аргумент при последующем вызове `setlocale` с соответствующим значением `category`, то параметры локализации будут восстановлены. Эта строка, на которую ссылается указатель, может изменяться программой и может быть переписана при последующем обращении к функции `setlocale`.

```
struct lconv *localeconv (void);
```

Функция `localeconv` задает элементам объекта типа `struct lconv`, ответственным за форматирование численных величин (монетарных и немонетарных), значения, соответствующие правилам текущей среды.

Все элементы структуры типа `char *` (за исключением `decimal_point`) могут ссылаться на пустую строку ""; это означает, что данный параметр недоступен в данной локализации или он равен нулю. Члены типа `char` имеют целые положительные значения и любой из них может принимать значение `CHAR_MAX`, которое указывает на то, что значение не доступно в текущей локализации. Ниже приводится описание членов структуры:

`char *decimal_point`

Символ десятичной точки, используемый для форматирования немонетарных величин.

`char *thousands_sep`

Символ, используемый для разделения групп цифр перед символом десятичной точки в форматированных немонетарных величинах.

`char *grouping`

Строка, элементы которой обозначают размер каждой группы цифр форматированной немонетарной величины.

`char *int_curr_symbol`

Указатель на строку символов, содержащую международный знак денежной единицы, применяемый в текущей локализации. Первые три символа содержат алфавитно-цифровой международный знак денежной единицы в

соответствии с его определением в ISO 4217:1987. Четвертый символ (непосредственно предшествующий нулевому символу) используется для отделения международного знака денежной единицы от значения денежной суммы.

char *currency_symbol

Знак местной денежной единицы, применяемый в текущей локализации.

char *mon_decimal_point

Символ десятичной точки, используемый для форматирования монетарных величин.

char *mon_thousands_sep

Символ, используемый для разделения групп цифр перед символом десятичной точки в форматированных монетарных величинах.

char *mon_grouping

Строка, элементы которой обозначают размер каждой группы цифр форматированной монетарной величины.

char *positive_sign

Строка, используемая для обозначения неотрицательных форматированных монетарных данных.

char *negative_sign

Строка, используемая для обозначения отрицательных форматированных монетарных данных.

char int_frac_digits

Количество цифр дробной части (т. е. после десятичной точки) в представлении международных форматированных монетарных величин.

char frac_digits

Количество цифр дробной части (т. е. после десятичной точки) в представлении форматированных монетарных величин.

char p_cs_precedes

Значение 1 или 0 этой величины определяет расположение знака денежной единицы в неотрицательных монетарных величинах: перед денежной суммой или после нее соответственно.

char p_sep_by_space

Значение 1 или 0 этой величины определяет, отделяется или не отделяется пробелом денежный символ в неотрицательных монетарных величинах от значения денежной суммы.

char n_cs_precedes

Значение 1 или 0 этой величины определяет расположение знака денежной единицы в отрицательных монетарных величинах: перед денежной суммой или после нее соответственно.

char n_sep_by_space

Значение 1 или 0 этой величины определяет, отделяется или не отделяется пробелом денежный символ в отрицательных монетарных величинах от значения денежной суммы.

char p_sign_posn

Эта величина указывает положение знака «плюс» в представлении неотрицательных монетарных величин.

char n_sign_posn

Эта величина указывает положение знака «минус» в представлении отрицательных монетарных величин.

Элементы **grouping** и **mon_grouping** могут принимать следующие значения:

CHAR_MAX Группировка далее не выполняется.

0 Предыдущий элемент применяется для оставшихся цифр.

Любое иное целое Задает число цифр, составляющих текущую группу. Следующий элемент определяет размер последующей группы цифр, которая предшествует текущей группе.

Элементы **p_sign_posn** и **n_sign_posn** могут принимать следующие значения:

0 Круглые скобки охватывают денежный символ и саму величину.

1 Стока знака предшествует величине и денежному символу.

2 Стока знака следует за величиной и денежным символом.

3 Стока знака предшествует непосредственно денежному символу.

4 Стока знака следует непосредственно за денежным символом.

Функция **localeconv** возвращает указатель на заполненный объект. Структура, на которую ссылается возвращаемое значение, не должна изменяться программой, но может оказаться переписанной при последующем обращении к функции **localeconv**. Кроме того, в результате вызова функции **setlocale** со значениями **category LC_ALL**, **LC_MONETARY** или **LC_NUMERIC** содержимое структуры может изменяться.

A.6. Математические функции < math.h>

HUGE_VAL

Символическая константа, представляющая положительное выражение типа **double**.

double acos(double x);

Вычисляет главное значение арккосинуса аргумента **x**. Если заданный аргумент не попадает в диапазон значений $[-1, +1]$, происходит ошибка выхода за допустимые пределы области определения (**EDOM**). Функция **acos** возвращает значение арккосинуса в диапазоне $[0, \pi]$ в радианах.

double asin(double x);

Вычисляет главное значение арксинуса аргумента **x**. Если заданный аргумент не попадает в диапазон значений $[-1, +1]$, происходит ошибка выхода за пределы области определения (**EDOM**). Функция **asin** возвращает значение арксинуса в диапазоне $[-\pi/2, +\pi/2]$ в радианах.

double atan(double x);

Вычисляет главное значение арктангенса аргумента **x**. Функция **atan** возвращает значение арктангенса в диапазоне $[-\pi/2, +\pi/2]$ в радианах.

double atan2(double y, double x);

Вычисляет главное значение арктангенса выражения y/x , используя знаки обоих аргументов для определения квадранта, в котором выбирается возвращаемое значение. Если оба аргумента равны нулю, происходит ошибка выхода за пределы области определения (EDOM). Функция atan2 возвращает значение арктангенса y/x в диапазоне $[-\pi/2, +\pi/2]$ в радианах.

double cos(double x);

Вычисляет косинус аргумента x , значение которого задается в радианах.

double sin(double x);

Вычисляет синус аргумента x , значение которого задается в радианах.

double tan(double x);

Вычисляет тангенс аргумента x , значение которого задается в радианах.

double cosh(double x);

Вычисляет гиперболический косинус аргумента x . Если величина x слишком велика, происходит ошибка выхода за диапазон допустимых значений (ERANGE).

double sinh(double x);

Вычисляет гиперболический синус аргумента x . Если величина x слишком велика, происходит ошибка выхода за диапазон допустимых значений (ERANGE).

double tanh(double x);

Вычисляет гиперболический тангенс аргумента x .

double exp(double x);

Вычисляет значение показательной функции аргумента x . Если величина x слишком велика, происходит ошибка выхода за диапазон допустимых значений (ERANGE).

double frexp(double value, int *exp);

Разбивает число с плавающей запятой на нормализованную дробную часть и целую часть, равную степени числа 2. Значение степени записывается в объект типа `int`, на который ссылается параметр `exp`. Функция `frexp` возвращает значение x типа `double`, которое лежит в интервале $[1/2, 1]$ или равно нулю; в итоге значение аргумента `value` представляется в виде произведения $x \cdot 2^{\text{exp}}$. Если значение `value` равно нулю, обе части вычисляемого функцией результата также равны нулю.

double ldexp(double x, int exp);

Умножает число с плавающей точкой на число 2, возведенное в целую степень. Может возникнуть ошибка выхода за диапазон допустимых значений (ERANGE). Функция возвращает значение, равное произведению $x \cdot 2^{\text{exp}}$.

double log(double x);

Вычисляет натуральный логарифм аргумента x . В случае, если аргумент имеет отрицательное значение, происходит ошибка выхода за пределы области определения (EDOM), а если значение аргумента равно нулю, то происходит ошибка выхода за диапазон допустимых значений (ERANGE).

```
double log10(double x);
```

Вычисляет десятичный логарифм аргумента x. Если аргумент имеет отрицательное значение, происходит ошибка выхода за пределы области определения (EDOM). Если значение аргумента равно нулю, происходит ошибка выхода за диапазон допустимых значений (ERANGE).

```
double modf(double value, double *iptr);
```

Разбивает значение аргумента на целую и дробную части, каждая из которых имеет тот же знак, что и аргумент. Функция сохраняет целую часть числа как тип double в объекте, на который указывает параметр iptr; дробная часть числа со знаком используется в качестве возвращаемого функцией значения.

```
double pow(double x, double y);
```

Возводит x в степень y. Если x имеет отрицательное значение, а y не является целым числом, то происходит ошибка выхода за пределы области определения (EDOM). Если значение x равно нулю, а значение y равно нулю или меньше нуля, происходит ошибка выхода за пределы области определения (EDOM). Возможно также появление ошибки выхода за пределы диапазона допустимых значений (ERANGE).

```
double sqrt(double x);
```

Вычисляет положительное значение квадратного корня из аргумента x. Если аргумент имеет отрицательное значение, происходит ошибка выхода за пределы области определения (EDOM).

```
double ceil(double x);
```

Вычисляет наименьшее целое, значение которого не меньше, чем x.

```
double fabs(double x);
```

Вычисляет абсолютное значение числа с плавающей запятой x.

```
double floor(double x);
```

Вычисляет наибольшее целое, по значению не превосходящее x.

```
double fmod(double x, double y);
```

Вычисляет остаток от деления x на y — двух чисел с плавающей запятой.

A.7. Нелокальные переходы <setjmp.h>

jmp_buf

Тип массива для сохранения информации об окружении, необходимой для последующего восстановления окружения.

```
int setjmp (jmp_buf env);
```

Записывает состояние окружения в параметр типа jmp_buf для последующего использования функцией longjmp.

В случае прямого вызова макрос setjmp возвращает нулевое значение. В случае возврата как результата вызова функции longjmp, setjmp возвращает ненулевое значение.

Вызов макроса setjmp должен появляться только в одном из следующих контекстов:

- выражение условия в целом в операторах выбора или цикла;

- операнд операции проверки на равенство или отношения в случае, когда второй operand является целочисленным константным выражением, а все выражение в целом является выражением условия в операторах выбора или цикла;
- operand унарной операции `!`, если при этом все выражение в целом является выражением условия в операторах выбора или цикла;
- оператор выражения в целом.

```
void longjmp (jmp_buf env, int val);
```

Эта функция восстанавливает окружение, которое было сохранено в соответствующем объекте типа `jmp_buf` при последнем вызове макроса `setjmp` в пределах той же функции, из которой вызывается `longjmp`. Если такой вызов `setjmp` не был сделан или если функция, содержащая вызов `setjmp`, уже завершила свое выполнение, поведение функции `longjmp` не определено.

Все доступные объекты восстанавливают свои значения, которые они имели во время вызова `longjmp`, за исключением автоматических объектов, локальных для функции, из которой вызывался соответствующий макрос `setjmp`, и которые не определены со спецификатором `volatile` и были изменены между вызовами `setjmp` и `longjmp` — значения таких объектов не определены.

Хотя `longjmp` обходит обычные механизмы вызова функции и возврата из функции, `longjmp` работает корректно в контексте прерываний, сигналов и функций, связанных с ними. Однако, если функция `longjmp` вызывается из вложенного обработчика сигналов (то есть из функции, вызванной в результате сигнала, посланного во время обработки другого сигнала), то ее поведение не определено.

После завершения `longjmp` выполнение программы продолжается так, как если бы соответствующий вызов макроса `setjmp` возвратил значение, определяемое параметром `val`. Функция `longjmp` не может заставить макрос `setjmp` вернуть значение 0; если `val` равно 0, `setjmp` возвратит значение 1.

A.8. Обработка сигналов <signal.h>

`sig_atomic_t`

Целочисленный тип, объекты которого доступны даже при асинхронных прерываниях.

`SIG_DFL`
`SIG_ERR`
`SIG_IGN`

Это символические константы (см. описание функции `signal`, поясняющее их смысл), тип которых совместим с типом второго параметра и возвращаемого значения функции `signal` и значения которых не должны совпадать с адресом ни одной из объявленных функций. Каждая из констант принимает положительное целочисленное значение, представляющее *номер сигнала* соответствующего типа:

<code>SIGABRT</code>	аварийное завершение программы, например в результате вызова функции <code>abort</code>
<code>SIGFPE</code>	ошибочная арифметическая операция, например, операция деления на нуль или операция, вызвавшая переполнение
<code>SIGILL</code>	обнаружение ошибочного образа функции, например, недопустимой команды
<code>SIGINT</code>	получение интерактивного сигнала

SIGSEGV	ошибка обращения к памяти
SIGTERM	посланный программе запрос о завершении работы

Программа не должна генерировать ни один из этих сигналов иначе, как явным вызовом функции `raise`.

```
void (*signal(int sig, void (*func) (int) ) ) (int);
```

Функция определяет один из трех возможных способов реакции на получение сигнала с номером `sig`. Если для параметра `func` используется значение `SIG_DFL`, то для такого сигнала будет использоваться обработка по умолчанию. Если значение `func` равно `SIG_IGN`, то сигнал будет игнорироваться. В остальных случаях, параметр `func` должен указывать на функцию, которая вызывается для обработки сигнала. Такая функция называется *обрабочиком сигнала*.

Если `func` указывает на функцию, то при возникновении сигнала сначала выполняются действия, эквивалентные вызову `signal(sig, SIG_DFL)`; , т.е. происходит перехват сигнала обработчиком системы по умолчанию. (Если значение `sig` равно `SIGILL`, то будет или не будет вызываться обработчик по умолчанию зависит от реализации системы.) После этого выполняется вызов `(* func) (sig);`. Выполнение функции `func` может завершаться оператором `return` или вызовом функций `abort`, `exit` или `longjmp`. Если функция `func` завершается оператором `return` и при этом значение параметра `sig` равно `SIG_FPE` или любому другому определяемому реализацией значению, соответствующему исключению, связанному с ошибкой в вычислениях, то результат не определен. В остальных случаях, программа продолжит выполнение с точки, в которой произошло прерывание.

В случае, если сигнал посыпается не в результате вызова функций `abort` и `raise`, то результат не определен, если обработчик сигнала вызывает любую функцию из стандартной библиотеки, кроме функции `signal` (со значением передаваемого в первом аргументе номера сигнала, соответствующего сигналу, который вызвал обработчик) или обращается к любому объекту типа `static`, если только он не объявлен как тип `volatile sig_atomic_t`. Кроме того, если такой вызов функции `signal` завершается возвратом значения `SIG_ERR`, то значение `errno` не определено.

При запуске системы, могут выполняться действия, эквивалентные вызовам

```
signal (sig, SIG_IGN);
```

для некоторого набора сигналов, установленных в данной системе программирования; для всех остальных сигналов, определенных в реализации системы, выполняются операции, эквивалентные вызовам

```
signal (sig, SIG_DFL);
```

Если запрос на обработку сигнала завершился успешно, то функция `signal` возвращает значение `func`, переданное функции `signal` в самом последнем обращении к ней для данного номера сигнала `sig`. В противном случае, возвращается значение `SIG_ERR` и `errno` присваивается положительное значение.

```
int raise(int sig);
```

Функция `raise` посылает сигнал `sig` выполняющейся программе. В случае успеха функция `raise` возвращает нуль, в случае неудачи — ненулевое значение.

A.9. Работа с функциями с переменным числом аргументов `<stdarg.h>`

`va_list`

Тип, предназначенный для хранения информации, необходимой макросам `va_start`, `va_arg` и `va_end`. Для того, чтобы использовать возможность вызова функции с переменным числом аргументов, вызываемая функция должна объявлять объект типа `va_list`, называемый в этом разделе `ар`. Объект `ар` может быть передан в качестве аргумента другой функции; если эта функция вызывает макрос `va_arg` с аргументом `ар`, то в вызывающей функции значение `ар` должно быть определено и в дальнейшем передано макросу `va_end`.

`void va_start(va_list ap, parmN);`

Этот макрос должен вызываться перед обработкой списка с переменным числом параметров в вызове функции. Макрос `va_start` инициализирует объект `ар` для последующего использования макросами `va_arg` и `va_end`. Параметр `parmN` задает имя самого правого из определенных параметров в списке в объявлении функции (это тот параметр, за которым сразу следует многоточие ...). Если параметр `parmN` объявлен с классом памяти `register`, или имеет тип функции или массива, или несовместим с типом, получаемым в результате определяемого по умолчанию продвижения параметров по шкале типов, то поведение макроса не определено.

`type va_arg(va_list ap, type);`

Макрос, расширяющийся до выражения, которое имеет тип и значение следующего параметра в списке. Параметр `ар` должен быть тем самым `ар`, который был инициализирован макросом `va_start`. Каждый вызов `va_arg` изменяет значение `ар` таким образом, чтобы в следующем вызове `va_arg` возвращалось значение следующего параметра. Параметр `type` задает имя типа ожидаемого параметра из списка параметров; это имя, за которым следует символ звездочки (*), используется для формирования указателя нужного типа, ссылающегося на очередной объект в списке параметров. Если параметров больше нет или если `type` несовместим с типом следующего параметра (в результате задаваемого по умолчанию приведения типов), то результат не определен. В первом после вызова `va_start` вызове макроса `va_arg` возвращается значение параметра, следующего за параметром, определенным в `parmN`. Все остальные вызовы `va_arg` последовательно возвращают значения оставшихся параметров.

`void va_end(va_list ap);`

Обеспечивает нормальный возврат из функции, в которой осуществлялась работа со списком параметров переменной длины путем создания макросом `va_start` объекта `ар` типа `va_list`. Макрос `va_end` может изменять объект `ар` таким образом, что он не будет больше пригоден для использования (пока не будет сделан новый вызов `va_start`). Если не был сделан соответствующий вызов `va_start` или перед возвратом из функции не был вызван `va_end`, то результат обработки списка параметров не определен.

A.10. Ввод-вывод <stdio.h>

_IOFBF
_IOLBF
_IONBF

Макросы, расширяемые до целочисленных константных выражений, имеющих различные значения и используемых в качестве третьего параметра функции `setvbuf`.

BUFSIZ

Целочисленное константное выражение, задающее размер буфера и используемое функцией `setbuf`.

EOF

Отрицательное целочисленное константное выражение, возвращаемое некоторыми функциями для обозначения того, что достигнут конец файла, т. е. что в потоке ввода нет больше данных.

FILE

Тип объекта, используемого для записи всей информации, необходимой для управления потоком, включая индикатор текущей позиции в файле, указатель на связанный с потоком буфер (если он имеется), индикатор ошибки чтения-записи и индикатор конца файла, показывающий, был ли достигнут конец файла или нет.

FILENAME_MAX

Целочисленное константное выражение, которое задает размер массива типа `char`, достаточный для хранения самой длинной строки с именем файла, который данная система гарантированно может открыть.

FOPEN_MAX

Целочисленное константное выражение, которое определяет минимальное число файлов, которое система гарантированно может открыть одновременно.

fpos_t

Тип объекта, который может быть использован для записи всей информации, необходимой для определения текущей позиции в файле.

L_tmpnam

Целочисленное константное выражение, которое задает размер массива типа `char`, достаточный для хранения строки с именем файла, генерируемой функцией `tmpnam`.

NULL

Константа нулевого указателя, зависящая от используемой системы.

SEEK_CUR
SEEK_END
SEEK_SET

Целочисленные константные выражения с различающимися значениями, используемые в качестве третьего параметра функции `fseek`.

size_t

Целый без знака тип результата операции `sizeof`.

stderr

Выражение типа «указатель на FILE», которое ссылается на объект типа FILE, связанный со стандартным потоком ошибок.

stdin

Выражение типа «указатель на FILE», которое ссылается на объект типа FILE, связанный со стандартным потоком ввода.

stdout

Выражение типа «указатель на FILE», которое ссылается на объект типа FILE, связанный со стандартным потоком вывода.

TMP_MAX

Целочисленное константное выражение, которое определяет минимальное число уникальных имен файлов, генерируемых функцией `tmpnam`. Значение макроса `TMP_MAX` должно быть равно по крайней мере 25.

int remove(const char *filename);

После вызова этой функции файл, имя которого задается параметром `filename`, становится недоступен. Последующая попытка открыть файл с этим именем закончится неудачей, если только файл не был создан заново. Если файл открыт, то поведение функции `remove` зависит от реализации системы. В случае успешного выполнения функция возвращает нулевое значение, в случае неудачи возвращается ненулевое значение.

int rename(const char *old, const char *new);

Изменяет имя файла, на которое ссылается указатель `old`, на новое имя, на которое ссылается указатель `new`. Файл под именем `old` становится недоступен. Если файл, на который ссылается указатель `new`, уже существовал до вызова функции `rename`, то результат будет зависеть от системы программирования. Функция `rename` возвращает нуль, если она завершилась успешно. В случае неудачи возвращается результат, отличный от нуля; при этом, если переименовываемый файл существовал ранее, то он остается под своим первоначальным именем.

FILE *tmpfile(void);

Создает временный бинарный файл, который будет автоматически удален при своем закрытии или по завершении программы. Если программа завершается аварийно, то будет ли удаляться открытый временный файл зависит от реализации системы. Файл открывается для обновления с режимом доступа «`wb+`». Функция `tmpfile` возвращает указатель на поток созданного файла. Если файл не может быть создан, функция `tmpfile` возвращает нулевой указатель.

char *tmpnam(char *s);

Функция `tmpnam` генерирует строку символов, которая может использоваться в качестве имени файла; при этом гарантируется, что файл с таким именем не существует. Функция `tmpnam` генерирует различающиеся строки при каждом своем вызове до `TMP_MAX` раз. Если функция вызывается больше, чем `TMP_MAX` раз, то результат зависит от системы программирования.

Если в качестве аргумента используется нулевой указатель, то функция `tmpnam` помещает результат во внутренний статический объект и возвращает указатель на этот объект. Последующие обращения к функции `tmpnam` могут изменять этот объект. Если аргумент не равен указателю нуль, то предполагается, что он ссылается на символьный массив размером не менее

`L_tmpnam`; функция `tmpnam` записывает результат в этот массив и возвращает ссылку на него.

```
int fclose(FILE *stream);
```

Функция `fclose` очищает поток, на который ссылается указатель `stream`, и закрывает связанный с ним файл. Все еще не записанные буферизованные данные потока, переданные в исполняющую систему, записываются в файл; любые еще не прочитанные буферизованные данные теряются. Поток отключается от файла. Если память под связанный с потоком буфер отводилась автоматически, то она освобождается. Функция `fclose` возвращает нуль, если поток был успешно закрыт, или `EOF`, если были обнаружены какие-либо ошибки.

```
int fflush(FILE *stream);
```

Если параметр `stream` ссылается на поток вывода или поток модификации, в котором не завершилась последняя операция, то вызов функция `fflush` приводит к тому, что все еще не записанные данные потока передаются исполняющей системе или записываются в файл; в других случаях, результат не определен.

Если значение `stream` — нулевой указатель, то функция `fflush` выполняет выталкивание данных из буферов всех потоков указанных выше типов. Функция `fflush` возвращает `EOF`, если происходит ошибка записи, иначе возвращается нуль.

```
FILE *fopen(const char *filename, const char *mode);
```

Функция `fopen` открывает файл с именем в виде строки, на которую ссылается указатель `filename`, и связывает с ним поток. Аргумент `mode` указывает на строку, начало которой может содержать:

<code>r</code>	открыть текстовый файл для чтения
<code>w</code>	сократить существующий файл до нулевой длины или создать текстовый файл для записи
<code>a</code>	добавить, что означает: открыть или создать текстовый файл для записи данных в конец файла
<code>rb</code>	открыть бинарный файл для чтения
<code>wb</code>	сократить существующий файл до нулевой длины или создать бинарный файл для записи
<code>ab</code>	добавить, что означает: открыть или создать бинарный файл для записи данных в конец файла
<code>r+</code>	открыть текстовый файл для обновления (чтения и записи)
<code>w+</code>	сократить существующий файл до нулевой длины или создать текстовый файл для обновления
<code>a+</code>	добавить, что означает: открыть или создать текстовый файл для обновления, причем записи данных заносить в конец файла
<code>r+b</code> или <code>rb+</code>	открыть бинарный файл для обновления (чтения и записи)
<code>w+b</code> или <code>wb+</code>	сократить файл до нулевой длины или создать бинарный файл для обновления
<code>a+b</code> или <code>ab+</code>	добавить, что означает: открыть или создать бинарный файл для обновления, причем записи данных заносить в конец файла

При открытии файла для чтение (первым символом аргумента *mode* является символ «г») происходит ошибка, если файл не существует или данные не могут быть прочитаны. Открытие файла для обновление (первым символом аргумента *mode* является символ «а») означает, что все последующие записи в файл будут производиться в текущий конец файла, независимо от вызовов функции *fseek*. В некоторых реализациях систем при открытии бинарного файла на обновление (вторым или третьим символом в параметре *mode* является символ «б»), индикатор позиции файла первоначально может быть установлен за последними записанными данными; это происходит из-за добавления нулевого символа.

При открытии файла и режиме обновления (вторым или третьим символом в параметре *mode* является символ «+») могут выполняться как операции ввода, так и операции вывода. Однако, за операцией вывода не может непосредственно следовать операция ввода без промежуточного вызова функции *fflush* или функций изменения текущей позиции в файле (*fseek*, *fsetpos* или *rewind*), а за вводом данных не может непосредственно следовать вывод без обращения к функции изменения текущей позиции в файле, кроме случая, когда в результате операции ввода индикатор текущей позиции в файле был установлен на конец файла. В некоторых системах вместо открытия или создания тестового файла в режиме обновления может открываться или создаваться бинарный файл.

Открываемый поток полностью буферизуется в том и только в том случае, если он не относится к интерактивному устройству. Индикаторы конца файла и ошибки в открываемом потоке очищаются. Функция *fopen* возвращает указатель на объект, управляющий этим потоком. Если попытка открыть файл закончилась неудачей, *fopen* возвращает нулевой указатель.

```
FILE *freopen(const char *filename, const char *mode,
              FILE *stream);
```

Функция *freopen* открывает файл с именем в виде строки, на которую ссылается указатель *filename*, и связывает с ним поток, на который ссылается указатель *stream*. Параметр *mode* используется точно так же, как и в функции *fopen*.

Функция *freopen* сначала пытается закрыть файл, который связан с заданным потоком. Если закрытие файла произошло с ошибкой, то она игнорируется. Индикаторы конца файла и ошибки открываемого потока очищаются. В случае неудачного открытия файла *freopen* возвращает нулевой указатель. При удачном открытии файла *freopen* возвращает значение *stream*.

```
void setbuf(FILE *stream, char *buf);
```

Функция *setbuf* эквивалентна функции *setvbuf*, вызываемой со значениями *_IOFBF* для *mode* и *BUFSIZ* для *size*, или (если значение *buf* — нулевой указатель) со значением *_IONBF* для *mode*. Функция *setbuf* не возвращает никакого значения.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Функция *setvbuf* может использоваться только после того, как поток, на который ссылается *stream*, был связан с открытым файлом, и прежде, чем над потоком будет выполняться какая-либо иная операция. Параметр *mode* определяет буферизацию *stream* следующим образом: значение *_IOFBF* означает, что ввод-вывод будет полностью буферизованным; *_IOLBF* означает, что ввод-вывод будет буферизоваться по строкам; *_IONBF* — небуферизованный ввод-вывод. Если параметр *buf* не является нулевым указателем, массив, на который он ссылается, может использоваться вместо буфера, выделяемого функцией *setvbuf*. Размер этого массива определяется параметром *size*. Со-

держимое массива в любой момент времени не определено. Функция **setvbuf** возвращает нуль при успешном завершении или значение, отличное от нуля, если для параметра **mode** было указано недопустимое значение или если функция не может выполнить запрос.

```
int fprintf(FILE *stream, const char *format, ...);
```

Функция **fprintf** выводит данные в поток, на который ссылается параметр **stream**; вывод управляется строкой форматирования, на которую ссылается параметр **format** и которая задает способ преобразования заданных аргументов при выводе. Если аргументов меньше, чем указано в строке формата, поведение функции не определено. Если строка формата исчерпана в то время, как параметры еще остаются, то лишние параметры обычно вычисляются, но иногда могут игнорироваться. Функция **fprintf** заканчивает свою работу при достижении конца строки формата. См. главу 11, в которой приводится детальное описание спецификаций форматирования. Функция **fprintf** возвращает число выведенных символов или отрицательное значение, если произошла ошибка вывода.

```
int fscanf(FILE *stream, const char *format, ...);
```

Функция **fscanf** вводит данные из потока, на который ссылается параметр **stream**, под управлением строки форматирования, на которую ссылается параметр **format** и которая определяет допустимые входные последовательности и задает способы преобразования при присваивании значений всех последующих аргументов, которые должны быть указателями на объекты, получающие форматированные входные данные. Если параметров недостаточно для строки формата, поведение функции не определено. Если строка формата исчерпана в то время, как параметры еще остаются, то лишние параметры обычно вычисляются, но иногда могут игнорироваться. См. главу 11, в которой приводится детальное описание спецификаций форматирования.

Функция **fscanf** возвращает значение **EOF**, если произошла ошибка ввода до выполнения форматирования. В противном случае, функция **fscanf** возвращает число введенных элементов данных, которое может быть меньше, чем предусмотрено списком параметров, или даже равно нулю в случае ошибок согласования типов.

```
int printf(const char *format, ...);
```

Функция **printf** эквивалентна функции **fprintf**, если вызвать **fprintf** с параметром **stdout** и списком параметров функции **printf**. Функция **printf** возвращает число выведенных символов или отрицательное значение, если произошла ошибка вывода.

```
int scanf(const char *format, ...);
```

Функция **scanf** эквивалентна функции **fscanf**, если вызвать **fscanf** с параметром **stdin** и списком параметров функции **scanf**. Функция **scanf** возвращает значение макроса **EOF**, если произошла ошибка ввода до форматирования данных. Иначе **scanf** возвращает число присвоенных значений, которое может быть меньше, чем предусмотрено, и даже равно нулю в случае ошибок согласования типов.

```
int sprintf (char *s, const char *format, ... );
```

Функция **sprintf** эквивалентна **fprintf** за исключением того, что результат вывода помещается не в поток, а в массив, на который ссылается параметр **s**. В конец строки записанных символов помещается нулевой символ; он не входит в подсчет числа символов, возвращаемого функцией. Результат копирования перекрывающихся объектов не определен. Функция **sprintf** воз-

возвращает число символов, записанных в массив, не считая завершающего нулевого символа.

```
int sscanf(const char *s, const char *format, ...);
```

Функция `sscanf` эквивалентна `fscanf` за исключением того, что параметр `s` определяет не поток, а строку, из которой происходит ввод данных. Достигение конца строки эквивалентно обнаружению конца файла для функции `fscanf`. Результат копирования перекрывающихся объектов не определен.

Функция `sscanf` возвращает значение макроса `EOF`, если произошла ошибка ввода до форматирования данных. Иначе функция `sscanf` возвращает число присвоенных значений, которое может быть меньше, чем предусмотрено, и даже равно нулю в случае ошибок согласования типов.

```
int vfprintf (FILE *stream, const char *format,
               va_list arg);
```

Функция `vfprintf` эквивалентна `fprintf`, в которой список параметров переменной длины заменен параметром `arg`, инициализированным макросом `va_start` (и возможно измененный последующими вызовами `va_arg`).

Функция `vfprintf` не вызывает макрос `va_end`. Функция `vfprintf` возвращает число переданных символов или отрицательное значение, если произошла ошибка вывода.

```
int vprintf (const char *format, va_list arg);
```

Функция `vprintf` эквивалентна функции `printf`, но вместо списка параметров переменной длины используется параметр `arg`, который должен быть инициализирован макросом `va_start` (и может быть изменен последующими вызовами `va_arg`). Функция `vprintf` не вызывает макрос `va_end`. Функция `vprintf` возвращает число переданных символов или отрицательное значение, если произошла ошибка вывода.

```
int vsprintf(char *s, const char *format, va_list arg);
```

Функция `vsprintf` эквивалентна функции `sprintf`, но вместо списка параметров переменной длины используется параметр `arg`, который должен быть инициализирован макросом `va_start` (и возможно последующими вызовами `va_arg`). Функция `vsprintf` не вызывает макрос `va_end`. Результат копирования перекрывающихся объектов не определен. Функция `vsprintf` возвращает число символов, записанных в массив, не считая завершающего нулевого символа.

```
int fgetc(FILE *stream);
```

Функция `fgetc` берет из указанного параметром `stream` потока ввода очередной символ (если он имеется) как `unsigned char`, преобразует его к типу `int` и перемещает связанный с файлом индикатор текущей позиции в файле (если этот индикатор определен). Функция `fgetc` возвращает очередной символ из входного потока, указанного параметром `stream`. При достижении конца файла в потоке устанавливается индикатор конца файла и `fgetc` возвращает `EOF`. Если происходит ошибка чтения, то для потока устанавливается индикатор ошибки и `fgetc` возвращает `EOF`.

```
char *fgets(char *s, int n, FILE *stream);
```

Функция `fgets` считывает из потока `stream` в массив, на который ссылается параметр `s`, не более `n-1` символа. Ввод символов прекращается, как только встретится символ новой строки (который сохраняется) или символ конца файла. За последним введенным в строку символом добавляется нулевой символ.

В случае успеха функция `fgets` возвращает строку `s`. Если достигнут конец файла и в массив не введен ни один символ, содержимое массива остается неизменным и возвращается нулевой указатель. Если в процессе выполнения функции происходит ошибка ввода, то содержимое массива не определено, а функция возвращает нулевой указатель.

```
int fputc(int c, FILE *stream);
```

Функция `fputc` записывает символ, определенный параметром `c` и преобразуемый к типу `unsigned char`, в поток вывода, указанный `stream`, в позицию, указанную соответствующим индикатором связанного с потоком файла (если этот индикатор определен), и изменяет значение индикатора текущей позиции файла. Если файл не поддерживает возможности позиционирования или если поток был открыт в режиме добавления, символы добавляются в конец потока вывода. Функция `fputc` возвращает выведенный символ. Если происходит ошибка записи, в потоке устанавливается индикатор ошибки и `fputc` возвращает значение `EOF`.

```
int fputs(const char *s, FILE *stream);
```

Функция `fputs` записывает строку `s` в поток, указанный параметром `stream`. Завершающий нулевой символ в поток не записывается. Функция `fputs` возвращает `EOF` при возникновении ошибки записи; в противном случае она возвращает неотрицательное значение.

```
int getc(FILE *stream);
```

Функция `getc` эквивалентна `fgetc` за исключением того, что если она реализуется в виде макроса, то значение `stream` может вычисляться неоднократно и при этом аргумент должен быть выражением без побочных эффектов.

Функция `getc` возвращает очередной символ из входного потока, указанного параметром `stream`. Если в потоке достигается конец файла, то устанавливается индикатор конца файла и `getc` возвращает `EOF`. Если происходит ошибка чтения, в потоке устанавливается индикатор ошибки и `getc` возвращает `EOF`.

```
int getchar(void);
```

Функция `getchar` эквивалентна `getc` со значением параметра `stdin`. Функция `getchar` возвращает очередной символ из входного потока `stdin`. Если в потоке достигается конец файла, в нем устанавливается индикатор конца файла и `getchar` возвращает `EOF`. Если происходит ошибка чтения, в потоке устанавливается индикатор ошибки и `getchar` возвращает `EOF`.

```
char *gets(char *s);
```

Функция `fgets` считывает символы из потока ввода `stdin` в массив, на который ссылается параметр `s`, до тех пор, пока не встретится символ конца файла или новой строки. Символ новой строки отбрасывается; за последним введенным в строку символом помещается нулевой символ. В случае успеха функция `gets` возвращает строку `s`. Если достигнут конец файла и в массив не введен ни один символ, содержимое массива остается неизмененным и возвращается нулевой указатель. Если в процессе выполнения функции происходит ошибка ввода, то содержимое массива не определено, а функция возвращает нулевой указатель.

```
int putc(int c, FILE *stream);
```

Функция `putc` эквивалентна `fputc` за исключением того, что если она реализуется в виде макроса, то значение `stream` может вычисляться неоднократно и при этом аргумент должен быть выражением без побочных эффектов.

Функция **putc** возвращает выведенный символ. Если происходит ошибка записи, в потоке устанавливается индикатор ошибки и **putc** возвращает **EOF**.

```
int putchar (int c);
```

Функция **putchar** эквивалентна **putc** со значением второго параметра **stdout**. Функция **putchar** возвращает выведенный символ. Если происходит ошибка записи, в потоке устанавливается индикатор ошибки и **putchar** возвращает **EOF**.

```
int puts(const char *s);
```

Функция **puts** записывает строку **s** в поток **stdout** и добавляет в поток вывода символ новой строки. Завершающий нулевой символ в поток не записывается. Функция **puts** возвращает **EOF**, если возникла ошибка записи; в остальных случаях она возвращает неотрицательное значение.

```
int ungetc(int c, FILE *stream);
```

Функция **ungetc** возвращает символ, определенный параметром **c** и преобразованный к типу **unsigned char**, обратно в поток ввода, на который указывает **stream**. Возвращенные в поток символы могут быть получены обратно последующими операциями чтения в последовательности, обратной последовательности их возвращения в поток. Если в интервале между возвратом символов и их последующим чтением происходит успешное обращение к функциям изменения текущей позиции в файле, связанном с потоком **stream** (к функциям **fseek**, **fsetpos** или **rewind**), все возвращенные в поток символы теряются. Данные во внешней памяти, соответствующие потоку, остаются неизменными.

Гарантируется возврат только одного символа. Если функция **ungetc** вызывается слишком много раз для одного и того же потока без промежуточных операций чтения или позиционирования файла этого потока, то операция возвращения символа может потерпеть неудачу. Если значение **c** равно значению макроса **EOF**, операция заканчивается неудачей и входной поток остается неизменным.

Успешное обращение к функции **ungetc** очищает индикатор конца файла потока. Значение индикатора текущей позиции в файле для этого потока после чтения или отбрасывания всех возвращенных символов должно быть равно тому значению этого индикатора, которое он имел до того, как символы были возвращены. Для текстового потока значение индикатора текущей позиции в файле после успешного обращения к функции **ungetc** не определено до того момента, пока все возвращенные в поток символы не будут прочитаны или отброшены. Для бинарного потока индикатор текущей позиции в файле определен после каждого успешного обращения к функции **ungetc**; если его значение равнялось нулю перед вызовом **ungetc**, то значение после вызова будет не определено. Функция **ungetc** возвращает вытолкнутый в поток символ, преобразовав его, или **EOF**, если операция терпит неудачу.

```
size_t fread(void *ptr, size_t size, size_t nmem,
            FILE *stream);
```

Функция **fread** считывает в массив, на который указывает **ptr**, не более **nmem** элементов размером **size** каждый из потока, на который указывает **stream**. Индикатор текущей позиции в файле (если он определен для потока) передвигается на число успешно прочитанных символов. В случае возникновения ошибки значение индикатора текущей позиции в файле для данного потока становится неопределенным. Если элемент прочитан частично, его значение не определено.

Функция `fread` возвращает число успешно прочитанных элементов данных, которое может быть меньше, чем `nmemb` в случае возникновения ошибок чтения или обнаружения конца файла. Если `size` или `nmemb` равны нулю, то `fread` возвращает нуль, а содержимое массива и состояние потока остаются без изменений.

```
size_t fwrite (const void *ptr, size_t size, size_t nmemb,
FILE *stream);
```

Функция `fwrite` записывает из массива, на который указывает `ptr`, до `nmemb` элементов размером `size` каждый в поток, на который указывает `stream`. Индикатор текущей позиции в файле (если он определен для потока) передвигается на число успешно записанных символов. В случае возникновения ошибки значение индикатора текущей позиции в файле для данного потока становится неопределенным. Функция `fwrite` возвращает число успешно записанных элементов, которое может быть меньше, чем `nmemb`, только в случае возникновения ошибок записи.

```
int fgetpos(FILE *stream, fpos_t *pos);
```

Функция `fgetpos` сохраняет значение индикатора текущей позиции в файле потока, на который указывает `stream`, в объекте, на который указывает `pos`. Сохраненное значение содержит неопределенную информацию, пригодную для использования функцией `fsetpos` при обратном позиционировании потока к позиции, полученной в результате вызова функции `fgetpos`. В случае успеха функция `fgetpos` возвращает нуль, при неудачном завершении функция `fgetpos` возвращает отличное от нуля значение и помещает определяемое реализацией системы положительное значение в `errno`.

```
int fseek(FILE *stream, long int offset, int whence);
```

Функция `fseek` устанавливает индикатор текущей позиции в файле для потока, указанного параметром `stream`. Для бинарного потока эта новая позиция, измеряемая в числе символов от начала файла, получается добавлением параметра `offset` к значению позиции, которое определяется параметром `whence`. Определяемая параметром `whence` позиция — это: начало файла, если значение `whence` равно `SEEK_SET`; текущее значение индикатора позиции файла, если значение `whence` равно `SEEK_CUR`; конец файла для значения `SEEK_END`. Для бинарного потока нет необходимости явно вызывать `fseek` со значением `SEEK_END` параметра `whence`. Для текстового потока значение `offset` должно быть равно либо нулю, либо значению, которое вернула функция `ftell` в предшествующем вызове для того же самого `stream`, а значение `whence` должно быть равно `SEEK_SET`.

Успешный вызов функции `fseek` очищает индикатор конца файла потока и отменяет результаты вызова функции `ungetc` для этого потока. После вызова `fseek` следующей операцией в потоке, открытом для обновления, может быть как операция ввода, так и операция вывода. Функция `fseek` возвращает отличный от нуля результат только для запроса, который не может быть выполнен.

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Функция `fsetpos` устанавливает значение индикатора текущей позиции файла в потоке, на который указывает `stream`, в соответствии со значением объекта, на который ссылается параметр `pos` и которое должно быть получено в результате предшествующего вызова функции `fgetpos` для того же потока. Успешный вызов функции `fsetpos` очищает индикатор конца файла для потока и отменяет результаты вызова функции `ungetc` для того же потока. После вызова `fseek` следующей операцией в потоке, открытом для обновления, может

быть как операция ввода, так и операция вывода. Функция `fseek` возвращает нуль в случае своего успешного завершения и отличный от нуля результат для запроса, который не может быть выполнен, помещая при этом в `errno` определяемое реализацией системы значение.

```
long int ftell(FILE *stream);
```

Функция `ftell` возвращает значение индикатора текущей позиции в файле для указанного потока `stream`. Возвращаемое значение для бинарного потока равно числу символов от начала файла. Для текстового потока индикатор позиции файла содержит неопределенную информацию, которую может использовать функция `fseek` для возвращения индикатора текущей позиции в файле к значению позиции, полученному в результате вызова `ftell`; разность между двумя такими возвращаемыми значениями не обязательно равна числу записанных или прочитанных символов. В случае успешного завершения `ftell` возвращает значение индикатора текущей позиции в файле. При возникновении ошибки `ftell` возвращает значение `-1L` и помещает в `errno` определенное в реализации системы программирования положительное значение.

```
void rewind(FILE *stream);
```

Функция `rewind` устанавливает индикатор текущей позиции в файле для потока, указанного параметром `stream`, на начало файла. Вызов этой функции эквивалентен вызову

```
(void) fseek(stream, 0L, SEEK_SET);
```

за исключением того, что индикатор ошибки потока также сбрасывается.

```
void clearerr(FILE *stream);
```

Функция `clearerr` сбрасывает индикаторы конца файла и ошибки для потока, на который указывает `stream`.

```
int feof(FILE *stream);
```

Функция `feof` проверяет состояние индикатора конца файла для потока, на который ссылается `stream`. Функция `feof` возвращает ненулевое значение в том и только в том случае, если индикатор конца файла для этого потока показывает конец файла.

```
int ferror(FILE *stream);
```

Функция `ferror` проверяет состояние индикатора ошибки потока, на который ссылается `stream`. Функция `ferror` возвращает ненулевое значение только тогда, когда индикатор ошибки в потоке показывает наличие ошибки.

```
void perror(const char *s);
```

Функция `perror` выводит сообщение об ошибке, соответствующее текущему значению номера ошибки, определяемого как целое значение `errno`. Функция выводит в стандартный поток ошибок следующую последовательность символов: сначала выводится строка, на которую ссылается указатель `s` (если `s` — не нулевой указатель и символ, на который ссылается `s`, не нулевой символ), затем выводятся двоеточие (`:`) и пробел, а затем выводится соответствующая строка сообщения об ошибке, завершаемая символом новой строки. Содержимое строк сообщения об ошибках зависит от реализации системы и совпадает со значением, возвращаемым функцией `strerror` при вызове ее с аргументом `errno`.

A.11. Утилиты общего назначения <stdlib.h>

EXIT_FAILURE
EXIT_SUCCESS

Символические константы, которые могут использоваться в качестве параметра функции `exit` для обозначения кода выхода из программы — успешного или неудачного, возвращаемого среди выполнения программы.

MB_CUR_MAX

Положительное целочисленное выражение, значение которого задает максимальное число байтов в многобайтовом символе для расширенного набора символов, определяемого текущей локализацией (категория `LC_CTYPE`); это значение не должно превышать значения `MB_LEN_MAX`.

NULL

Константа нулевого указателя, зависящая от используемой системы.

RAND_MAX

Целочисленное константное выражение, значение которого задает максимальное значение, возвращаемое функцией `rand`. Значение `RAND_MAX` должно быть равно по крайней мере 32767.

div_t

Тип структуры, возвращаемой функцией `div`.

ldiv_t

Тип структуры, возвращаемой функцией `ldiv`.

size_t

Целый без знака тип результата операции `sizeof`.

wchar_t

Тип целочисленных данных, диапазон значений которых обеспечивает представление кодов всех элементов наибольшего расширенного набора символов среди поддерживаемых имеющимися локализациями; значение кода нулевого символа должно быть равно нулю, а каждый символ из основного набора должен иметь значение кода, равное значению при использовании этого символа в целочисленной символьной константе.

double atof(const char *nptr);

Преобразует начальную часть строки символов, на которую ссылается `nptr`, в число типа `double`. Функция `atof` возвращает результат преобразования.

int atoi(const char *nptr);

Преобразует начальную часть строки символов, на которую ссылается `nptr`, в число типа `int`. Функция `atoi` возвращает результат преобразования.

long int atol(const char *nptr);

Преобразует начальную часть строки символов, на которую ссылается `nptr`, в число типа `long`. Функция `atol` возвращает результат преобразования.

double strtod(const char *nptr, char **endptr);

Преобразует строку символов, на которую ссылается `nptr`, в число типа `double`. Сначала функция делит строку на три части: начальную, возможно пус-

тую, последовательность символов разделителей (понимаемых в соответствии с классификацией функции `isspace`), основную последовательность, представляющую константу с плавающей точкой как таковую, и конечную часть, состоящую из одного или более нераспознаваемых символов, включая завершающий нулевой символ строки ввода. Затем функция пытается преобразовать основную последовательность строки в число с плавающей запятой и возвращает полученное значение.

Расширенная форма основной последовательности включает необязательный знак «плюс» или знак «минус», за которым следует непустая последовательность цифр, которая может содержать символ десятичной точки, затем может следовать необязательная часть порядка числа; суффиксы числа с плавающей точкой не распознаются. Основная последовательность определяется как максимально длинная подстрока начальной части входной строки, начинающаяся с первого значащего символа (не разделителя) и соответствующая ожидаемой форме. Основная последовательность не содержит символов, если строка ввода пустая, или целиком состоит из символов разделителей, или если первый значащий символ не является знаком «плюс» или «минус», цифрой или символом десятичной точки.

Если основная последовательность имеет ожидаемую форму: последовательность символов, начинающуюся с цифры или десятичной точки (что бы из них ни встретилось первым), то она интерпретируется как константа с плавающей точкой. В случае, если символ десятичной точки используется в качестве точки в конце предложения, за которой не следует степенная или дробная части числа, то предполагается, что точка поставлена за последней цифрой в строке. Если последовательность начинается со знака «минус», то значение, получаемое в результате преобразования, является отрицательным числом. В объект, на который ссылается `endptr`, помещается указатель на конец обработанной подстроки, если только `endptr` не нулевой указатель.

Если основная последовательность пуста или имеет не ожидаемую форму, преобразование не производится; в объект, на который ссылается `endptr`, помещается значение `ptr` при условии, что `endptr` не является нулевым указателем.

В случае успешного преобразования функция `strtod` возвращает преобразованное число. Если преобразование не может быть выполнено, возвращается нуль. Если результат преобразования выходит за границы диапазона допустимых значений, возвращается положительное или отрицательное значение `HUGE_VAL` (в соответствии со знаком результата) и в `errno` помещается значение макроса `ERANGE`. Если в процессе вычисления результата происходит ошибка потери значимости, возвращается нуль и в `errno` помещается значение макроса `ERANGE`.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Преобразует начальную часть строки, на которую ссылается `ptr`, в число типа `long int`. Сначала функция делит строку на три части: начальную последовательность, возможно пустую, символов разделителей (понимаемых в соответствии с классификацией функции `isspace`), основную последовательность, представляющую целую константу как таковую в системе счисления, указанной параметром `base`, и конечную часть, состоящую из одного или более нераспознаваемых символов, включая завершающий нулевой символ строки. Затем, функция пытается преобразовать основную последовательность в целое число и возвращает полученный результат.

Если значение `base` равно нулю, то ожидаемая форма последовательности — целочисленная константа с необязательным предшествующим знаком «плюс» или «минус», не имеющая целочисленных суффиксов. Если значение `base`

задано в интервале между 2 и 36, то ожидаемая форма ведущей последовательности — последовательность символов и цифр, используемых для представления целых чисел в системе счисления, основание которой определяется параметром **base**, с необязательным предшествующим знаком «плюс» или «минус» и не имеющая целочисленных суффиксов. Символы от **a** (A) до **z** (Z) обозначают цифры от 10 до 35 в системах счисления по основанию от 10 до 36; допускаются только символы, соответствующие значениям, меньшим, чем основание системы счисления. Если значение **base** равно 16, то строка цифр может начинаться с символов **0x** (**0X**), перед которыми может находиться знак числа, если он задается.

Основная последовательность определяется как максимально длинная подстрока начальной части входной строки, начинающаяся с первого значащего символа и соответствующая ожидаемой форме. Основная последовательность не содержит символов, если строка ввода пустая или целиком состоит из символов разделителей, или если первый значащий символ не является знаком, цифрой или допустимым символом.

Если форма основной последовательности соответствует ожидаемой и значение **base** равно нулю, то последовательность символов, начинающаяся с цифры, интерпретируется как целочисленная константа. Если основная последовательность имеет ожидаемую форму и значение **base** лежит между 2 и 36, то это значение используется в качестве основания системы счисления для преобразования, в котором каждому символу приписывается значение, как это описано выше. Если основная последовательность начинается со знака «минус», то в результате преобразования будет получено отрицательное значение. В объект, на который ссылается **endptr**, помещается указатель на конец обработанной подстроки, если только **endptr** не нулевой указатель.

Если основная последовательность пуста или не имеет ожидаемой формы, преобразование не производится; в объект, на который ссылается **endptr**, помещается значение **ptr** при условии, что **endptr** не является нулевым указателем.

В случае успешного преобразования функция **strtoul** возвращает преобразованное число. Если преобразование не может быть выполнено, возвращается нуль. Если результат преобразования выходит за границы диапазона допустимых значений, возвращаются значения **LONG_MAX** или **LONG_MIN** (в зависимости от знака результата) и в **errno** помещается значение макроса **ERANGE**.

```
unsigned long int strtoul(const char *nptr, char **endptr,
                           int base);
```

Преобразует начальную часть строки, на которую ссылается **ptr**, в число типа **unsigned long int**. Функция **strtoul** работает идентично функции **strtol**. Функция **strtoul** возвращает преобразованное значение, если оно было получено. Если преобразование не может быть выполнено, возвращается нуль. Если результат преобразования выходит за границы диапазона допустимых значений, возвращается значение **ULONG_MAX** и в **errno** помещается значение макроса **ERANGE**.

```
int rand(void);
```

Функция **rand** вычисляет последовательность псевдослучайных целых чисел в диапазоне значений 0 — **RAND_MAX**. Функция **rand** возвращает псевдослучайное целое число.

void srand(unsigned int seed);

Использует значение параметра в качестве начального числа для выработки новой последовательности псевдослучайных чисел, которые будут возвращаться при последующих обращениях к функции **rand**. Если **srand** вызвать затем с тем же самым значением параметра, то последовательность псевдослучайных чисел будет повторена. Если **rand** вызывается прежде, чем будет сделан вызов **srand**, будет генерирована последовательность, соответствующая вызову **srand** со значением начального числа 1. Следующие функции определяют переносимые версии функций **rand** и **srand**:

```
static unsigned long int next = 1;
```

```
int rand(void) /* RAND_MAX предполагается 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 память для массива из **nmemb** объектов, каждый из которых имеет размер **size**. Выделяемая память инициализируется нулевыми битами. Функция **calloc** возвращает указатель на выделенную область памяти или нулевой указатель.

void free(void *ptr);

Освобождает область памяти, на которую ссылается указатель **ptr**, и делает ее доступной для последующего распределения. Если **ptr** является нулевым указателем, то никаких действий не производится. В противном случае, если аргумент не соответствует указателю, возвращенному ранее функциями **calloc**, **malloc**, или **realloc**, или если память уже была освобождена вызовами функций **free** или **realloc**, результат вызова **free** не определен.

void *malloc(size_t size);

Выделяет память для объекта размером **size** с неопределенным значением. Функция **malloc** возвращает указатель на выделенную область памяти или нулевой указатель.

void *realloc(void *ptr, size_t size);

Изменяет размер объекта, на который указывает **ptr**, до размера, определяемого параметром **size**. Содержимое общей части нового и старого объекта остается неизменным. Если новый размер объекта больше предыдущего, то значение выделенной дополнительной области памяти не определено. Если значение **ptr** равно нулевому указателю, функция **realloc** ведет себя подобно функции **malloc** при выделении памяти указанного объема. В противном случае, если аргумент не соответствует указателю, возвращенному ранее функциями **calloc**, **malloc**, или **realloc**, или если память уже была освобождена в результате вызова функций **free** или **realloc**, результат вызова **realloc** не определен. Если память не может быть выделена, объект, на который ссылается **ptr**, остается неизмененным. Если значение **size** равно нулю и **ptr** не является нулевым указателем, объект, на который указывает параметр **ptr**, освобождается. Функция **realloc** возвращает указатель на область памяти, размер которой, возможно, был изменен, или нулевой указатель.

```
void abort(void);
```

Вызывает аварийное завершение работы программы, если только сигнал SIGABRT не перехвачен и не было возврата из обработчика сигнала. В зависимости от реализации могут выполняться (или не выполняться) следующие действия: очистка буферов открытых потоков вывода, закрытие открытых потоков, удаление временных файлов. Определяемый реализацией системы код аварийного завершения программы может быть возвращен среди выполнения посредством вызова функции `raise(SIGABRT)`. Функция `abort` не может возвращаться в точку вызова.

```
int atexit(void (*func) (void));
```

Регистрирует функцию, на которую указывает `func`, которая должна вызываться без параметров при нормальном завершении программы. Реализация системы должна поддерживать регистрацию по крайней мере 32 функций. Функция `atexit` возвращает нуль в случае успешной регистрации и отличное от нуля значение в случае неудачи.

```
void exit(int status);
```

Вызывает нормальное завершение программы. Если в программе выполняется больше, чем одно обращение к функции `exit`, то результат не определен. Сначала выполняются все функции, зарегистрированные функцией `atexit`, в последовательности, обратной последовательности их регистрации. Каждая функция вызывается столько раз, сколько она была зарегистрирована. Затем, все открытые потоки с незаписанными буферизованными данными очищаются, закрываются все открытые потоки, и удаляются все файлы, созданные функцией `tmpfile`.

В заключение управление возвращается среде выполнения. Если значение `status` равно нулю или `EXIT_SUCCESS`, то среда выполнения получает определяемый реализацией код успешного завершения. Если значение `status` равно `EXIT_FAILURE`, то среда выполнения получает определяемый реализацией код аварийного завершения. В других случаях возвращаемое значение определяется реализацией. Функция `exit` не может возвращаться в точку вызова.

```
char *getenv(const char *name);
```

Ищет в списке переменных окружения строку, совпадающую со строкой, на которую указывает `name`. Набор имен переменных окружения и методы изменения списка определяются реализацией. Функция возвращает указатель на строку, связанную с найденным элементом списка. Стока, на которую ссылается указатель, не должна изменяться программой, но она может оказаться измененной при следующем обращением к функции `getenv`. Если заданное имя не найдено, возвращается нулевой указатель.

```
int system(const char *string);
```

Передает строку символов, на которую указывает `string`, *командному процессору* окружающей системы для выполнения. Способ выполнения зависит от реализации системы программирования. В качестве значения параметра может быть использован нулевой указатель, чтобы просто запросить, существует ли командный процессор. В случае, если аргумент является нулевым указателем, функция `system` возвращает отличное от нуля значение только в том случае, если командный процессор доступен. Если значение аргумента не является нулевым указателем, функция `system` возвращает определяемое реализацией значение.

```
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar) (const void *, const void *) );
```

Ищет в массиве из `nmemb` объектов, на первый элемент которого ссылается `base`, элемент, соответствующий объекту, на который ссылается параметр `key`. Размер элемента массива определяется значением параметра `size`. Функция сравнения, на которую указывает `compar`, вызывается с двумя аргументами, ссылающимися на объект `key` и на элемент массива именно в этом порядке. Функция должна возвращать целое число, меньшее нуля, или нуль, или большее нуля, если объект `key` оценивается соответственно как меньший, равный или больший, чем элемент массива. Массив должен быть упорядочен следующим образом: первыми идут элементы, меньшие, чем объект `key`, затем все элементы, которые равны ему, и затем все элементы, которые больше, чем объект `key`.

Функция `bsearch` возвращает указатель на элемент массива, соответствующий `key`, или нулевой указатель, если соответствие не обнаружено. Если два элемента удовлетворяют критерию поиска, то неизвестно, который из них будет выбран алгоритмом поиска.

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar) (const void *, const void *));
```

Сортирует массив из `nmemb` объектов. Начальный элемент указывается параметром `base`. Размер элемента определяется параметром `size`. Элементы массива сортируются по возрастанию с помощью функции сравнения, на которую указывает параметр `compar`. Эта функция вызывается с двумя аргументами, ссылающимися на сравниваемые объекты. Функция возвращает целое число меньшее чем нуль, равное нулю или большее нуля, если первый параметр соответственно меньше, равен, или больше, чем второй параметр. Если два элемента оцениваются как равные, их последовательность в сортируемом массиве не определена.

```
int abs(int j);
```

Вычисляет абсолютное значение целого числа `j`. Если результат не может быть представлен, то поведение не определено. Функция `abs` возвращает абсолютное значение.

```
div_t div(int numer, int denom);
```

Вычисляет целое частное `quot` и остаток от деления `rem` делимого `numer` на делитель `denom`. Если деление нельзя выполнить точно, в качестве значения частного выбирается ближайшее целое число, не превосходящее значения алгебраического частного. Если результат `quot` не может быть представлен, то поведение функции не определено; иначе выражение `quot * denom + rem` должно давать значение `numer`. Функция `div` возвращает структуру типа `div_t`, включающую в себя частное и остаток. Структура должна содержать следующие элементы в любом порядке:

```
int quot; /* частное */
int rem; /* остаток */
```

```
long int labs(long int j);
```

Функция, аналогичная функции `abs` за исключением того, что параметр и возвращаемое значение имеют тип `long int`.

```
ldiv_t ldiv(long int numer, long int denom);
```

Аналог функции `div` за исключением того, что параметры и все члены возвращаемой структуре 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 байтов в многобайтовом символе, на который указывает `s`. Если `s` — нулевой указатель, то функция `mblen` возвращает отличный от нуля результат или нуль в зависимости от того, является или не является кодировка многобайтового символа зависящей от состояния. Если `s` — не нулевой указатель, функция `mblen` возвращает либо 0 (если `s` указывает на нулевой символ), либо количество байтов в многобайтовом символе (если следующие `n` или меньшее число байтов образуют допустимый многобайтовый символ), либо возвращает -1 (если указанные байты не образуют допустимый многобайтовый символ).

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Если значение `s` не равно нулевому указателю, функция `mbtowc` определяет число байтов в многобайтовом символе, на который ссылается `s`. Затем она определяет код для значения типа `wchar_t`, соответствующего указанному многобайтовому символу. (Значение кода, соответствующего нулевому символу, равно нулю.) Если многобайтовый символ имеет допустимое значение и `pwc` не является нулевым указателем, функция `mbtowc` помещает полученный код в объект, на который указывает `pwc`. Исследуется не более, чем `n` байтов массива, указанного параметром `s`.

Если `s` является нулевым указателем, функция `mbtowc` возвращает результат отличный от нуля или нуль в зависимости от того, зависит или не зависит кодировка многобайтового символа от состояния. Если `s` не является нулевым указателем, то функция `mblen` возвращает либо 0 (если `s` указывает на нулевой символ), либо количество байтов в преобразуемом многобайтовом символе (если следующие `n` или меньшее количество байтов образуют допустимый многобайтовый символ), либо возвращает -1 (если указанные байты не образуют допустимый многобайтовый символ). В любом случае возвращаемое значение не может быть большим, чем `n` или значение макроса `MB_CUR_MAX`.

```
int wctomb(char *s, wchar_t wchar);
```

Функция `wctomb` определяет необходимое количество байтов (с учетом любых изменений состояния) для представления многобайтового символа, соответствующего коду, значение которого находится в `wchar`. Многобайтовый символ помещается в символьный массив, на который указывает `s` (если `s` — не нулевой указатель). В массив помещается максимум `MB_CUR_MAX` символов. Если результирующее значение `wchar` — нуль, то функция `wctomb` не изменяет начальное состояние сдвига.

Если значение `s` — нулевой указатель, функция `wctomb` возвращает результат отличный от нуля или нуль, если кодировка многобайтового символа соответственно зависит или не зависит от состояния. Если `s` — не нулевой указатель, функция `wctomb` возвращает количество байтов в многобайтовом символе, соответствующем значению `wchar`, или значение -1 (если значение `wchar` не является допустимым многобайтовым символом). В любом случае возвращаемое значение не может быть большим, чем значение макроса `MB_CUR_MAX`.

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Функция `mbstowcs` выполняет преобразование последовательности многобайтowych символов с начальным состоянием из массива, на который указывает `s`, в последовательность соответствующих кодов символов и помещает не более

n кодов в массив, на который ссылается *pwcs*. Многобайтовые символы, которые следуют за нулевым символом (преобразуемым в код со значением нуль), не рассматриваются и не преобразовываются. Преобразование многобайтовых символов выполняется так же, как и при вызове функции *mbtowc*, за исключением того, что состояние сдвига функции *mbtowc* не влияет.

Изменяются не более чем *n* элементов массива *pwcs*. Результат копирования перекрывающихся объектов не определен. Если встречается недопустимый многобайтовый символ, функция *mbstowcs* возвращает значение (*size_t*) -1. В противном случае функция *mbstowcs* возвращает число преобразованных элементов массива, не включая завершающий нулевой символ, если он имеется.

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Функция *wcstombs* выполняет преобразование последовательности кодов многобайтовых символов, на которую указывает параметр *pwcs*, в последовательность многобайтовых символов с начальным состоянием и помещает результат преобразования в массив, на который указывает *s*; преобразование прекращается, как только суммарное количество байтов многобайтовых символов превысит значение *n* или встретится нулевой символ. Преобразование кодов выполняется как при вызове функции *wctomb* за исключением того, что состояние сдвига функции *wctomb* не влияет.

Изменяются значения не более чем *n* элементов массива *s*. Результат копирования перекрывающихся объектов не определен. Если встречается код, который не соответствует допустимому многобайтому символу, функция *wcstombs* возвращает значение (*size_t*) -1. В противном случае, функция *wcstombs* возвращает число преобразованных байтов, не включая завершающий нулевой символ, если он имеется.

A.12. Обработка строк <string.h>

NULL

Константа нулевого указателя, зависящая от используемой системы.

size_t

Целый без знака тип результата операции *sizeof*.

```
void *memcp(void *s1, const void *s2, size_t n);
```

Функция копирует *n* символов из объекта, указанного параметром *s2*, в объект, на который указывает *s1*. При копировании перекрывающихся объектов поведение функции не определено. Функция *memcp* возвращает значение *s1*.

```
void *memmove(void *s1, const void *s2, size_t n);
```

Функция копирует *n* символов из объекта, указанного параметром *s2*, в объект, на который указывает *s1*. Копирование происходит так, как будто *n* символов из объекта *s2* сначала копируются во временный массив из *n* символов, не перекрывающийся с массивами, на которые ссылаются параметры *s1* и *s2*, а затем *n* символов из этого временного массива копируются в объект, на который указывает *s1*. Функция *memmove* возвращает значение *s1*.

```
char *strcpy(char *s1, const char *s2);
```

Функция *strcpy* копирует строку, указанную параметром *s2* (включая завершающий нулевой символ), в массив, на который указывает *s1*. При копировании перекрывающихся объектов поведение функции не определено. Функция *strcpy* возвращает значение *s1*.

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Функция `strncpy` копирует не более чем n символов массива, указываемого параметром `s2` (символы, следующие за нулевым символом, не копируются), в массив, на который указывает `s1`. При копировании перекрывающихся объектов поведение функции не определено. Если массив, на который ссылается `s2`, является строкой длины, меньшей чем n , то в массив `s1` дописывается нулевой символ столько раз, сколько нужно, чтобы общее количество скопированных символов равнялось n . Функция `strcpy` возвращает значение `s1`.

```
char *strcat(char *s1, const char *s2);
```

Функция `strcat` добавляет копию строки, на которую указывает `s2` (включая завершающий нулевой символ) в конец строки, на которую указывает `s1`. Начальный символ строки `s2` записывается поверх нулевого символа в конце строки `s1`. Если объекты перекрываются, поведение функции не определено. Функция `strcat` возвращает значение `s1`.

```
char *strncat(char *s1, const char *s2, size_t n);
```

Функция `strncat` добавляет не более чем n символов массива, на который указывает `s2` (символы, следующие за нулевым символом, не копируются), в конец строки, на которую указывает `s1`. Начальный символ строки `s2` записывается поверх нулевого символа в конце строки `s1`. В конец итоговой строки всегда добавляется завершающий нулевой символ. Если объекты перекрываются, поведение функции 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, на который указывает `s1`, с первыми n символами объекта, на который указывает `s2`. Функция `memcmp` возвращает целочисленное значение, большее, равное, или меньшее нуля, если объект, на который ссылается `s1`, соответственно больше, равен, или меньше объекта, на который ссылается `s2`.

```
int strcmp(const char *s1, const char *s2);
```

Функция `strcmp` сравнивает строку, на которую указывает `s1`, со строкой, на которую указывает `s2`. Функция `strcmp` возвращает целочисленное значение, большее, равное, или меньшее нуля, если строка, на которую ссылается `s1`, соответственно больше, равна, или меньше строки, на которую ссылается `s2`.

```
int strcoll(const char *s1, const char *s2);
```

Функция `strcoll` сравнивает строку, на которую указывает `s1`, со строкой, на которую указывает `s2`, в соответствии со спецификацией категории `LC_COLLATE` текущей локализации. Функция `strcoll` возвращает целочисленное значение, большее, равное, или меньшее нуля, если строка, на которую ссылается `s1`, соответственно больше, равна, или меньше строки, на которую ссылается `s2`; сравнение выполняется в соответствии с требованиями текущей локализации.

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Функция `strncmp` сравнивает не более n первых символов массива, на который указывает `s1` (символы, следующие за нулевым символом, в сравнении не участвуют), с символами массива, на который указывает `s2`. Функция `strncmp` возвращает целочисленное значение, большее, равное, или меньшее нуля, если соответственно массив, возможно заканчивающийся нулевым символом, на который ссылается `s1`, больше, равен, или меньше массива, возможно заканчивающегося нулевым символом, на который ссылается `s2`.

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Функция `strxfrm` выполняет преобразование строки, на которую указывает `s2`, и помещает строку, полученную в результате преобразования, в массив, на который ссылается `s1`. Преобразование выполняется таким образом, что результат сравнения двух преобразованных строк при помощи функции `strcmp` будет таким же, как результат сравнения двух исходных строк при помощи функции `strcoll`. В массив, на который ссылается `s1`, помещается не более чем `n` символов результирующей строки, включая завершающий нулевой символ. Если значение `n` равно нулю, то `s1` может быть нулевым указателем. Если копируются перекрывающиеся объекты, поведение функции не определено. Функция `strxfrm` возвращает длину преобразованной строки (не считая завершающего нулевого символа). Если возвращаемое значение равно или больше чем `n`, то содержимое массива, адресуемого `s1`, не определено.

```
void *memchr(const void *s, int c, size_t n);
```

Функция `memchr` ищет первое включение символа `c` (преобразованного к типу `unsigned char`) среди первых `n` символов (интерпретируемых как `unsigned char`) объекта, на который указывает `s`. Функция `memchr` возвращает указатель на обнаруженный символ или нулевой указатель, если символ не найден.

```
char *strchr(const char *s, int c);
```

Функция `strchr` ищет первое включение символа `c` (преобразованного к типу `char`) в строке, на которую указывает `s`. Завершающий нулевой символ рассматривается как часть строки. Функция `strchr` возвращает указатель на обнаруженный символ или нулевой указатель, если символ в строке не найден.

```
size_t strcspn(const char *s1, const char *s2);
```

Функция `strcspn` вычисляет максимальную длину такой подстроки в строке, на которую указывает `s1`, среди символов которой нет ни одного символа из строки, на которую указывает `s2`. Функция `strcspn` возвращает длину найденной подстроки.

```
char *strpbrk(const char *s1, const char *s2);
```

Функция `strpbrk` находит первое вхождение любого символа из строки, на которую указывает `s2`, в строку, на которую указывает `s1`. Функция `strpbrk` возвращает указатель на найденный символ или нулевой указатель, если ни один символ из `s2` не обнаружен в `s1`.

```
char *strrchr(const char *s, int c);
```

Функция `strrchr` ищет последнее включение символа `c` (преобразованного к типу `char`) в строке, на которую указывает `s`. Завершающий нулевой символ рассматривается как часть строки. Функция `strrchr` возвращает указатель на обнаруженный символ или нулевой указатель, если символ в строке не найден.

```
size_t strspn(const char *s1, const char *s2);
```

Функция `strspn` вычисляет максимальную длину такой подстроки, найденной в строке, указанной `s1`, все символы которой являются символами строки, на которую указывает `s2`. Функция `strspn` возвращает длину найденной подстроки.

```
char *strstr(const char *s1, const char *s2);
```

Функция `strstr` находит в строке, на которую ссылается `s1`, первое вхождение строки, на которую указывает `s2` (за исключением завершающего нулевого символа). Функция `strstr` возвращает указатель на обнаруженную строку или нулевой указатель, если строка не найдена. Если `s2` указывает на строку нулевой длины, функция возвращает значение `s1`.

```
char *strtok(char *s1, const char *s2);
```

Последовательность вызовов функции `strtok` разбивает строку, адресуемую `s1`, на последовательность лексем, каждая из которых завершается символом из строки, на которую указывает `s2`. При первом вызове функция `strtok` имеет в качестве первого аргумента значение `s1`, при всех последующих вызовах в качестве первого параметра передается нулевой указатель. Стока символов-разделителей, на которую ссылается `s2`, может меняться от вызова к вызову.

При первом вызове в строке, на которую указывает `s1`, производится поиск первого символа, который не содержится в текущей строке разделителей, на которую ссылается `s2`. Если такой символ не найден, то строка, на которую ссылается `s1`, не будет разбиваться на лексемы и функция `strtok` возвратит нулевой указатель. Если такой символ найден, то он будет началом первой лексемы.

Затем функция `strtok`, начиная с найденного символа, ищет в строке любой символ, который содержится в текущей строке разделителей. Если ни одного такого символа не будет найдено, то текущая лексема расширяется до конца строки, адресуемой `s1`, и все последующие поиски лексем возвратят нулевые указатели. Если такой символ будет найден, то на его место будет записан нулевой символ, завершающий текущую лексему. Функция `strtok` сохраняет значение указателя, ссылающегося на следующий символ, начиная с которого должен осуществляться поиск следующей лексемы.

Каждый последующий вызов функции с нулевым указателем в качестве значения первого аргумента продолжает поиск лексем начиная с того места в строке, на которое указывает возвращенное в результате последнего вызова значение указателя; поиск выполняется как было описано выше. Библиотечные функции не должны вызывать функцию `strtok`. Функция `strtok` возвращает указатель на первый символ лексемы или нулевой указатель, если лексемы не обнаружены.

```
void *memset(void *s, int c, size_t n);
```

Функция `memset` копирует значение символа `c` (преобразованного к типу `unsigned char`) в каждый из первых `n` символов объекта, на который указывает параметр `s`. Функция `memset` возвращает значение `s`.

```
char *strerror(int errnum);
```

Функция `strerror` возвращает адрес строки сообщения об ошибке, соответствующей коду ошибки, помещенному в `errnum`. Библиотечные функции не должны вызывать функцию `strerror`. Функция `strerror` возвращает указатель на строку, содержимое которой зависит от реализации системы. Программа не должна изменять содержимое этого массива, но оно может оказаться измененным при следующем вызове функции `strerror`.

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

Функция `strlen` вычисляет длину строки, на которую указывает параметр `s`. Функция `strlen` возвращает число символов, предшествующих завершающему нулевому символу.

A.13. Функции даты и времени <time.h>

CLOCKS_PER_SEC

Число интервалов в секунду; значение, возвращаемое функцией `clock`.

NULL

Константа нулевого указателя, зависящая от используемой системы.

clock_t

Арифметический тип, используемый для представления времени.

time_t

Арифметический тип, используемый для представления времени.

size_t

Целый без знака тип результата операции `sizeof`.

struct tm

Структура, используемая для хранения календарного времени, *расчлененного на компоненты*. Структура должна содержать в любом порядке по крайней мере указанные ниже элементы. Назначение этих элементов и диапазон их нормальных значений приводятся в комментариях.

```
int tm_sec;      /* секунды после минуты - [0, 61] */
int tm_min;      /* минуты после часа - [0, 59] */
int tm_hour;     /* часы, начиная с полуночи - [0, 23] */
int tm_mday;     /* день с начала месяца - [1, 31] */
int tm_mon;      /* месяцы, начиная с января - [0, 11] */
int tm_year;     /* годы, начиная с 1900 */
int tm_wday;     /* день недели, начиная с воскресенья - [0, 6] */
int tm_yday;     /* дни, начиная с 1 января - [0, 365] */
int tm_isdst;    /* флаг летнего времени */
```

Значение флага `tm_isdst` равно положительному значению, если установлено летнее время, нулю, если летнее время не установлено, и отрицательному значению, если информации по этому вопросу нет.

clock_t clock(void);

Функция `clock` используется для определения времени, использованного процессором. Функция `clock` возвращает наилучшее обеспечиваемое реализацией приближение процессорного времени, потраченного программой с начала ее выполнения. Чтобы определить время в секундах, нужно значение, возвращенное функцией `clock`, разделить на значение макроса `CLOCKS_PER_SEC`. Если значение использованного процессорного времени недоступно или не может быть представлено, функция возвращает значение (`clock_t`) -1.

double difftime(time_t time1, time_t time0);

Функция `difftime` вычисляет разность между двумя значениями календарного времени: `time1` и `time0`. Функция `difftime` возвращает время, выраженное в секундах, как значение типа `double`.

time_t mktime(struct tm *timeptr);

Функция `mktime` преобразует расчлененное на компоненты местное время, записанное в структуре, на которую указывает `timeptr`, в значение календарного времени с использованием того механизма представления, который использует функция `time`. Значения элементов `tm_wday` и `tm_yday` игнори-

руются, а значения других элементов структуры не ограничиваются указанными выше диапазонами. В случае успешного завершения элементы структуры `tm_wday` и `tm_yday` получают соответствующие значения, значения других элементов также изменяются в соответствии с диапазонами, обозначенными выше. Конечное значение `tm_mday` устанавливается только тогда, когда определены значения `tm_mon` и `tm_year`. Функция `mktime` возвращает определенное календарное время в виде значения типа `time_t`. Если значение календарного времени не может быть представлено, функция возвращает значение `(time_t) -1`.

```
time_t time(time_t *timer);
```

Функция `time` определяет текущее календарное время. Функция `time` возвращает наилучшее приближение текущего календарного времени, обеспечиваемое реализацией. Если календарное время не доступно, то возвращается значение `(time_t) -1`. Если `timer` не является нулевым указателем, то возвращаемое значение помещается в объект, на который ссылается указатель `timer`.

```
char *asctime(const struct tm *timeptr);
```

Функция `asctime` преобразует расчлененное на компоненты время из структуры, на которую ссылается `timeptr`, в строку формата

```
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, на которое ссылается `timer`, к значению местного времени, выраженного в форме строки. Вызов этой функции эквивалентен вызову

```
asctime (localtime (timer))
```

Функция `ctime` возвращает указатель, возвращаемый функцией `asctime` с тем же значением расчлененного на компоненты времени в качестве аргумента.

```
struct tm *gmtime(const time_t *timer);
```

Функция `gmtime` преобразует календарное время, на которое ссылается `timer`, в расчлененное на компоненты время, представленное в формате универсального скоординированного времени (UTC). 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, на которое ссылается `timer`, в расчлененное на компоненты время, представленное в формате местного времени. Функция `localtime` возвращает указатель на этот объект.

```
size_t strftime(char *s, size_t maxsize,
                const char *format, const struct tm *timeptr);
```

Функция `strftime` помещает символы в массив, на который указывает `s`, в соответствии со строкой форматирования, на которую ссылается `format`. Стока формата состоит из возможно нулевого числа спецификаций формата и многобайтовых символов. Все обычные символы (включая нулевой символ) копируются в массив в неизмененном виде. Если происходит копирование перекрывающихся объектов, результат не определен. В массив помещается не более, чем `maxsize` символов. Каждый спецификатор формата заменяется соответствующими символами в соответствии с приведенным ниже описанием.

ем. Подставляемые символы определяются категорией LC_TIME текущей локализации и значениями, содержащимися в структуре, адресуемой `timeptr`.

- %**a** заменяется сокращенным именем дня недели в текущей локализации.
- %**A** заменяется полным именем дня недели текущей локализации.
- %**b** заменяется сокращенным именем месяца текущей локализации.
- %**B** заменяется полным именем месяца текущей локализации.
- %**c** заменяется соответствующим представлением даты и времени текущей локализации.
- %**d** заменяется десятичным значением дня месяца (01-31).
- %**H** заменяется десятичным значением часа (24-х часовое исчисление): (00-23).
- %**I** заменяется десятичным значением часа (12-ти часовое исчисление): (01-12).
- %**j** заменяется десятичным значением дня года (001-366).
- %**m** заменяется десятичным значением месяца (01-12).
- %**M** заменяется десятичным значением минуты (00-59).
- %**p** заменяется принтым в данной текущей локализации эквивалентом обозначения AM/PM, используемого при 12-ти часовой записи времени.
- %**S** заменяется десятичным значением секунды (00-61).
- %**U** заменяется десятичным значением номера недели с начала года (первое воскресенье считается первым днем недели номер 1): (00-53).
- %**w** заменяется десятичным значением дня недели (0-6), где воскресенье имеет значение 0.
- %**W** заменяется десятичным значением номера недели с начала года (первый понедельник считается первым днем недели номер 1): (00-53).
- %**x** заменяется соответствующим представлением даты текущей локализации.
- %**X** заменяется соответствующим представлением времени текущей локализации.
- %**y** заменяется десятичным значением года без указания столетия (00-99).
- %**Y** заменяется десятичным значением года с указанием столетия.
- %**Z** заменяется именем или сокращением часового пояса или ничем, если часовой пояс не определен.
- %**%** заменяется на %.

Попытка использования спецификатора формата, который не был определен в вышеупомянутом списке, приведет к неопределенному результату. Если общее количество символов после форматирования, включая завершающий нулевой символ, не превышает значения `maxsize`, функция `strftime` возвращает число символов, помещенное в массив, адресуемый `s`, не считая завершающего нулевого символа. В противном случае возвращается нулевое значение, а содержимое массива не определено.

A.14. Ограничения реализации системы программирования

`<limits.h>`

Следующие макросы должны иметь определенные здесь значения или превосходить их по абсолютной величине.

`#define CHAR_BIT`

8

Число битов для самого маленького объекта, который не является битовым полем (подразумевается 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 `SCHAR_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 количество байтов в многобайтовом символе, поддерживаемое всеми локализациями.

`#define SHRT_MIN`

-32767

Минимальное значение объекта типа `short int`.

`#define SHRT_MAX`

+32767

Максимальное значение объекта типа `short int`.

`#define USHRT_MAX`

65535

Максимальное значение объекта типа `unsigned short int`.

`#define INT_MIN`

-32767

Минимальное значение объекта типа `int`.

`#define INT_MAX`

+32767

Максимальное значение объекта типа `int`.

`#define UINT_MAX`

65535

Максимальное значение объекта типа `unsigned int`.

`#define LONG_MIN`

-2147483647

Минимальное значение объекта типа `long int`.

`#define LONG_MAX`

+2147483647

Максимальное значение объекта типа `long int`.

#define ULONG_MAX 4294967295

Максимальное значение объекта типа `unsigned long int`.

`<float.h>`

#define FLT_ROUNDS

Режим округления при сложении чисел с плавающей запятой.

- 1 режим не определен
- 0 в направлении к нулю
- 1 до ближайшего
- 2 в направлении к положительной бесконечности
- 3 в направлении к отрицательной бесконечности

Следующие макросы должны иметь определенные здесь значения или превосходить их по абсолютной величине.

#define FLT_RADIX

2

Основание степени экспоненциального представления числа, b .

#define FLT_MANT_DIG

#define LDBL_MANT_DIG

#define DBL_MANT_DIG

Число цифр системы счисления по основанию `FLT_RADIX`, используемое для представления значащей части чисел с плавающей запятой, p .

#define FLT_DIG

6

#define LDBL_DIG

10

#define DBL_DIG

10

Число десятичных цифр q , достаточное для прямого и обратного преобразования любого числа с плавающей запятой с q десятичными значащими цифрами в число с плавающей запятой, записанное в системе счисления с основанием b с помощью p цифр; при преобразованиях не должны изменяться все q цифр исходного числа.

#define FLT_MIN_DIG

#define LDBL_MIN_DIG

#define DBL_MIN_DIG

Минимальное отрицательное целое число, такое, что `FLT_RADIX`, введенное в степень на единицу меньшую, чем значение этого целого числа, дает нормализованное число с плавающей запятой.

#define FLT_MIN_10_DIG

-37

#define LDBL_MIN_10_DIG

-37

#define DBL_MIN_10_DIG

-37

Минимальное отрицательное целое число такое, что число 10, введенное в степень этого числа, равно числу, которое еще находится в диапазоне нормализованных чисел с плавающей запятой.

```
#define FLT_MAX_EXP  
#define LDBL_MAX_EXP  
#define DBL_MAX_EXP
```

Максимальное целое число такое, что при возведении **FLT_RADIX** в степень, на единицу меньшую значения этого числа, получается результат, еще представимый в виде конечного числа с плавающей запятой.

```
#define FLT_MAX_10_EXP +37  
#define LDBL_MAX_10_EXP +37  
#define DBL_MAX_10_EXP +37
```

Максимальное целое число, такое, что число 10, введенное в степень этого числа, еще находится в диапазоне допустимых значений чисел с плавающей запятой.

Следующие макросы должны иметь определенные здесь значения или превосходить их.

```
#define FLT_MAX 1E+37  
#define LDBL_MAX 1E+37  
#define DBL_MAX 1E+37
```

Максимальное число, которое может представлено с плавающей запятой.

Следующие макросы должны иметь определенные здесь или меньшие значения.

```
#define FLT_EPSILON 1E-5  
#define LDBL_EPSILON 1E-9  
#define DBL_EPSILON 1E-9
```

Разность между числом 1.0 и наименьшим числом, значение которого больше, чем 1.0, представимая как число с плавающей запятой.

```
#define FLT_MIN 1E-37  
#define LDBL_MIN 1E-37  
#define DBL_MIN 1E-37
```

Минимальное нормализованное число с плавающей запятой.

Б

Таблица приоритета операций

Операции	Ассоциативность
:: (унарная; справа налево) :: (бинарная; слева направо)	
() [] -> .	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.

п р и л о ж е н и е

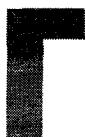
B

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.

п р и л о ж е н и е



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.

План

- Г.1. Введение
- Г.2. Сокращенная запись двоичных чисел в восьмеричной и шестнадцатеричной системах счисления
- Г.3. Преобразование восьмеричных и шестнадцатеричных чисел в двоичные
- Г.4. Преобразование двоичных, восьмеричных и шестнадцатеричных чисел в десятичные
- Г.5. Преобразование десятичных чисел в двоичные, восьмеричные и шестнадцатеричные
- Г.6. Представление отрицательных двоичных чисел: дополнение до двух

Резюме • Терминология • Упражнения для самопроверки • Ответы на упражнения для самопроверки • Упражнения

Г.1 Введение

В этом приложении мы поговорим об основных системах представления чисел, используемых программистами на C++, особенно теми из них, кто работает над проектами программного обеспечения, активно взаимодействующего с аппаратными средствами на «машинном уровне». Такие проекты включают в себя: операционные системы, программное обеспечение работы в сети, компиляторы, системы управления базами данных и прикладные программы, от которых требуется высокая эффективность работы.

Когда в программе на C++ мы используем целое число типа 227 или -63, то предполагаем, что оно записано в *десятичной системе счисления* (с основанием 10). Цифрами в десятичной системе счисления являются 0, 1, 2, 3, 4, 5, 6, 7, 8, и 9. Самая младшая цифра — 0, а самая старшая цифра — 9; она на единицу меньше, чем 10, являющееся *основанием* системы счисления. Для внутреннего представления данных в компьютере используется *двоичная система счисления* (с основанием 2). Двоичная система счисления обходится только двумя цифрами, а именно, 0 и 1. Самая младшая цифра — 0, а самая старшая цифра — 1; она на единицу меньше основания системы счисления 2.

Как мы увидим далее, двоичные числа намного длиннее их десятичных эквивалентов. Программистам, пишущим на ассемблере или на языках высокого уровня, которые, подобно C++, дают возможность работать и на «машинном уровне», неудобно использовать двоичные числа. Они предпочитают две другие системы счисления: *восьмеричную систему счисления* (с основанием 8) и *шестнадцатеричную систему счисления* (с основанием 16), завоевавшие себе популярность прежде всего возможностью сокращенной записи двоичного числа.

В восьмеричной системе счисления используются цифры от 0 до 7. Поскольку и в двоичной системе счисления, и в восьмеричной системе применяется меньшее количество цифр, чем в десятичной системе счисления, то в качестве цифр в этих системах просто используются соответствующие цифры десятичной системы.

В шестнадцатеричной системе счисления возникает проблема, связанная с тем, что для записи чисел требуется уже шестнадцать цифр: самая младшая цифра — 0, а самая старшая цифра должна иметь десятичное значение 15 (на единицу меньшее, чем основание системы 16). В соответствии с принятым соглашением, для представления шестнадцатеричных цифр, соответствующих десятичным значениям от 10 до 15, используются символы от A до F. Таким образом в шестнадцатеричной записи мы можем встретить числа типа 876, состоящие исключительно из цифр десятичной системы счисления, и числа типа 8A55F, состоящие из цифр и букв, и числа вида FFE, состоящих исключительно из букв. Иногда, попадаются шестнадцатеричные числа типа FACE или FEED, имеющие второй смысл (в переводе ЛИЦО и ПИЩА), что кажется странным для некоторых программистов, привыкших работать с десятичными числами.

Каждая из перечисленных систем счисления использует *позиционную запись*, когда каждая позиция, в которой записывается цифра, имеет свое *позиционное значение*. Например, про десятичное число 937 мы говорим, что цифра 7 записана в позиции *единиц*, цифра 3 записана в позиции *десятков* и 9 — в позиции *сотен* (цифры 9, 3 и 7, при этом, называются *числовыми значениями*). Обратите внимание, что значения этих позиций равны основанию системы счисления (в данном случае — 10), возведенному в степень, начинающуюся с нулевой и увеличивающуюся на 1 при перемещении в числе на одну позицию влево (рис. Г.3).

Для более длинных десятичных чисел, чем то что мы взяли в качестве примера, следующими позициями слева будут позиции: *тысяч* (10 в степени 3), *десятков тысяч* (10 в степени 4), *сотен тысяч* (10 в степени 5), *миллионов* (10 в степени 6), *десятков миллионов* (10 в степени 7) и так далее.

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	A (десятичное значение 10)
		9	B (десятичное значение 11)
			C (десятичное значение 12)
			D (десятичное значение 13)
			E (десятичное значение 14)
			F (десятичное значение 15)

Рис. Г.1. Цифры двоичной, восьмеричной, десятичной и шестнадцатеричной систем счисления

Атрибут	Двоичная система	Восьмеричная система	Десятичная система	Шестнадцатеричная система
Основание	2	8	10	16
Младшая цифра	0	0	0	0
Старшая цифра	1	7	9	F

Рис. Г.2. Сравнение двоичной, восьмеричной, десятичной и шестнадцатеричной систем счисления

Значения позиций в десятичной системе счисления			
Десятичная цифра	9	3	7
Название позиции	Сотни	Десятки	Единицы
Позиционное значение	100	10	1
Значение позиции как степень основания (10)	10^2	101	10^0

Рис. Г.3. Значения позиций в десятичной системе счисления

Про двоичное число 101 мы говорим, что его самая правая единица записана в позиции *единиц*, 0 записан в позиции *двоек*, а самая левая единица находится в позиции *четверок*. Обратите внимание, что каждая из этих позиций имеет значение, равное степени основания системы счисления (в данном случае 2), и что значение степени начинается со значения 0 и увеличивается на 1 при перемещении в числе на одну позицию влево (рис. Г.4).

Значения позиций в двоичной системе счисления			
Двоичная цифра	1	0	1
Название позиции	Четверки	Двойки	Единицы
Позиционное значение	4	2	1
Значение позиции как степень основания (2)	2^2	2^1	2^0

Рис. Г.4. Значения позиций в двоичной системе счисления

Для более длинных двоичных чисел следующими позициями слева будут *позиция восьми* (2 в степени 3), *позиция шестнадцати* (2 в степени 4), *позиция тридцати двух* (2 в степени 5), *позиция шестидесяти четырех* (2 в степени 6) и так далее.

В восьмеричном числе 425 цифра 5 записана в *позиции единиц*, цифра 2 записана в *позиции восьми* и 4 — в *позиции шестидесяти четырех*. Заметьте, что значение каждой из этих позиций равно степени основания системы счисления (в этом случае равной 8) и что показатель степени изменяется от 0, увеличиваясь на 1 при передвижении в числе на одну позицию влево (рис. Г.5).

Для больших по величине восьмеричных чисел, следующие позиции слева будут *позиции пятисот двенадцати* (8 в степени 3), *четырех тысяч девяноста шести* (8 в степени 4), *тридцати двух тысяч семисот шестидесяти восьми* (8 в степени 5) и так далее.

В шестнадцатеричном числе 3DA цифра A записана в *позиции единиц*, цифра D записана в *позиции шестнадцати* и 3 — в *позиции двухсот пятидесяти шести*. Заметим, что значение каждой из этих позиций равно основанию системы счисления (в данном случае 16), возведенному в степень, показатель которой при продвижении в числе на одну позицию влево увеличивается на 1, а начинается со значения 0 (рис. Г.6).

Значения позиций в восьмеричной системе счисления			
Двоичная цифра	4	2	5
Название позиции	Шестьдесят четыре	Восьмерки	Единицы
Позиционное значение	64	8	1
Значение позиции как степень основания (8)	8^2	8^1	80

Рис. Г.5. Значения позиций в восьмеричной системе счисления

Значения позиций в шестнадцатеричной системе счисления			
Двоичная цифра	3	D	A
Название позиции	Двести пятьдесят шесть	Шестнадцать	Единицы
Позиционное значение	256	16	1
Значение позиции как степень основания (16)	16^2	16^1	16^0

Рис. Г.6. Значения позиций в шестнадцатеричной системе счисления

Для шестнадцатеричных чисел большей длины следующими левыми позициями будут позиции четырех тысяч девяноста шести (16 в степени 3), тридцати двух тысяч семисот шестидесяти восьми (16 в степени 4) и так далее.

Г.2. Сокращенная запись двоичных чисел в восьмеричной и шестнадцатеричной системах счисления

В основном восьмеричная и шестнадцатеричная системы счисления используются для сокращенного представления длинных двоичных чисел. Рис. Г.7 подтверждает нам тот факт, что длинные двоичные числа могут быть выражены более кратко в системах счисления с более высокими основаниями.

Между восьмеричной и шестнадцатеричной системами счисления с одной стороны и двоичной системой с другой стороны имеется важное соотношение, заключающееся в том, что основания восьмеричной и шестнадцатеричной систем (8 и 16 соответственно) являются степенями основания двоичной системы счисления. Рассмотрим приведенное ниже двоичное число, состоящее из 12 двоичных цифр, и его эквиваленты в восьмеричной и шестнадцатеричной системах счисления. Возможно, вы уже догадались, как на основании указанной выше связи между системами счисления можно легко и удобно перевести двоичное число в восьмеричное или шестнадцатеричное представление. Ответ приводится ниже.

Двоичное
число
100011010001

Восьмеричный
эквивалент числа
4321

Шестнадцатеричный
эквивалент числа
8D1

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	10000	20	10

Рис. Г.7. Пример записи чисел в десятичной, двоичной, восьмеричной и шестнадцатеричной системах счисления

Преобразовать двоичное число в восьмеричное очень просто; разбейте взятое нами для примера двоичное число из 12 цифр на группы из трех последовательных разрядов каждая и запишите под этими группами соответствующие им восьмеричные цифры следующим образом:

100	011	010	001
4	3	2	1

Обратите внимание, что восьмеричная цифра, расположенная под каждой группой из трех разрядов, равна значению этого двоичного числа из 3 цифр, в соответствии с рис. Г.7.

Тот же самый способ применяется при преобразовании двоичных чисел в шестнадцатеричные. В нашем примере, разбейте двоичное число из 12 цифр в группы по четыре последовательных разрядов каждая и запишите под этими группами соответствующие им шестнадцатеричные цифры следующим образом:

1000	1101	0001
8	D	1

Заметьте, что шестнадцатеричная цифра, которая записана под каждой группой из четырех разрядов, соответствует шестнадцатеричному эквиваленту этого 4-х разрядного двоичного числа (см. рис. Г.7).

Г.3. Преобразование восьмеричных и шестнадцатеричных чисел в двоичные

В предыдущем разделе мы узнали, что для преобразования двоичного числа в восьмеричное или шестнадцатеричное представление достаточно разбить его на группы двоичных цифр и записать вместо этих групп соответствующие им по значению восьмеричные или шестнадцатеричные цифры. Этот процесс может использоваться в обратной последовательности для преобразования восьмеричного или шестнадцатеричного числа в двоичное.

Например, восьмеричное число 653 преобразовывается в двоичное число следующим образом: цифра 6 заменяется ее двоичным эквивалентом из 3 цифр 110, цифра 5 заменяется ее двоичным эквивалентом из 3 цифр 101 и цифра 3 заменяется ее двоичным эквивалентом 011; в итоге получается двоичное число 110101011 из 9 цифр.

Шестнадцатеричное число FAD5 преобразовывается в двоичное заменой цифры F на ее двоичный эквивалент из 4 цифр 1111, цифры A — на ее двоичный эквивалент 1010, цифры D — на ее двоичный эквивалент 1101 из 4 цифр и 5 — на ее двоичный эквивалент из 4 цифр 0101. В итоге получается двоичное число из 16 цифр 1111101011010101.

Г.4. Преобразование двоичных, восьмеричных и шестнадцатеричных чисел в десятичные

Поскольку мы приучены к работе с десятичными числами, часто бывает нужно преобразовать двоичное, восьмеричное или шестнадцатеричное число в десятичное число, чтобы узнать его более привычное значение. В таблицах раздела Г.1 приводятся значения позиций различных систем счисления, выраженные в десятичном виде. Для того, чтобы преобразовать к десятичному виду число, записанное в другой системы счисления, умножьте десятичный эквивалент каждой цифры числа на соответствующее позиционное значение и просуммируйте полученные величины. Например, двоичное число 110101 преобразовывается в десятичное число 53, как показано на рис. Г.8.

Преобразование двоичного числа в десятичное						
Позиционные значения	32	16	8	4	2	1
Числовые значения	1	1	0	1	0	1
Произведения	$1 \cdot 32 = 32$	$1 \cdot 16 = 16$	$0 \cdot 8 = 0$	$1 \cdot 4 = 4$	$0 \cdot 2 = 0$	$1 \cdot 1 = 1$
Сумма	$32 + 16 + 0 + 4 + 0 + 1 = 53$					

Рис. Г.8. Преобразование двоичного числа в десятичное

Для преобразования восьмеричного числа 7614 в десятичное число 3980 мы используем ту же самую методику, применяя соответствующие значения позиций восьмеричной системы счисления, как показано на рис. Г.9.

Преобразование восьмеричного числа в десятичное				
Позиционные значения	512	64	8	1
Числовые значения	7	6	1	4
Произведения	$7 \cdot 512 = 3584$	$6 \cdot 64 = 384$	$1 \cdot 8 = 8$	$4 \cdot 1 = 4$
Сумма	$3584 + 384 + 8 + 4 = 3980$			

Рис. Г.9. Преобразование восьмеричного числа в десятичное

Преобразование шестнадцатеричного числа AD3B в десятичное 44347 выполняется тем же самым способом с использованием соответствующих позиционных значений шестнадцатеричной системы счисления, как показано на рис. Г.10.

Преобразование шестнадцатеричного числа в десятичное				
Позиционные значения	4096	256	16	1
Числовые значения	A	D	3	B
Произведения	$A \cdot 4096 = 40960$	$D \cdot 256 = 3328$	$3 \cdot 16 = 48$	$B \cdot 1 = 11$
Сумма	$40960 + 3328 + 48 + 11 = 44347$			

Рис. Г.10. Преобразование шестнадцатеричного числа в десятичное

Г.5. Преобразование десятичных чисел в двоичные, восьмеричные и шестнадцатеричные

Правила преобразования, изложенные в предыдущем разделе, естественным образом вытекают из правил записи числа в позиционной системе счисления. Преобразование десятичных чисел к двоичному, восьмеричному или шестнадцатеричному представлению также выполняется в соответствии с этими соглашениями.

Предположим, что мы хотим преобразовать десятичное число 57 в двоичное. Начнем с того, что выпишем значения позиций двоичной системы в колонки, справа налево, пока не дойдем до позиционного значения, превосходящего наше десятичное число. Эта позиция нам не нужна и мы отбрасываем этот столбец. Таким образом, сначала мы запишем:

Позиционные значения: 64 32 16 8 4 2 1

Затем мы отбрасываем столбец с позиционным значением 64:

Позиционные значения: 32 16 8 4 2 1

После этого начинается процесс обработки столбцов, с крайнего левого столбца и далее, до крайнего правого. Мы делим 57 на 32, получаем 1 и остаток 25; записываем 1 в столбце 32. Далее, мы делим остаток 25 на 16, получаем 1 и 9 в остатке; записываем 1 в столбце 16. Затем, делим 9 на 8, получаем 1 и 1 в остатке. Для следующих двух столбцов частное от деления остатка равного 1 на их позиционные значения равны 0, так что в эти столбцы мы записываем нули. Для крайней правой позиции число 1, деленное на 1, дает 1 и мы записываем 1 в столбец единиц. В результате получаем:

Позиционные значения: 32 16 8 4 2 1

Значения цифр 1 1 1 0 0 1

и, таким образом, десятичному числу 57 будет соответствовать двоичное число 111001.

Для преобразования десятичного числа 103 в восьмеричное начнем с того, что последовательно выпишем позиционные значения восьмеричной системы в столбцы, пока не дойдем до значения позиции, превосходящей наше десятичное число. Этот столбец нам не нужен и мы его отбросим. Таким образом, сначала мы запишем:

Позиционные значения: 512 64 8 1

Затем мы отбрасываем столбец с позиционным значением 512 и получаем:

Позиционные значения: 64 8 1

После этого мы начинаем обработку с крайнего левого столбца и далее до крайнего правого. Мы делим 103 на 64, получаем 1, остаток 39 и записываем 1 в столбце 64.

Далее, мы делим 39 на 8, получаем 4 и 7 в остатке и записываем 4 в столбце восьмерок. И, наконец, мы делим 7 на 1, получаем 7 и 0 в остатке и записываем 7 в столбце единиц. В результате получаем:

Позиционные значения:	64 8 1
Значения цифр:	1 4 7

Таким образом, десятичному числу 103 соответствует восьмеричное число 147.

Для преобразования десятичного числа 375 в шестнадцатеричное начнем с того, что выпишем значения позиций шестнадцатеричной системы в столбцы, пока не дойдем до позиционного значения, превосходящего наше десятичное число. Этот столбец нам не нужен и мы его отбросим. Таким образом, сначала мы запишем:

Позиционные значения: 4096 256 16 1

Затем мы отбрасываем столбец с позиционным значением 4096 и получаем:

Позиционные значения: 256 16 1

После этого мы начинаем обработку столбцов с крайнего левого столбца и далее до крайнего правого. Мы делим 375 на 256, получаем 1, остаток 119 и записываем 1 в столбце 256. Далее, мы делим 119 на 16, получаем 7 и 7 в остатке и записываем 7 в столбце 16. И, наконец, мы делим 7 на 1, получаем 7 и 0 в остатке и записываем 7 в столбце единиц. В результате получаем:

Позиционные значения:	256 16 1
Значения цифр:	1 7 7

Таким образом, десятичному числу 375 соответствует шестнадцатеричное число 177.

Г.6. Представление отрицательных двоичных чисел: дополнение до двух

Все предыдущие разделы этого приложения касались только положительных чисел. В этом разделе мы расскажем, как в памяти компьютера представляются отрицательные числа при помощи метода *дополнения до двух*. Сначала мы опишем сам метод получения дополнения заданного двоичного числа до двух, а затем покажем, что это дополнение действительно представляет значение данного двоичного числа со знаком минус.

Рассмотрим вычислительную машину, в которой для представления целых чисел используется 32 бита. Определим значение

```
int value = 13;
```

Представление `value` в виде 32-битового целого будет иметь следующий вид:

```
00000000 00000000 00000000 00001101
```

Чтобы сформировать отрицательное значение `value`, сначала выполним *дополнение до единицы (поразрядное дополнение)*, при помощи поразрядной операции дополнения (`-`) языка C++ :

```
ones_complement_of_value = ~value;
```

Внутреннее представление `-value` получается из представления `value` побитовым инвертированием: единицы заменяются нулями, а нули становятся единицами, как это показано ниже:

```
value:
00000000 00000000 00000000 00001101
```

`~value` (т. е. побитовое дополнение числа `value`):
`11111111 11111111 11111111 11110010`

Чтобы получить *дополнение до двух* (*точное дополнение*) для значения `value`, нужно просто добавить единицу к дополнению этого числа до единицы. Таким образом, дополнение до двух значения `value`:

`11111111 11111111 11111111 11110010`

Теперь, если полученное число действительно равно -13, то мы можем прибавить его к двоичному значению числа 13 и должны в результате получить 0.

Давайте попробуем:

<code>00000000 00000000 00000000 00001101</code>
<code>+11111111 11111111 11111111 11110011</code>
<hr/>
<code>00000000 00000000 00000000 00000000</code>

Разряд переноса, в который выталкивается единица из крайнего левого столбца, отбрасывается и мы действительно получаем нуль. Если мы сложим число и дополнение этого числа до единицы, то в результате получим значение, все разряды которого равны единице. При использовании дополнения до двух все разряды получаются нулевыми только потому, что дополнение до двух на единицу больше, чем дополнение до единицы. При добавлении 1 в любой разряд, содержащий 1, получается 0 для данного разряда и единица переносится в следующий разряд слева. Процесс переноса продолжается, пока единица не будет вытолкнута из крайнего левого разряда и отброшена; в результате все разряды итогового числа будут нулевыми и мы получим в ответе нуль.

При выполнении операции вычитания типа

`x = a - value;`

компьютер, на самом деле, выполняет операцию сложения `a` и дополнения до двух величины `value`, т. е.:

`x = a + (~value + 1);`

Предположим что значение `a` равно 27 и значение `value` равно 13, как и прежде. Если дополнение до двух `value` дает значение -13, то при сложении этой величины со значением `a` должен быть получен результат 14. Давайте проверим, так ли это:

<code>a (т. е. 27)</code>	<code>00000000 00000000 00000000 00011011</code>
<code>+(~value + 1)</code>	<code>+11111111 11111111 11111111 11110011</code>
<hr/>	<hr/>
	<code>00000000 00000000 00000000 00001110</code>

В итоге мы действительно получили число, равное 14.

Резюме

- Когда мы используем в программе на C++ целое число, например 19, или 227, или -63, то предполагаем, что оно записано в *десятичной системе счисления* (с основанием 10). Цифрами в десятичной системе счисления являются 0, 1, 2, 3, 4, 5, 6, 7, 8, и 9. Самая младшая цифра — это 0, а самая старшая цифра — 9, на единицу меньшая, чем 10 — *основание* системы счисления.
- Для внутреннего представления данных в компьютере используется *двоичная система счисления* (с основанием 2). Двоичная система счисления обходится только двумя цифрами, а именно, 0 и 1. Самая младшая цифра — 0, а самая старшая цифра — 1, на единицу меньшая основания системы счисления 2.
- Восьмеричная система счисления* (с основанием 8) и *шестнадцатеричная система счисления* (с основанием 16) завоевали себе популярность прежде всего возможностью сокращенной записи двоичных чисел.

- Цифрами восьмеричной системы счисления являются цифры десятичной системы от 0 до 7.
- В шестнадцатеричной системе счисления возникает проблема, связанная с тем, что для записи чисел требуется уже шестнадцать цифр: самая младшая цифра — 0, а самая старшая цифра должна иметь десятичное значение 15 (на единицу меньшее, чем основание системы 16). В соответствии с принятым соглашением, для представления шестнадцатеричных цифр, соответствующих десятичным значениям от 10 до 15, используются символы от A до F.
- Каждая из рассмотренных нами систем счисления использует *позиционную запись*, когда каждая позиция, в которой записывается цифра, имеет свое *позиционное значение*.
- Между восьмеричной и шестнадцатеричной системами счисления и двоичной системой имеется важная связь, которая заключается в том, что основания восьмеричной и шестнадцатеричной систем (8 и 16 соответственно) являются степенями основания двоичной системы счисления (равного 2).
- Чтобы преобразовать восьмеричное число в двоичное, нужно заменить каждую восьмеричную цифру числа на ее двоичное трехразрядное значение.
- Чтобы преобразовать шестнадцатеричное число в двоичное, нужно заменить каждую шестнадцатеричную цифру числа на ее двоичное четырехразрядное значение.
- Поскольку мы приучены к работе с десятичными числами, часто бывает нужно преобразовать двоичное, восьмеричное, или шестнадцатеричное число в десятичное, чтобы узнать привычное нам значение числа.
- Для того, чтобы преобразовать к десятичному виду число из другой системы счисления, умножьте десятичный эквивалент каждой цифры числа на ее *позиционное значение* и просуммируйте полученные величины.
- В памяти компьютера отрицательные числа представляются при помощи метода *дополнения до двух*.
- Чтобы получить отрицательное значение заданной величины, нужно сначала получить ее *дополнение до единицы* при помощи поразрядной операции дополнения (\sim). После этой операции значения всех битов числа меняются на противоположные. Затем, чтобы получить *дополнение* заданного значения *до двух*, нужно просто добавить единицу к дополнению этого числа 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 (\sim)

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 самопроверки

- Г.1.** Основаниями десятичной, двоичной, восьмеричной и шестнадцатеричной систем счисления являются числа _____, _____, _____ и _____ соответственно.
- Г.2.** Обычно десятичное, восьмеричное и шестнадцатеричное представления двоичного числа содержат (больше или меньше) цифр, чем исходное двоичное число.
- Г.3.** (Верно или неверно) Десятичная система счисления стала настолько популярной благодаря тому, что ее можно использовать для сокращенной записи двоичного числа путем простой замены каждой из десятичных цифр на группу из четырех двоичных разрядов.
- Г.4.** (Восьмеричное, или шестнадцатеричное, или десятичное) представление большого двоичного значения является наиболее коротким (среди предложенных вариантов).
- Г.5.** (Верно или неверно) В любой системе счисления старшая цифра больше основания системы на единицу.
- Г.6.** (Верно или неверно) В любой системе счисления младшая цифра меньше основания системы на единицу.
- Г.7.** Значение самой правой позиции в любом числе, записанном в двоичной, восьмеричной, десятичной или шестнадцатеричной системах счисления, всегда равно _____.
- Г.8.** Значение позиции, следующей за самой правой позицией в любом числе, записанном в двоичной, восьмеричной, десятичной или шестнадцатеричной системах счисления, всегда равно _____.
- Г.9.** Заполните в этой таблице отсутствующие значения четырех правых позиций для каждой из указанных систем счисления:
- | | | | | |
|---------------------------|------|-----|-----|-----|
| десятичная система | 1000 | 100 | 10 | 1 |
| шестнадцатеричная система | ... | 256 | ... | ... |
| двоичная система | ... | ... | ... | ... |
| восьмеричная система | 512 | ... | 8 | ... |
- Г.10.** Преобразуйте двоичное число 110101011000 в восьмеричное и шестнадцатеричное.
- Г.11.** Преобразуйте шестнадцатеричное число FACE в двоичное.
- Г.12.** Преобразуйте восьмеричное число 7316 в двоичное.
- Г.13.** Переведите шестнадцатеричное число 4FEC в восьмеричное. (Подсказка: сначала преобразуйте 4FEC в двоичное значение, а затем полученное двоичное число — в восьмеричное.)
- Г.14.** Преобразуйте двоичное число 1101110 в десятичное.
- Г.15.** Преобразуйте восьмеричное значение 317 в десятичное.
- Г.16.** Преобразуйте шестнадцатеричное число EFD4 в десятичное.
- Г.17.** Выполните преобразование десятичного числа 177 к двоичному, восьмеричному и шестнадцатеричному виду.

- Г.18.** Напишите двоичное представление десятичного числа 417. Затем вычислите его дополнения до единицы и до двойки.
- Г.19.** Что получится, если дополнение до единицы сложить с самими собой?

Ответы на упражнения для самопроверки

Г.1. 10, 2, 8, 16.

Г.2. Меньше.

Г.3. Неверно.

Г.4. Шестнадцатеричное.

Г.5. Неверно — старшая цифра в любой системе счисления на единицу меньше основания системы.

Г.6. Неверно — младшая цифра в любой системе счисления равна нулю.

Г.7. 1 (основание системы счисления, возведенное в нулевую степень).

Г.8. Основанию системы счисления.

Г.9. В приведенной ниже диаграмме заполнены отсутствующие значения четырех правых позиций для каждой из указанных систем счисления:

двоичная система	1000	100	10	1
шестнадцатеричная система	4096	256	16	1
двоичная система	8	4	2	1
восьмеричная система	512	64	8	1

Г.10. Восьмеричное число: 6530; шестнадцатеричное — D58.

Г.11. Двоичное число: 1111 1010 1100 1110.

Г.12. Двоичное число: 111 011 001 110

Г.13. Двоичное число: 0 100 111 111 101 100; восьмеричное — 47754.

Г.14. Десятичное число: $2+4+8+32+64=110$.

Г.15. Десятичное число: $7+1*8+3*64=7+8+192=207$.

Г.16. Десятичное число: $4+13*16+15*256+14*4096=61396$.

Г.17. Десятичное число 177 преобразуется в следующие значения:

двоичное значение:

$$\begin{array}{cccccccccc} 256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ (1*128) + (0*64) + (1*32) + (1*16) + (0*8) + (0*4) + (0*2) + (1*1) \\ 10110001 \end{array}$$

восьмеричное значение:

$$\begin{array}{ccccccc} 512 & 64 & 8 & 1 \\ 64 & 8 & 1 \\ (2*64) + (6*8) + (1*1) \\ 261 \end{array}$$

шестнадцатеричное значение:

256	16	1
16	1	
$(11 * 16) + (1 * 1)$		
$(B * 16) + (1 * 1)$		
B1		

Г.18. Двоичное значение:

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: сложение исходного двоичного числа и его дополнения до двух:

110100001
001011111

000000000

Г.19. Нуль.

Упражнения

Г.20. Есть люди, которые утверждают, что многие наши вычисления было бы проще выполнять в системе счисления по основанию 12, потому что 12 имеет больше делителей, чем число 10 (основание десятичной системы). Какова самая младшая цифра в системе по основанию 12? Какой символ можно было бы использовать для обозначения самой старшей цифры в системе по основанию 12? Чему равны значения четырех самых правых позиций в системе по основанию 12?

Г.21. Какую величину будет обозначать наибольшее числовое значение в позиции, следующей за самой правой позицией, во всех рассмотренных нами системах счисления.

Г.22. Дополните следующую таблицу позиционных значений четырех правых позиций для каждой из указанных систем счисления:

десятичная система	1000	100	10	1
система по основанию 6	6	...
система по основанию 13	...	169
система по основанию 3	27

Г.23. Преобразуйте двоичное число 10010111010 в восьмеричное и шестнадцатеричное значение.

Г.24. Переведите шестнадцатеричное значение 3A7D в двоичное.

Г.25. Выполните преобразование шестнадцатеричного числа 765F в восьмеричное. (Подсказка: сначала преобразуйте 765F к двоичному виду, а затем полученное двоичное число переведите в восьмеричное значение.)

Г.26. Преобразуйте двоичное число 1011110 в десятичное.

Г.27. Переведите восьмеричное число 426 в десятичное.