

ЯЗЫК ПРОГРАММИРОВАНИЯ С++. ЛЕКЦИИ И УПРАЖНЕНИЯ

Изучение языка программирования C++ является приключением, ведущим к открытиям. Это связано с тем, что C++ непрерывно развивается. В этой книге рассказывается о стандартном языке C++ без привязки к какому-либо одному виду компьютера, операционной системы или компилятора. Здесь вы найдете подробнейшее описание синтаксиса языка, сопровождаемое многочисленными практическими примерами. Цель книги - научить читателя создавать эффективные программы и получать от этого удовольствие.

Издание рассчитано на пользователей с любым уровнем подготовки. Для начинающих эта книга послужит превосходным учебником, а опытные программисты смогут воспользоваться ею в качестве справочника, облегчающего их ежедневный труд.

Учитывая уровень рассмотрения теоретических вопросов и наличие тщательно подобранных упражнений, книгу можно рекомендовать как методическое пособие преподавателям высших и средних учебных заведений, а также в качестве учебника для студентов.

Краткое оглавление

Глава 1. Готовимся изучать язык C++	21
Глава 2. Приступаем к изучению языка C++	32
Глава 3. Представление данных	52
Глава 4. Производные типы данных	79
Глава 5. Циклы и выражения сравнения	115
Глава 6. Операторы ветвления и логические операции	145
Глава 7. Функции языка C++	164
Глава 8. Работа с функциями	196
Глава 9. Объекты и классы	245
Глава 10. Работа с классами	274
Глава 11. Классы и динамическое распределение памяти	306
Глава 12. Наследование классов	345
Глава 13. Повторное использование программного кода в C++	381
Глава 14. Дружественные конструкции, исключения и прочее	424
Глава 15. Класс string и стандартная библиотека шаблонов	465
Глава 16. Ввод/вывод данных и работа с файлами	516
Приложение А. Системы счисления	568
Приложение В. Ключевые слова языка C++	570
Приложение С. Таблица кодов ASCII	571
Приложение Д. Приоритет операций	575
Приложение Е. Другие операции	578
Приложение F. Класс шаблона STRING	583
Приложение G. Методы и функции библиотеки STL	594
Приложение Н. Рекомендуемая литература	615
Приложение I. Преобразование программного кода в соответствии со стандартом ANSI/ISO C++	617

Оглавление

Глава 1. Готовимся изучать язык C++	21
Изучение языка C++	21
Немного истории	22
Язык С	22
Философия программирования, заложенная в языке С	23
Объектно-ориентированное программирование	23
Обобщенное программирование	24
Язык C++	24
Переносимость и стандарты	25
Методика создания программ	26
Создание исходного кода	27
Компиляция и компоновка	28
Компиляция и компоновка в операционной системе UNIX	28
Реализации Turbo C++ 2.0 и Borland C++ 3.1 (DOS)	29
Компиляторы для операционной системы Windows	29
Компиляторы для компьютеров Macintosh	30
Соглашения, используемые в этой книге	31
Наша система	31
Глава 2. Приступаем к изучению языка C++	32
Начальные сведения о языке C++	32
Функция main()	33
Заголовок функции в роли интерфейса	34
Когда функция main() не соответствует своему названию?	35
Комментарии в языке C++	35
Препроцессор C++ и файл iostream	36
Имена заголовочных файлов	36
Области имен	36
Вывод данных в языке C++ с использованием объекта cout	37
Символ новой строки (\n)	38
Форматирование исходного кода C++	39
Формат исходного кода программ C++	40
Краткие сведения об операторах языка C++	40
Операторы объявления и переменные	40
Оператор присваивания	41
Важный момент, связанный с объектом cout	41
Еще несколько операторов языка C++	42
Применение объекта cin	42
И снова объект cout	43
Несколько слов о классах	43
Функции	44
Применение функций с возвращаемым значением	44

Разновидности функций	46
Функции, определяемые пользователем	47
Формат определения функции	47
Заголовки функций	48
Определяемая пользователем функция с возвращаемым значением	49
Итак, операторы	50
Резюме	50
Вопросы для повторения	51
Упражнения по программированию	51
Глава 3. Представление данных	52
Простые переменные	52
Имена переменных	53
Целочисленные типы данных	54
Типы данных short, int и long	54
Примечания к программе	56
Типы данных без знака	57
Какой тип выбрать?	58
Целочисленные константы	59
Определение типа константы в C++	60
Тип данных char: символы и малые целые числа	60
Примечания к программе	61
Функция-элемент: cout.put()	62
Константы типа char	62
Типы данных signed char и unsigned char	64
Тип данных wchar_t	64
Новый тип данных bool	65
Квалификатор const	65
Числа с плавающей точкой	66
Запись чисел с плавающей точкой	66
Типы данных с плавающей точкой	67
Примечания к программе	69
Константы с плавающей точкой	69
Преимущества и недостатки типов данных с плавающей точкой	69
Арифметические операции языка C++	70
Приоритет операций и ассоциативность	71
Разновидности операции деления	71
Операция деления по модулю	72
Преобразования типов данных	73
Преобразование при присваивании	73
Преобразования данных в выражениях	74
Преобразования данных при передаче аргументов	75
Приведение типов	75
Резюме	77
Вопросы для повторения	77
Упражнения по программированию	78

Глава 4. Производные типы данных	79
Краткий обзор массивов	79
Примечания к программе	81
Еще об инициализации массива	81
Строки	82
Конкатенация строк	83
Использование строк в массиве	83
Примечания к программе	84
Возможные нюансы при строковом вводе	84
Строчно-ориентированный ввод: getline() и get()	85
Пустые строки и другие проблемы	87
Смешанный строчно-числовой ввод	87
Краткий обзор структур	88
Примечания к программе	90
Другие свойства структуры	90
Массивы структур	91
Разрядные поля	92
Объединения	92
Перечисления	93
Установка значений перечислителя	94
Диапазоны значений для перечислений	94
Указатели и свободная память	95
Объявление и инициализация указателей	97
Указатели и числа	98
Распределение памяти с помощью оператора new	99
Примечания к программе	100
Освобождение памяти с помощью	100
оператора delete	100
Использование оператора new для создания динамических массивов	101
Создание динамического массива с помощью оператора new	101
Использование динамического массива	102
Указатели, массивы и арифметика указателей	103
Примечания к программе	104
Основные сведения об указателях	105
Указатели и строки	106
Примечания к программе	106
Использование оператора new для создания динамических структур	109
Пример использования операторов new и delete	110
Примечания к программе	111
Автоматическая, статическая и свободная память	111
Автоматические переменные	112
Статическая память	112
Свободная память	112
Резюме	112
Вопросы для повторения	113

Упражнения по программированию	114
Глава 5. Циклы и выражения сравнения	115
Обзор возможностей цикла for	115
Составные элементы цикла for	116
Выражения и операторы	117
Операторы и выражения	119
Незначительное нарушение правил	119
Снова о цикле for	120
Примечания к программе	120
Изменение шага цикла	121
Доступ к символам строки с помощью цикла for	121
Операторы инкремента (++) и декремента (--)	122
Комбинированные операторы присваивания	123
Составные операторы или блоки	123
Оператор "запятая" (или дополнительные синтаксические приемы)	124
Примечания к программе	125
Особенности применения оператора "запятая"	126
Выражения сравнения	126
Типичная ошибка программирования	126
Сравнение строк	128
Примечания к программе	129
Цикл while	129
Примечания к программе	130
Сравнение циклов for и while	131
Небольшая пауза	132
Цикл do while	133
Циклы и ввод текста	134
Применение простого оператора cin для ввода данных	134
Примечания к программе	135
На помощь приходит функция cin.get(char)	135
Выбор функции cin.get()	136
Условие конца файла	136
Конец файла означает конец ввода	137
Распространенные идиомы	138
Еще одна разновидность функции cin.get()	138
Вложенные циклы и двумерные массивы	140
Инициализация двумерного массива	141
Резюме	142
Вопросы для повторения	143
Упражнения по программированию	143
Глава 6. Операторы ветвления и логические операции	145
Оператор if	145
Оператор if else	146
Форматирование операторов if else	147
Конструкция if else if else	147

Логические выражения	148
Операция логического ИЛИ:	148
Операция логического И: &&	149
Примечания к программе	150
Определение диапазонов с помощью операции &&	150
Примечания к программе	151
Операция логического НЕ: !	151
Примечания к программе	152
Немного о логических операциях	152
Библиотека символьных функций ctype	153
Оператор ?:	154
Оператор switch	155
Использование перечислителей в качестве меток	157
Операторы switch и if else	157
Операторы break и continue	158
Примечания к программе	158
Циклы считывания чисел	159
Примечания к программе	161
Резюме	161
Вопросы для повторения	161
Упражнения по программированию	162
Глава 7. Функции языка C++	164
Обзор функций	164
Определение функции	165
Прототипирование и вызов функций	166
Почему именно прототипы?	167
Синтаксис прототипа	167
Польза прототипов	168
Аргументы функции и передача по значению	169
Функции с несколькими аргументами	169
Примечания к программе	171
Еще одна функция с двумя аргументами	171
Примечания к программе	172
Функции и массивы	172
Массивы и указатели (продолжение)	173
Трудности, возникающие при использовании массивов в качестве аргументов	174
Примечания к программе	175
Другие виды функций, выполняющих обработку массивов	176
Заполнение массива	176
Отображение массива и его защита с помощью спецификатора const	177
Модификация элементов массива	177
Объединение частей в единое целое	178
Примечания к программе	179

Указатели и спецификатор const	179
Функции и строки в стиле C	180
Примечания к программе	181
Функции, возвращающие строки	182
Примечания к программе	182
Функции и структуры	183
Передача и возврат структур	183
Еще один пример	184
Примечания к программе	186
Передача адресов структур	187
Рекурсия	188
Примечания к программе	189
Указатели на функции	189
Назначение указателя на функцию	190
Получение адреса функции	190
Объявление указателя на функцию	190
Использование указателя для вызова функции	191
Резюме	192
Вопросы для повторения	193
Упражнения по программированию	193
Глава 8. Работа с функциями	196
Встроенные функции	196
Ссылочные переменные	198
Создание ссылочных переменных	198
Ссылки в роли параметров функции	200
Примечания к программе	202
Свойства и особенности ссылок	202
Временные переменные, ссылочные аргументы и модификатор const	203
Использование ссылок при работе со структурами	204
Примечание к программе	205
Некоторые соображения по вопросу о том, когда возвращать ссылку или указатель	206
Когда имеет смысл пользоваться ссылочными аргументами	207
Аргументы, заданные по умолчанию	207
Примечания к программе	209
Полиморфизм функций (перегрузка функций)	209
Пример перегрузки	211
Когда целесообразно использовать перегрузку функции	212
Шаблоны функций	212
Перегруженные шаблоны	214
Явная специализация	215
Метод первой генерации	216
Вторая генерация	216

Третья генерация	216
Пример	217
Образование шаблонов и специализация	218
Выбор функций	219
Точное соответствие и наилучшее соответствие	220
Функции со многими аргументами	221
Раздельная компиляция	221
Классы памяти, диапазоны доступа и связывание	224
Диапазон доступа и связывание	224
Автоматическая память	225
Автоматические переменные и работа со стеком	226
Переменные типа register	227
Статический класс памяти	228
Внешние переменные	228
Примечания к программе	229
Модификатор static (локальные переменные)	230
Связывание и внешние переменные	231
Спецификаторы классов памяти:const, volatile и mutable	233
Более подробно о спецификаторе const	234
Классы памяти и функции	235
Языковое связывание	235
Классы памяти и динамическое распределение	236
Пространства имен	236
Традиционные пространства имен языка C++	237
Новые свойства пространства имен	238
Объявления использования и директивы using	238
Немногое о свойствах пространства имен	240
Неименованные пространства имен	241
Пространства имен и будущее	241
Резюме	241
Вопросы для повторения	242
Упражнения по программированию	243
Глава 9. Объекты и классы	245
Процедурное и объектно-ориентированное программирование	245
Абстрагирование и классы	246
Что представляет собой тип	247
Класс	247
Общедоступный или приватный?	249
Реализация классов и функций-элементов	250
Примечания, касающиеся функций-элементов	251
Встроенные методы	252
Выбор объекта	252
Использование классов	253
Текущее состояние дел	254
Деструкторы и конструкторы классов	255

Объявление и определение конструкторов	256
Использование конструктора	256
Конструктор, заданный по умолчанию	257
Деструкторы	258
Совершенствование класса Stock	258
Заголовочный файл	259
Файл реализации	259
Клиентский файл	260
Примечания к программе	260
Функции-элементы типа const	261
Обзор конструкторов и деструкторов	262
Работа с указателем this	263
Массив объектов	266
Диапазон доступа класса	267
Абстрактный тип данных	268
Резюме	271
Вопросы для повторения	272
Упражнения по программированию	272
Глава 10. Работа с классами	274
Перегрузка операций	275
Время в нашем распоряжении	276
Добавление операции сложения	277
Ограничения при выполнении перегрузки	278
Другие перегруженные операции	279
Использование дружественных структур	280
Создание дружественных конструкций	281
Общий вид дружественной конструкции: перегрузка операции <<	282
Первая версия перегрузки операции <<	282
Вторая версия перегрузки операции <<	283
Перегруженные операции: дружественные и обычные функции	285
Перегрузка: класс Vector	286
Использование элементов состояния	291
Еще немного о перегрузке	291
Умножение	292
Некоторые уточнения: перегрузка и перегруженная операция	292
Комментарий к реализации	293
Применение класса Vector к решению задачи случайного блуждания	293
Примечания к программе	295
Автоматические преобразования и приведение типов для классов	295
Примечания к программе	298
Функции преобразования	299
Автоматическое выполнение преобразования типов	300
Преобразования и дружественные конструкции	302
Осуществление выбора	303
Резюме	303

Вопросы для повторения	304
Упражнения по программированию	305
Глава 11. Классы и динамическое распределение памяти	306
Динамическая память и классы	306
Обзорный пример и элементы статических классов	307
Примечания к программе	311
Новый подход к использованию операторов new и delete	313
Устранение проблем, связанных с классом String	314
Неявные функции-элементы	315
Конструктор, заданный по умолчанию	315
Конструктор копирования	315
Условия применения конструктора копирования	315
Функции конструктора копирования	316
К чему могут привести возможные ошибки	316
Оператор присваивания	318
Применение оператора присваивания	318
Функции оператора присваивания	319
К чему могут привести возможные ошибки	319
Фиксированное присваивание	319
Новый, усовершенствованный класс String	320
Применение оператора new в конструкторах	325
Применение указателей при работе с объектами	326
Обзор технических методов	328
Перегрузка операции <<	328
Функции преобразования	329
Классы, конструкторы которых применяют оператор new	329
Моделирование очереди	329
Класс Queue	330
Интерфейс	330
Реализация	330
Методы класса	332
Немного сведений о других методах классов	334
Класс Customer	335
Моделирование	338
Резюме	341
Вопросы для повторения	342
Упражнения по программированию	343
Глава 12. Наследование классов	345
Простой базовый класс	346
Наследование — отношение is-a	347
Объявление производного класса	349
Реализация производного класса	351
Инициализация объектов объектами	352
Другие функции-элементы	352
Примечания к программе	355

Управление доступом — protected	355
Отношение is-a, ссылки и указатели	356
Виртуальные функции-элементы	357
Активизация динамического связывания	358
Зачем нужны два вида связывания?	360
Как работают виртуальные функции	361
Что следует знать о виртуальных функциях	362
Конструкторы	362
Деструкторы	362
Дружественные конструкции	362
Отсутствие переопределения	362
Переопределение скрывает методы	362
Наследование и присваивание	363
Смешанное присваивание	364
Присваивание и динамическое распределение памяти	365
Случай 1. Производный класс не использует оператор new	366
Случай 2. Производный класс использует оператор new	367
Абстрактные базовые классы	370
Обзор структуры класса	371
Функции-элементы, которые генерирует компилятор	371
Заданный по умолчанию конструктор	371
Конструктор копирования	372
Оператор присваивания	372
Другие соображения по поводу методов класса	372
Конструкторы	372
Деструкторы	372
Преобразования	373
Передача объекта по значению и передача по ссылке	373
Возврат объекта и ссылки	373
Использование const	374
Соображения по поводу общедоступного наследования	374
Отношение is-a	374
Какие объекты не наследуются	375
Оператор присваивания	375
Приватные и защищенные элементы	376
Виртуальные методы	376
Деструкторы	376
Итоговый анализ функций класса	377
Резюме	377
Вопросы для повторения	378
Упражнения по программированию	378
Глава 13. Повторное использование программного кода в C++	381
Классы, включающие элементы объектов	381
Класс ArrayDb	382
Работа с operator[]()	384

Альтернатива с использованием спецификатора const	384
Пример класса Student	385
Инициализация включенных объектов	385
Использование интерфейса для включенного объекта	388
Использование нового класса	389
Приватное наследование	390
Пример класса Student (новая версия)	390
Инициализация компонентов базового класса	390
Использование методов базового класса	391
Использование измененного класса Student	392
Включение или приватное наследование?	393
Защищенное наследование	393
Переопределение доступа с помощью объявления using	394
Шаблоны классов	394
Определение шаблона класса	395
Использование класса шаблона	397
Более подробное рассмотрение шаблона класса	398
Неправильное использование стека указателей	398
Корректное использование стека указателей	399
Примечания к программе	401
Шаблон массива и аргументы, не являющиеся типами	401
Использование шаблона вместе с семейством классов	402
Примечание к программе	406
Многосторонность шаблона	406
Специализации шаблонов	408
Неявные образования экземпляров	408
Явные образования экземпляров	408
Явные специализации	408
Частичные специализации	409
Множественное наследование	409
Определение количества рабочих	410
Виртуальные базовые классы	411
Новые правила конструктора	412
Выбор метода	412
Смешанные виртуальные и невиртуальные базовые классы	414
Виртуальные базовые классы и доминирование	414
Некоторые итоги по теме множественного наследования	419
Резюме	420
Вопросы для повторения	421
Упражнения по программированию	422
Глава 14. Дружественные конструкции, исключения и прочее	424
Дружественные структуры	424
Дружественные классы	424
Дружественные функции-элементы	428
Другие дружественные отношения	429

Общедоступные дружественные элементы	431
Шаблоны и дружественные элементы	431
Вложенные классы	432
Вложенные классы и доступ	433
Диапазон доступа	433
Управление доступом	434
Вложение в шаблоне	434
Исключения	437
Примечания к программе	439
Механизм исключений	439
Примечания к программе	439
Разносторонность исключений	441
Многочисленные блоки try	442
Разворачивание стека	443
Дополнительные опции	444
Исключения и классы	445
Исключения и наследование	446
Класс exception	451
Исключение bad_alloc и оператор new	453
Проблемы, связанные с исключениями	453
Замечание об исключениях	455
Библиотека RTTI	456
Назначение RTTI	456
Принципы функционирования RTTI	456
Оператор dynamic_cast	457
Оператор typeid и класс type_info	458
Проблемы, возникающие при использовании RTTI	461
Операторы приведения типов	462
Резюме	463
Вопросы для повторения	464
Упражнения по программированию	464
Глава 15. Класс string и стандартная библиотека шаблонов	465
Класс string	465
Создание строки	465
Примечания к программе	466
Реализация ввода в классе string	468
Работа со строками	469
Примечания к программе	472
Что еще?	472
Класс auto_ptr	472
Использование шаблона auto_ptr	473
Некоторые замечания	474
Стандартная библиотека шаблонов	475
Класс шаблонов vector	476
Операции, допустимые при работе с шаблонами vector	477

Дополнительные операции с векторами	480
Обобщенное программирование	483
Почему именно итераторы?	483
Типы итераторов	486
Итератор ввода	486
Итератор вывода	486
Прямой итератор	487
Двусторонний итератор	487
Итератор произвольного доступа	487
Иерархия итераторов	488
Концепции, уточнения и модели	488
Указатель как итератор	489
Итераторы copy(), ostream_iterator и istream_iterator	489
Другие полезные итераторы	490
Типы контейнеров	492
Концепция контейнера	493
Последовательности	495
Класс vector	496
Класс deque	497
Класс list	497
Примечания к программе	498
Инструментарий для работы с контейнером list	499
Класс queue	499
Класс priority_queue	499
Класс stack	500
Ассоциативные контейнеры	500
Пример класса set	500
Пример класса multimap	503
Функциональные объекты (функторы)	504
Концепции функторов	505
Предопределенные функторы	506
Адаптируемые функторы и функции-адаптеры	507
Алгоритмы	508
Группы алгоритмов	509
Общие свойства	509
Использование STL	510
Другие библиотеки	513
Резюме	513
Вопросы для повторения	514
Упражнения по программированию	514
Глава 16. Ввод/вывод данных и работа с файлами	516
Обзор ввода/вывода данных в C++	516
Потоки и буферы	517
Потоки, буферы и файл iostream	518
Перенаправление	520

Вывод с помощью cout	520
Перегруженная операция <<	521
Вывод и указатели	521
Конкатенация вывода	522
Другие методы класса ostream	522
Очистка буфера вывода	523
Форматирование вывода с помощью cout	524
Изменение системы счисления при выводе	525
Установка ширины полей	526
Символы-заполнители	527
Установка точности при выводе чисел с плавающей точкой	527
Вывод замыкающих нулей и десятичной точки	528
Подробнее о функции setf()	528
Стандартные манипуляторы	532
Заголовочный файл iomanip	533
Ввод данных с помощью cin	534
Как cin >> рассматривает поток ввода	535
Состояние потока	536
Установка состояния	536
Ввод/вывод и прерывания	537
Эффекты состояния потока	537
Другие методы класса istream	538
Односимвольный ввод	538
Выбор формы односимвольного ввода	540
Строковый ввод: getline(), get() и ignore()	540
Неожиданный строковый Ввод	542
Другие методы класса istream	543
Примечания к программе	544
Ввод/вывод файлов	545
Простой файловый ввод/вывод	546
Открытие нескольких файлов	548
Работа в режиме командной строки	548
Проверка потока и функция is _open()	550
Режимы файлов	550
Добавление к файлу	552
Двоичные файлы	554
Произвольный доступ к файлам	557
Внутреннее форматирование	560
Что дальше?	564
Резюме	564
Вопросы для повторения	565
Упражнения по программированию	566
Приложение А. Системы счисления	568
Восьмеричные числа	568
Шестнадцатиричные числа	568

Двоичные числа	569
Двоичные и шестнадцатиричные числа	569
Приложение В. Ключевые слова языка C++	570
Приложение С. Таблица кодов ASCII	571
Приложение D. Приоритет операций	575
Приложение Е. Другие операции	578
Поразрядные операции	578
Операции смещения	578
Логические поразрядные операции	579
Несколько стандартных приемов по работе с битами	580
Установка бита	580
Переключение бита	580
Отмена установленного бита	581
Проверка значения бита	581
Операции разыменования элементов	581
Приложение F. Класс шаблона STRING	583
Тринадцать типов и констант	583
Информация о данных, конструкторах и другие сведения	584
Стандартные конструкторы	585
Конструкторы, использующие массив	586
Конструкторы, использующие часть массива	586
Конструктор копирования	586
Конструктор, использующий п копий символа	587
Конструкторы, использующие диапазон значений	587
Ячейки памяти	587
Доступ к строке	588
Базовые методы присваивания	588
Поиск строковых величин	588
Семейство методов find()	588
Семейство методов rfind()	589
Семейство методов find_first_of()	589
Семейство методов find_last_of()	589
Семейство методов find_first_not_of()	589
Семейство методов find_last_not_of()	590
Функции и методы сравнения	590
Модификаторы строковых значений	591
Конкатенирование и дописывание	591
Дополнительные средства присваивания	591
Методы вставки	591
Методы удаления	592
Методы замены	592
Другие методы изменения: copy() и swap()	592
Вывод и ввод данных	593
Приложение G. Методы и функции библиотеки STL	594
Элементы, общие для всех контейнеров	594

Дополнительные элементы для векторов, списков и очередей с двухсторонним доступом	595
Дополнительные элементы для наборов и карт	596
Функции STL	597
Неизменяющаяся последовательность операций	598
Изменяющаяся последовательность операций	601
Операции сортировки и связывания	606
Сортировка	606
Двоичный поиск	608
Слияние	609
Операции над наборами	609
Операции над кучей	611
Минимум и максимум	611
Перестановки	612
Операции с числами	613
Приложение Н. Рекомендуемая литература	615
Стандарт ISO/ANSI	616
Приложение I. Преобразование программного кода в соответствии со стандартом ANSI/ISO C++	617
Директивы препроцессора	617
При объявлении констант лучше использовать const вместо #define	617
Для определения небольших функций используйте ключевое слово inline вместо #define	618
Используйте прототипы функций	619
Приведение типов	619
Познакомьтесь со свойствами языка C++	619
Используйте новую структуру заголовочных файлов	619
Использование пространств имен	619
Использование шаблона auto_ptr	620
Использование класса string	620
Использование STL	621
Приложение J. Ответы на вопросы для повторения	622
Предметный указатель	634

Предметный указатель

Символы

* 96
++ 122
-- 122
<< 37, 521
= 41
|| 148

А

Абстрагирование 246
Автоматическая память 225

Алгоритм 508
Аргумент 34, 44
 заданный по умолчанию 207
Арифметические операции 70
Ассоциативность 71
Б
Байт 54
Библиотека 27
 RTTI 456
 STL 594

символьных функций с type 153

Бит 54

Буфер 517

В

Ввод 87

смешанный строчно-числовой 87
данных 534
текста 134

Ввод/вывод 516, 545, 593
перенаправление 520

Вектор 286, 476

Виртуальные методы 376

Виртуальный базовый класс 411

Вывод данных 37

Выражение 70, 126

сравнения 126

Д

Двоичный поиск 608

Деструктор 255, 264, 362, 372

Диапазон доступа класса 224, 250, 267

Динамическая память 306

Динамическое распределение памяти
236, 365

Динамическое связывание 358

Директива Using 37, 238

Дружественные элементы 431

З

Значащие цифры 67

И

Имена в C++ 236

Индекс 80

Инициализация 55

Инкапсуляция 249

Интегрированная среда разработки
(IDE) 27, 29

Исключения 437

Bad_alloc 453

непредвиденное 453

Исходный код 26

Итератор 486

copy() 489

istream_iterator 489

ostream_iterator 489

ввода 486

вывода 486

двусторонний 487

другие 490

произвольного доступа 487

прямой 487

К

Квалификатор 65

const 65

Класс

Allocator 472

Auto_ptr 472

Customer 335

Deque 497

Ellipse 370

Exception 451

Istream 43

Ust 497

Multimap 503

Ostream 43, 521, 525

Priority_queue 499

Push_back() 496

Push_front() 496

Queue 330, 499

Stack 500

Stock 258

String 307, 320, 465, 583

конструкторы 466

Vector 26, 476, 496

абстрактный базовый 370

базовый 346

виртуальный базовый 411

включающий элементы объектов
381

вложенный 332, 432

диапазон доступа 433

дружественный 424

наследование 345

производный 346

шаблоны классов 394

Классы памяти 227, 235

статические 228

Ключевые слова языка C++ 48, 570

Командная строка 548

Комментарий 35

Компилятор 22

Компиляция 28, 221

раздельная 221
Компоновка 28
Компьютер 40
Конкатенация строк 83
Константы 56, 59, 62
 с плавающей точкой 69
 символические 56
 символьные 62
 целочисленные 59
Конструктор 255, 262, 362, 372, 585
 копирования 315, 372, 586
 стандартный 585
 строковый 585
Контейнер 492, 493, 500
 ассоциативный 500
Куча 611

Л

Лексема 39
Логические выражения 148
Логические операции 145

М

Малые целые числа 60
Массив 59, 79, 103, 140, 172
 Двумерный 140
 Динамический 101
 Имя массива 106
 Индекс 80
Метод
 Begin() 485
 Copy() 592
 End() 485
 Erase() 478, 592
 Find() 588
 Find_first_not_of() 589
 Find_first_of() 589
 Find_last_not_of() 590
 Find_last_of() 589
 Get(char &) 541
 Get(void) 541
 Insert() 591
 Read() 555
 Replace() 592
 Rfind() 589
 Swap() 592
 Width() 526

О

Write() 555
виртуальный 376
включения 381
встроенный 252
приватного и защищенного
 наследования 381
сравнения 590
Множественное наследование 409
Моделирование 338
Модификатор 230, 591
 static 230
 const 203

Н

Наследование 363, 390, 446
 приватное 390
Нехватка памяти 100

О

Область имен 36, 37
Обобщенное программирование 483
Объединение 92
 анонимное 93
Объект 23, 37, 245, 520
 cin 42, 534
 cout 37, 520
Объектно-ориентированное
 программирование 249
Операнд 70
Оператор 34, 40, 55, 62, 99, 122, 145, 453
 ?: 154
 break 158
 continue 158
 if 145
 if else 146
 If else if else 147
 new 99, 101, 325, 453
 switch 155
 декремента (--) 122
 "запятая" 124
 инкремента (++) 122
 комбинированные операторы
 присваивания 123
 объявления 40
 приведения типов 462
 принадлежности 62

присваивания 41, 375
составной 123

Операция 55, 70, 96, 145, 148, 521
* (косвенное значение) 96
<< 521
арифметическая 70
деления 71
логическая 145
И (&&) 149
ИЛИ (||) 148
НЕ(!) 151
приоритет операций 71
над наборами 609

Операционная система 22

Освобождение памяти 100
Отношение 347, 374, 381
has-a ("содержит объект") 348,
381
is-a ("является объектом") 347,
374, 382
uses-a 348

Очередь 329

П

Память 95, 99, 111, 225, 236, 306
автоматическая 111, 225
динамическая 236, 306
нехватка памяти 100
освобождение памяти 100
распределение 99
свободная 95, 111
статическая 111

Параметр 34, 44

Перегрузка 209
операции 38, 275
<< 282, 328

Переменная 40, 52, 112, 198
Boolean 65
автоматическая 112
внешняя 228, 232
временная 203
глобальная 230
именование 53
локальная 230
простая 52
ссылочная 198

Р

Раздельная компиляция 221

Разрядные поля 92

Распределение памяти
динамическое 236, 365

Расширение типов 558

Рекурсия 188

С

Свободная память 95

Связывание 101, 227, 231, 235, 358
динамическое 101, 105, 358
статическое 101, 105, 357
языковое 235

Сигнатура 210

Символ

типа register 227

Переносимость 25

Перестановки 612

Перечисления 93

Подпрограмма 47

Поиск
двоичный 608

Полиморфизм 209

Последовательность 495

Поток 517

Преобразование типов данных 73

Препроцессор C++ 36

Приватное наследование 390

Приведение типов 75

Приоритет операций 71, 575

Присваивание 363
смешанное 364

Профаммирование 21, 245, 483
обобщенное 24, 483

объектно-ориентированное 21, 23,
245, 249

процедурное 21, 23, 245

сверху вниз 23

снизу вверх 24

структурное 23

Произвольный доступ к файлам 557

Пространство имен 236

Прототип 167

Процедура 47

Псевдоним типа 133

Р

Раздельная компиляция 221

Разрядные поля 92

Распределение памяти
динамическое 236, 365

Расширение типов 558

Рекурсия 188

С

Свободная память 95

Связывание 101, 227, 231, 235, 358
динамическое 101, 105, 358
статическое 101, 105, 357
языковое 235

Сигнатура 210

Символ

-заполнитель 527
новой строки (\n) 38
сигнальной метки 134
Символические константы 56
Система счисления 568
восьмеричная 568
двоичная 569
шестнадцатиричная 568
Слияние 609
Сокрытие данных 249
Сортировка 481, 606
Специализация 215, 218, 396, 408
 явная 215
Спецификатор const 179
Спецификаторы классов памяти 233
Сравнение строк 128
Среда разработки 27
Ссылка 200, 356
Стандарт 25
 ANSI 31
 ISO/ANSI 616
Стандартная библиотека шаблонов (STL) 465, 475
Стандартные конструкторы 585
Статический класс памяти 228
Статическое связывание 357
Стек 270, 443
Стоун 49
Строка 37, 82
 конкатенация строк 83
Структура 88, 183, 250, 332
 вложенная 332
 динамическая 109
 дружественная 280
 Т
Таблица 361
Таблица кодов ASCII 571
Тип данных 54, 247, 268
 Boolean 65
 абстрактный 268
 без знака (unsigned) 57
 основной (char, short, int и long)
 54
 преобразование типов данных 73
с плавающей точкой (float, double и long double) 67
символьный (char) 62
целочисленный (short, int и long)
 54
 У
Указатель 95, 98, 103, 106, 173,
 326, 356, 521
this 263
 j
арифметика указателей 105
на функции 189 >
объявление указателей 105
присвоение значений указателям
 105
разыменование указателей 105
Управление доступом 355
Управляющие последовательности языка C++ 63
 Ф
Файл 36, 518
 iomanip 533
 iostream 36, 518
Флаг 528
Форматирование 560
 внутреннее 560
Функции 428
 дружественные 428
Функции-адаптеры 507
Функциональные объекты (функции)
 504
 адаптируемые 507
 предопределенные 506
Функция
 Accumulate() 613
 Adjacent_difference() 614
 Adjacent_find() 600
 Binary_search() 609
 Cin.get() 540
 Cin.get(ch) 540
 Cin.get(char) 135
 Copy() 489, 601
 Copy_backward() 601
 Count_if() 600
 Count() 600
 Cout.put() 62

Equal() 600
Equal_range() 609
Exit() 252
Fill() 604
Fill_n() 604
Find() 480, 599
Find_ar() 485
Find_end() 599
Find_first_of() 600
Find_if() 599
Find_ll() 485
For_each() 599
Gcount() 543
Generator() 604
Generator_n() 604
Get() 540
Get(char *, int, char) 541
Getline() 85, 540
Getline(char *, int, char) 542
Ignore() 540
Includes() 610
Inner_product() 614
Inplace_merge() 609
Is_open() 550, 552
Isspace() 538
Iter_swap() 603
Lexicographical_compare() 612
Lower_bound() 608
Main() 33
Makeheap() 611
Max() 612
Max_element() 612
Merge() 609
Min() 612
Min_etement() 612
Next_permutation() 613
Nth_element() 608
Operator>() 320
Partial_sort() 608
Partial_sort_copy() 608
Partial_sum() 614
Partition() 605
Peek() 543
Pop_heap() 611
Pow() 46
Precision() 527
Previous_permutation() 613
Printf() 42
Push_heap{} 611
Putbackf() 543
Random_shuffle() 605
Read() 543
Remove() 604
Remove_copy() 604
Remove_copyjf() 604
Remove_if() 604
Replace_if() 603
Replace_copy() 603
Replace_copy_if() 603
Reverse_copy() 605
Reversed 605
Rotated 605
Rotate_copy() 605
Search() 601
Search_n() 601
Seekg() 558
Set_difference() 610
Set_intersection() 610
Set_symmetric_difference() 611
Set_union() 610
Setf() 528
Sort() 480, 606
Sqrt() 45
Stable_partition() 606
Stable_sort() 608
strcmp() 320
Swap() 603
Swap_ranges() 603
Transform() 603
Unique() 605
Unique_copy() 605
Upper_bound() 608
WorseThan() 481
аргументы 169
встроенная 196
вызов 166
обзор 164
определенная пользователем 47
прототипирование 166
со многими аргументами 221

сравнения 590
Функция-элемент 62, 250, 309, 357
 виртуальная 357, 362, 371
 неявная 315
 свойства 377
 статическая 309

Ц
Целочисленная константа 59
Целые числа 54, 60
Цикл 115
 Do while 135
 For 115
 While 129
 вложенный 140
 шаг цикла 117

Цифры
 значащие 67

Ч
Числа 54, 100
с плавающей точкой 66 целые 54

Ш
Шаблон 212, 412, 431
 класса 394, 412

перегруженный 214
Э
Экземпляр 396
 шаблона 218

Элемент
 статических данных 308
 статического класса 307
 дружественный 431

Я
Язык 22
 Ассемблер 22
 машинный 26
 С 22
 C++ 24

Иностранные термины
ADT - abstract data types 271
ANSI 31
FIFO 329
IDE 29
UFO 270, 329
RTTI 456
STL - Standard Template Library 465, 594

Об авторе

Стивен Прата (Stephen Prata) преподает астрономию, физику, вычислительную технику и программирование в морском колледже города Кентфилд штата Калифорния. Он получил степень бакалавра наук в Калифорнийском технологическом институте и степень доктора философии в Калифорнийском университете (г. Беркли). Стивен Прата является автором и соавтором более десяти книг Waite Group, включая *Artificial Life Playhouse* и *Certified Course in Visual Basic 4*. Он также написал книгу *New C Primer Plus* Waite Group. Ассоциация "Компьютерпресс" (Computer Press) наградила эту книгу премией "Лучшая книга по компьютерам класса How-to в 1990 году". Еще одна его книга, *C++ Primer Plus* Waite Group, была выдвинута на соискание премии "Лучшая книга по компьютерам класса How-to в 1991 году".

Посвящение

Моим коллегам и студентам из морского колледжа, вместе с которыми мне очень приятно работать.

— Стивен Прата

Благодарности

По третьему изданию

Я хотел бы поблагодарить редакторов издательства Макмиллан и Waite Group за их помощь при создании этой книги: Трейси Дункельбергер (Tracy Dunkelberger), Сюзан Уолтон (Susan Walton) и Андреа Розенберг (Andrea Rosenberg). Благодарю также Русса Джэкобса (Russ Jacobs) за литературное и техническое редактирование текста. Еще я хотел бы сказать спасибо сотрудникам компании Metrowerks: Дейву Марку (Dave Mark), Алексу Харперу (Alex Harper) и особенно Рону Лиекти (Ron Liechty) за их неоценимый вклад в работу над этой книгой.

По второму изданию

Благодарю Митчела Уэйта (Mitchell Waite) и Скота Каламара (Scott Calamar) за помощь при работе над вторым изданием, а также Джоэля Фугазотто (Joel Fugazzotto) и Джоан Миллер (Joanne Miller) за то, что они довели этот проект до успешного завершения. Спасибо Майклу Маркотти (Michael Marcotty) из Metrowerks за ответы на мои вопросы относительно бета-версии компилятора CodeWarrior. Хочу также выразить свою признательность инструкторам Джеффу Бакуолтеру (Jeff Buckwalter), Эрлу Бриннеру (Earl Brynnier), Майку Холланду (Mike Holland), Энди Яо (Andy Yao), Лэрри Сэндерсу (Larry Sanders), Шаину Момтази (Shahin Momtazi) и Дону Сти-

венсу (Don Stephens) за то, что они потратили свое время и написали отзывы на первое издание. И наконец, благодаря Хейди Брумбаух (Heidi Brumbaugh) за ее придирчивое редактирование нового и переработанного материала.

По первому изданию

Много людей внесли свой вклад в создание этой книги. В частности, я хочу поблагодарить Митча Уэйта (Mitch Waite) за его участие в разработке, оформлении и переоформлении этой книги, а также за критические замечания по поводу рукописи. Выражаю свою признательность Гарри Хендерсону (Harry Henderson) за критические замечания по некоторым последним главам и за тестирование программного кода с помощью компилятора Zortech C++. Спасибо Дэвиду Джерольду (David Gerrold) за критические замечания по всей рукописи и за защиту интересов наименее опытных читателей. Не меньшей благодарности заслуживает также Хэнк Шифман (Hank Schiffman), который осуществил проверку программ с помощью компилятора Sun C++, и Кент Уильямс (Kent Williams), выполнивший проверку программ с помощью транслятора AT&T cfront и компилятора C++. Спасибо Нэн Борресон (Nan Borreson) из компании Borland International за то, что она охотно и содружественно помогала в работе с компиляторами Turbo C++ и Borland C++. Благодарю вас, Рут Майерс (Ruth Myers) и Кристина Буш (Christine Bush), за обработку нескончаемого бумажного потока, сопутствовавшего данному проекту. И наконец, спасибо Скотту Каламару за то, что он руководил всей работой.

Предисловие к третьему американскому изданию

Изучение языка C++ является приключением, ведущим к открытиям. Это связано, в частности, с тем, что язык C++ непрерывно развивается. Со времени выхода в свет второго издания данной книги и по настоящее время продолжалась эволюция языка C++, направляемая комитетом ISO/ANSI. Язык C++ также становился более зрелым по мере того, как программисты лучше осваивали его особенности. Но в настоящий момент имеется стандарт языка C++, и никаких крупных изменений в ближайшее время не ожидается. Таким образом, сейчас подходящее время, чтобы описать этот язык в его состоянии на текущий момент; это и является целью третьего издания книги *C++ Primer Plus*.

Вот основные изменения в содержании этой книги:

- В ней отражены изменения и дополнения, внесенные в язык C после выхода предыдущего издания книги, в том числе возросшая роль шаблонов и обобщенного программирования.

- В дополнительной главе рассматриваются класс `string` и стандартная библиотека шаблонов (Standard Template Library, сокращенно STL). Одна из главных особенностей языка C++ — это повторно используемые коды программ, а упомянутые выше библиотеки классов предоставляют полезные и эффективные примеры реализации таких программ.
- Некоторые разделы были переработаны с тем, чтобы еще больше упростить подачу материала и сделать его более понятным.

Как и в предыдущих изданиях, в этой книге рассказывается о стандартном языке C++, так что она не призвана к какому-нибудь одному виду компьютера, операционной системы или компилятора. Все программы были протестированы с помощью компиляторов CodeWarrior Pro 2 (в операционных средах Macintosh и Windows) и Microsoft Visual C++ 5.0, а при тестировании большинства из них применялись компиляторы Borland C++Builder 1.0, Symantic C++ 8.0, версия 5 (для ПК Macintosh), Watcom C++ 10.6 (для компьютеров, совместимых с IBM PC) и Gnu g++ 2.7.1, работающего под управлением Linux. Ни одна из этих реализаций не соответствовала стандарту полностью, но этого и следовало ожидать, так как они появились до того, как стандарт был принят. Язык C++ предоставляет в распоряжение программиста огромное богатство возможностей; желающим вам в полной мере воспользоваться этим богатством.

Предисловие ко второму американскому изданию

Изучить язык C++ непросто. Он не только обладает огромным набором программных возможностей, но и вводит такой стиль программирования (объектно-ориентированное программирование), который может потребовать пересмотра всех ваших представлений о программировании. Более того, для программирования на языке C++ необходимо знать правила употребления некоторых элементов, не являющихся стандартными для этого языка. Например, чтобы правильно использовать такое свойство языка, как наследование, программист должен знать не только соответствующие правила языка (для того, чтобы компилятор успешно скомпилировал программу), но и концептуальные правила относительно того, когда можно применять наследование, а когда нет. Кроме того, язык C++ непрерывно развивается, и со временем выхода в свет первого издания настоящей книги он прошел значительный путь в своей эволюции.

Эта книга нацелена на то, чтобы облегчить изучение языка C++ и даже сделать его приятным. Итак, в новом издании:

- Рассматриваются дополнения к языку C++, в частности, шаблоны, исключения, RTTI и области имен.
- Подчеркиваются изменения, внесенные в язык C++, например, описываются правила управления ссылочными аргументами.
- Представлен развивающийся стандарт ANSI/ISO языка C++.
- На более высоком теоретическом уровне даются указания по применению тех или иных особенностей языка, например, по использованию общедоступного наследования для моделирования того, что известно как отношения *is-a* (отношения типа “является”).
- Иллюстрируются широко распространенные способы построения смысловых программных конструкций языка C++, а также используемые в этом языке методы программирования.
- Чтобы читатель научился на практике применять новые идеи и понятия, в конце каждой главы приводятся упражнения по программированию.
- Больше внимания уделяется организации классов C++, для чего материал книги разделен на большее число глав, переработан и расширен.

В этом издании, так же как и в первом, описывается стандартный C++. Это означает, что его можно использовать с любой современной реализацией языка C++. Мы проверяли примеры с помощью разнообразных компиляторов, включая Borland C++ 3.1, Borland C++ 4.0, GNU C++ 2.0, Metrowerks CodeWarrior CW 3.5, Microsoft Visual C++ 1.0, Symantec C++ 6.0 (PC) и Symantec C++ 7.0 (Mac). В идеальном случае C++ все равно остается C++, но компиляторы различаются тем, в какой степени они соответствуют черновому стандарту. Например, для многих из этих компиляторов отсутствуют шаблоны или исключения; естественно, что с помощью таких компиляторов нельзя выполнять примеры, в которых используются эти свойства. Однако, помимо этого, мы столкнулись лишь с несколькими незначительными различиями, которые отмечены в настоящей книге. Вообще же, в настоящее время различные реализации языка C++ лучше согласованы друг с другом, чем во время выхода первого издания, и это хорошая новость для программистов.

Успехов вам в изучении языка C++!

Предисловие к первому американскому изданию

Когда в 1984 году Waite Group выпустила первое издание книги *C Primer Plus*, язык C существовал уже около десяти лет, но только в это время его известность стала расти. Мы гордимся той важной ролью, которую сыгра-

ла наша книга в популяризации языка С среди программистов. В настоящее время язык C++, который происходит от языка С, достиг аналогичной стадии в своей эволюции. Популярность этого языка бурно растет, так как он предлагает новую парадигму программирования — объектно-ориентированное программирование (ООП), которое полностью отвечает современным потребностям. Например, компания AT&T переписывает операционную систему UNIX с применением языка C++, так как язык C++ повышает степень надежности, облегчает сопровождение и повторное использование программ. По тем же самым причинам, а также потому, что ООП естественным образом подходит к таким особенностям программ, как поля и диалоговые окна, компания Apple в настоящее время разрабатывает программное обеспечение для своих компьютеров Macintosh на языке C++. Программисты переходят на язык C++, поскольку его новые особенности вновь делают программирование волнующим и увлекательным занятием. Естественно, сейчас наступило подходящее время для того, чтобы выпустить книгу *C++ Primer Plus* и поддержать этот бум.

Однако между тем и нынешним временем имеется одно различие: о языке C++ написано намного больше книг, чем было написано о языке С, когда он был еще новым. Но ни одна из новых книг по языку C++ не играет такой важной роли, как учебник для начинающих Waite Group. Во многих книгах, посвященных языку C++, предполагается, что вы уже знаете язык С, и знаете его хорошо. Вряд ли такие книги принесут пользу тем, кто хочет перейти на C++, скажем, с языка Pascal или BASIC, или для тех, для кого язык С был просто увлечением, и они не освоили его на профессиональном уровне. Во многих других книгах этой тематики описаны не только новые свойства языка, но и весь язык полностью. При этом опять же предполагается, что вы хорошо знаете язык С и владеете навыками программирования. Некоторые книги по языку C++ являются отличными справочниками, хотя изучать по ним язык было бы очень тяжело. Но есть и такие, которые нельзя назвать удачными. Это просто старые книги, дополненные несколькими новыми главами, причем либо новый материал интегрирован в состав книги не в полной мере, либо в них не уделяется должного внимания новым важным объектно-ориентированным особенностям языка C++.

Прочтите книгу *C++ Primer Plus* Waite Group. Мы не рассчитываем на то, что вы знаете язык С, поэтому изложение особенностей языка C++ идет параллельно с изложением основ языка С. Мы предполагаем, что у вас имеется хотя бы небольшой опыт программирования, но все-таки сочли нужным представить некоторые основы программирования. В общем и в целом мы стремились к тому, чтобы данная книга обладала всеми традиционными достоинствами учебников для начинающих, выпускаемых группой Waite Group:

- Учебник должен быть простым и удобным руководством.
- В учебнике не предполагается, что вы уже знакомы со всеми необходимыми понятиями программирования.
- Делается упор на практическое изучение с помощью кратких, легко вводимых в компьютер примеров программ, которые углубляют понимание одной или двух концепций языка.
- Концепции языка объясняются с помощью иллюстраций.
- Для проверки усвоения материала имеются упражнения, что делает учебник пригодным как для самостоятельного изучения, так и для занятий в аудитории (под руководством преподавателя).

В книге *C++ Primer Plus* представлены основы языка C++, которые иллюстрируются с помощью коротких программ, имеющих отношение к конкретным вопросам. Эти программы можно легко копировать, с ними можно легко экспериментировать. В данной книге не излагаются, как в энциклопедии, все особенности и нюансы языка C++, но в ней освещаются наиболее важные аспекты языка и в то же время закладывается фундамент для его дальнейшего изучения. Вы узнаете о различных способах обработки данных, о том, как применять функции, как осуществляется ввод и вывод данных, как в программах выполняются повторяющиеся действия и производится выбор (того или иного варианта действий). Вы узнаете о таких важных понятиях объектно-ориентированного программирования, как сокрытие информации (забавная вещь), полиморфизм и наследование. Помимо основных методов объектно-ориентированного программирования, вы изучите также его философию. Мы со своей стороны сделали все, что могли, чтобы изложение материала было кратким, простым и интересным. Наша цель состоит в том, чтобы к концу книги вы могли писать солидные и эффективные программы и получать от этого удовольствие.

Замечания для преподавателей

Одна из целей третьего издания заключается в том, чтобы эту книгу можно было использовать как для самостоятельного изучения, так и для аудиторных занятий с преподавателем. Вот некоторые особенности третьего издания книги *C++ Primer Plus*, характеризующие его как учебник для аудиторных занятий:

- В этой книге рассматривается стандартный язык C++, поэтому она не привязана ни к какой конкретной реализации этого языка.
- Материал основан на результатах работы комитета по стандартам ISO/ANSI языка C++; рассматриваются шаблоны, стандартная библиотека шаблонов, класс string, исключения, RTTI и области имен.

- Поскольку знание языка С не предполагается, эту книгу можно использовать на курсах, где язык С предварительно не изучается. (Однако желательно знание азов программирования.)
- Темы размещены таким образом, что на тех курсах, где предварительно изучается язык С, первые главы могут быть пройдены в ускоренном темпе, обзорно.
- В конце глав имеются вопросы для повторения и упражнения по программированию.
- В книге рассматривается несколько тем, которые подходят для курсов по основам вычислительной техники и программированию; речь идет, в частности, об абстрактных типах данных, стеках, очередях, простых списках, моделировании, обобщенном программировании и использовании метода рекурсии для реализации стратегии “разделяй-и-властвуй”.
- Многие главы являются достаточно короткими, и их можно пройти за неделю или даже быстрее.
- В книге рассматривается, когда и как можно использовать те или иные особенности языка. Например, общедоступное наследование связывается в ней с отношениями типа *is-a* (*является*), а композиция и приватное наследование — с отношениями типа *has-a* (*имеет*); в книге также объясняется, когда можно применять виртуальные функции, а когда нет.

Как организована эта книга

Данная книга состоит из 16 глав и 10 приложений. Ниже приводится их краткое описание.

Глава 1. Готовимся изучать язык C++

В данной главе описывается, как Бъярн Строуструп (Bjarne Stroustrup) создал язык программирования C++, добавляя к языку С элементы объектно-ориентированного программирования. Вы познакомитесь с различиями между процедурными языками, такими как С, и объектно-ориентированными, такими как C++. Вы прочтаете о совместной работе ANSI и ISO по разработке стандарта языка C++. В главе рассматривается процесс создания программы на языке C++ и схема работы нескольких современных компиляторов языка C++ в общих чертах. И наконец в ней приводятся соглашения, используемые в этой книге.

Глава 2. Приступаем к изучению языка C++

В главе 2 вы пройдете все этапы процесса создания простых программ на языке C++. Вы узнаете, какую роль играет функция `main()` и некоторые виды операторов, применяемых в программах C++. Для ввода и вывода данных вы будете использовать в программах предопределенные объекты `cout` и `cin`, а также узнаете, как создавать и использовать переменные. Кроме того, вы познакомитесь с функциями — программными модулями языка C++.

Глава 3. Представление данных

В языке C++ имеется несколько типов данных, используемых для представления данных двух видов: целых чисел (чисел без дробной части) и чисел с плавающей точкой (чисел с дробной частью). Для удовлетворения разнообразных требований программистов в языке C++ имеется несколько типов данных для каждой из этих двух категорий данных. В настоящей главе рассматриваются все эти типы данных, включая создание переменных и запись констант различных типов. Вы также узнаете, как в языке C++ осуществляются явные и неявные преобразования одних типов данных в другие.

Глава 4. Производные типы данных

На основе встроенных, базовых типов данных в языке C++ можно создавать более сложные типы данных. Наиболее сложной формой данных является класс, который рассматривается в главах 9–13. В настоящей главе описываются другие формы данных, включая массивы, которые содержат несколько значений одного типа; структуры, которые содержат несколько элементов различных типов; и указатели, которые идентифицируют области памяти. Вы узнаете, как создавать текстовые строки, запоминать их и как осуществлять ввод и вывод текстовых данных. Кроме того, будут рассмотрены некоторые способы распределения памяти в языке C++, включая операторы `new` и `delete`, применяемые для явного управления памятью.

Глава 5. Циклы и выражения сравнения

В программах часто приходится выполнять повторяющиеся действия, и для этого в языке C++ имеется три типа циклов: `for`, `while` и `do while`. Программа должна иметь возможность определять, когда такие циклы должны заканчиваться. Для этой цели в языке C++ имеются операторы сравнения, выполняющие проверку условий, с помощью которых осуществляется управление этими циклами. В этой главе рассказывается о том, как создавать циклы, позволяющие считывать и обрабатывать входные данные символ за символом, а также о том, как создавать двумерные массивы и обрабатывать их с помощью вложенных циклов.

Глава 6. Операторы ветвления и логические операции

Программа способна действовать разумно, если она может изменять свое поведение в зависимости от различных обстоятельств. В этой главе вы узнаете, как изменять ход выполнения программы с помощью операторов `if`, `if else` и `switch` и условной операции. Научитесь использовать логические операции для программирования текстов, определяющих принятие решений. Кроме того, вы познакомитесь с библиотекой `cctype`, в которой содержатся функции для определения отношений между символами, например, для проверки того, является ли символ цифрой или непечатаемым символом.

Глава 7. Функции языка C++

Функции — это основные “кирпичики”, из которых построена программа, написанная на языке C++. В данной главе внимание сконцентрировано на общих свойствах функций C++ и функций С. В частности, рассмотрен общий формат определения функции и показано, как прототипы функций повышают степень надежности программ. Вы поймете, как создаются функции для обработки массивов, строк символов и структур. Затем вы узнаете, что такое рекурсия (это когда функция вызывает саму себя) и как этот метод можно использовать для реализации стратегии “разделяй-и-властвуй”. И наконец, вы познакомитесь с указателями на функции, которые с помощью аргумента функции позволяют указывать функции, чтобы она использовала другую функцию.

Глава 8. Работа с функциями

В данной главе рассматриваются новые свойства, которые появляются у функций в языке C++. Рассматриваются встроенные функции, которые могут ускорить выполнение программы за счет увеличения ее размера. Вы сможете поработать со ссылочными переменными, которые предоставляют альтернативный способ передачи информации функции. Аргументы функции, используемые по умолчанию, дают возможность программе автоматически подставлять значения аргументов функции, которые не были указаны при вызове функции. Перегрузка функций позволяет создавать функции, имеющие одинаковые имена, но работающие с разными наборами аргументов. Все эти особенности часто используются при проектировании классов. Кроме того, вы познакомитесь с шаблонами функций, представляющими собой заготовки для семейств родственных функций, и научитесь создавать многофайловые программы. И наконец, рассмотрите классы памяти, области действия переменных, связывание и области имен, которые определяют, в каких частях программы известно о существовании переменной.

Глава 9. Объекты и классы

Класс — это определяемый пользователем тип данных, а объект является экземпляром класса, он подобен переменной. Настоящая глава знакомит вас с объектно-ориентированным программированием и проектированием классов. Объявление класса описывает информацию, хранимую в объекте класса, а также операции (методы класса), допустимые для этого объекта класса. Одна часть объекта видна извне (общедоступная часть), а другая скрыта от внешнего мира (приватная часть). Специальные методы класса (конструкторы и деструкторы) служат соответственно для создания и уничтожения объектов. В настоящей главе обо всех этих и других особенностях классов будет рассказано подробно. Кроме того, вы увидите как можно использовать клас-

сы для реализации абстрактных типов данных (АТД), таких как стеки.

Глава 10. Работа с классами

Настоящая глава поможет вам углубить свои знания о классах. Вы узнаете, что такая перегрузка операций, которая позволяет определять, каким образом такие операции, как +, будут выполняться над объектами классов. Познакомитесь с дружественными функциями, которые могут обеспечить доступ к тем данным класса, которые недоступны для внешнего мира в целом. Получите представление о том, как некоторые конструкторы и функции-элементы перегруженных операций могут использоваться для управления преобразованием одних классов в другие.

Глава 11. Классы и динамическое распределение памяти

Часто бывает полезно, чтобы элемент класса указывал на динамически выделяемую память. Если для распределения динамической памяти в конструкторе класса используется оператор new, то таким образом, вы возлагаете на себя ответственность за предоставление соответствующего деструктора, за определение явного конструктора копирования и явной операции присваивания. В настоящей главе показано, как это делается, кроме того, рассматривается поведение функций-элементов, генерируемых неявно, если вам не удалось создать явные определения. Вы также пополните свой опыт работы с классами путем использования указателей на объекты и изучения проблемы моделирования очередей.

Глава 12. Наследование классов

Одной из наиболее сильных особенностей объектно-ориентированного программирования является метод наследования, посредством которого производный класс наследует свойства базового класса, позволяя вам повторно использовать код базового класса. В этой главе рассматривается публичное наследование, которое моделирует отношения типа “является”, означающие, что производный объект является частным случаем базового объекта. Например, физик является частным случаем класса учёных. Реализация отношений типа “является” делает необходимым использование нового вида функций-элементов, которые называются виртуальными функциями. Все эти вопросы обсуждаются в настоящей главе, при этом указывается, когда можно применять метод публичного наследования, а когда нет.

Глава 13. Повторное использование программного кода в C++

Публичное наследование — это лишь один способ повторного использования кодов программ. В настоящей главе рассматривается несколько других способов. Включение — это когда один класс содержит элементы, ко-

торые являются объектами другого класса. Он может быть использован для моделирования отношений типа “имеет”, которые означают, что один класс имеет компоненты другого класса. Например, автомобиль имеет мотор. Для моделирования этих отношений можно также использовать методы приватного и защищенного наследования. В настоящей главе показано, как это делается, и объясняется, чем отличаются различные подходы. Помимо этого, вы познакомитесь с шаблонами классов, представляющими собой некие абстрактные родовые классы, на основе которых можно создавать конкретные классы. Например, шаблон стека позволяет создавать стек целых чисел или стек строк. И наконец, вы узнаете о множественном публичном наследовании, благодаря которому какой-либо класс может быть производным сразу от нескольких классов.

Глава 14. Дружественные конструкции, исключения и прочее

Настоящая глава продолжает тему дружественных конструкций, в ней рассматриваются дружественные классы и дружественные функции-элементы. Затем представлено несколько новых разработок в языке C++, начиная с исключений, которые предоставляют механизм для обработки в исключительных ситуациях в программе, таких как неправильные значения аргументов функций или выход за пределы памяти. Затем вы познакомитесь с RTTI (routine type information — информация о типе программы) — механизмом, служащим для идентификации типов объектов. И наконец, узнаете о более безопасных альтернативах неограниченному приведению типов.

Глава 15. Класс *string* и стандартная библиотека шаблонов

В этой главе описываются некоторые библиотеки классов, недавно добавленные в язык C++. Класс *string* — это удобная альтернатива традиционному использованию строк в языке С. Класс *auto_ptr* помогает управлять динамически распределяемой памятью. Стандартная библиотека шаблонов содержит несколько обобщенных контейнеров, включая шаблоны массивов, очередей, списков, наборов и карт. Она содержит также довольно эффективную библиотеку обобщенных алгоритмов, которые могут использоваться с контейнерами STL, а также с обычными массивами.

Глава 16. Ввод/вывод данных и работа с файлами

В этой главе рассматривается, как в языке C++ осуществляется ввод/вывод данных, а также форматирование выходных данных. Вы узнаете, как с помощью методов классов определять состояние входных и выходных потоков и, например, узнавать, не было ли обнаружено несоответствие типа входных данных или конец файла. С помощью метода наследования в языке C++ создаются

производные классы для управления вводом и выводом файлов. Вы получите представление о том, как открывать файлы для ввода и вывода, как присоединять к файлу данные, как применять двоичные файлы и как получить произвольный доступ к файлу. Кроме того, вы научитесь применять стандартные методы ввода/вывода для чтения данных из строк и для записи данных в строки.

Приложение A. Различные системы счисления

В этом приложении рассматриваются восьмеричные, шестнадцатиричные и двоичные числа.

Приложение B. Ключевые слова языка C++

В этом приложении перечислены ключевые слова языка C++.

Приложение C. Набор символов ASCII

В этом приложении приводится набор символов ASCII и соответствующие им десятичные, восьмеричные, шестнадцатиричные и двоичные значения (числа).

Приложение D. Приоритет операций

В приложении перечислены операции языка C++ в порядке понижения их приоритета.

Приложение E. Прочие операции языка C++

В этом приложении перечислены те операции языка C++ (например, поразрядные операции), которые не описаны в основной части книги.

Приложение F. Класс шаблона *string*

В этом приложении кратко описываются методы и функции класса *string*.

Приложение G. Методы и функции стандартной библиотеки шаблонов (STL)

В этом приложении кратко описываются методы контейнеров STL и общие функции алгоритмов STL.

Приложение H. Рекомендуемая литература

В этом приложении перечислены некоторые книги, которые помогут вам углубить свои знания в отношении языка C++.

Приложение I. Преобразование программного кода в соответствии со стандартом ANSI/ISO C++

В этом приложении даны рекомендации по переходу с языка С и более старых реализаций языка C++ на стандартный язык C++.

Приложение J. Ответы на вопросы для повторения

Это приложение содержит ответы на вопросы для повторения, расположенные в конце каждой главы.

Готовимся изучать язык C++

В этой главе рассматривается следующее:

- Как в язык С были добавлены понятия объектно-ориентированного программирования
- Как в язык С были добавлены понятия обобщенного программирования
- История и философия языка С
- Процедурное и объектно-ориентированное программирование
- История и философия языка C++
- Стандарты языка программирования
- Методика создания программ
- Соглашения, используемые в данной книге

Приглашаем вас к изучению языка C++! Этот замечательный язык, объединяющий в себе свойства языка С и объектно-ориентированного программирования, стал один из главных языков программирования в 90-е годы и твердо обещает оставаться таким в первом десятилетии XXI века. Язык C++ получил в наследство от языка С такие качества, как эффективность, компактность, быстрота выполнения и переносимость программ. От объектно-ориентированного программирования язык C++ получил новую методологию программирования, позволяющую справиться с возросшей сложностью современных задач программирования. А такие элементы языка, как улучшенные шаблоны, привносят в язык C++ еще одну новую методологию программирования: обобщенное программирование. Это тройное наследство является для языка C++ одновременно и благословением, и проклятием. Оно делают язык очень мощным, но в то же время и сложным; а это означает, что программистам приходится больше изучать.

В настоящей главе мы рассмотрим сначала общие вопросы, имеющие отношение к языку C++, а затем некоторые основные правила создания программ на языке C++. Далее в книге будет более подробно описан язык C++, начиная с его основ, а также объектно-ориентированного программирования (и сопутствующих ему новых терминов, таких как: объекты, классы, инкапсуляция, сокрытие данных, полиморфизм и наследование) и заканчивая обобщенным программированием.

(Безусловно, когда вы выучите язык C++, эти термины из непонятных слов превратятся в неотъемлемую часть языка грамотного общения специалистов.)

Изучение языка C++

В языке C++ соединены воедино три различных принципа программирования: процедурное программирование (представленное языком С), объектно-ориентированное программирование (представленное таким понятием, как класс, что повышает мощность языка C++ по сравнению с языком С) и обобщенное программирование (представленное шаблонами языка C++). Эти принципы кратко рассматриваются в настоящей главе. Но сначала давайте посмотрим, какое влияние оказывает это тройное наследство на методику изучения языка C++. Одна из причин применения языка C++ — это возможность использовать преимущества объектно-ориентированного программирования. Для этого необходимо прочное знание стандартного языка С, так как основные типы данных, операции, управляющие структуры и синтаксические правила были позаимствованы из этого языка. Поэтому, если вы уже знаете язык С, значит, вы готовы изучать C++. Но дело не ограничивается изучением нескольких новых слов или конструкций. Для перехода с языка С на C++ требуется почти столько же труда, сколько и для первоначального изучения самого языка С. Кроме того, если вы знаете язык С, то при переходе на язык C++ должны будете расстаться с неко-

торыми привычками, выработавшимися при программировании на языке С. Если вы не знаете язык С, то, чтобы изучить С++, вам придется овладеть компонентами языка С, компонентами ООП и компонентами обобщенного программирования; но при этом вам, по крайней мере, не придется избавляться от некоторых привычек. Если вы начинаете думать, что изучение языка С++ может потребовать от вас некоторого умственного напряжения, то вы правы. Настоящая книга делает процесс изучения языка С++ простым, понятным и постепенным, так что умственное напряжение будет незначительным и за состояние своей головы можете не беспокоиться.

В книге *Язык программирования С++. Лекции и упражнения* используется следующий подход к изучению языка С++: читатель учится не только работать с новыми компонентами языка С++, но и изучает его базис — язык С. Поэтому предварительное знание языка С не требуется. Изучение начинается с тех элементов языка, которые являются общими для языков С и С++. Даже если вы знаете язык С, данная часть книги может оказаться для вас полезным кратким курсом языка С. Кроме того, она особо обращает ваше внимание на те понятия языка, которые станут важными позже, а также на различия между языками С и С++. После того как вы твердо усвоите основы языка С, к ним добавится надстройка в виде новых элементов языка С++. Вы сможете составить полное представление о том, что такое объекты, классы и как они реализуются в языке С++.

Эта книга не является полным справочником по языку С++; в ней не рассматривается каждая мелкая деталь языка. Однако при этом описываются все основные элементы языка, включая такие нововведения, как шаблоны, исключения и области имен.

А теперь кратко рассмотрим некоторые вопросы, связанные с языком С++.

Немного истории

В последние несколько десятилетий компьютерная технология развивалась поразительными темпами. В настоящее время переносной компьютер может хранить больше информации и производить вычисления быстрее, чем большая ЭВМ тридцать лет назад. (Очень немногие программисты могут вспомнить, как приходилось носить большие колоды перфокарт, которые вводились в мощные, заполняющие целую комнату вычислительные системы с грандиозным (по тем временам) объемом памяти 100 Кб. Теперь этого объема недостаточно для запуска хорошей игры на персональном компьютере.) Языки программирования также претерпели значительную эволюцию. Изменения, возможно, не были такими впечатляющими, однако они были очень важными. Более мощные компьютеры порождали более крупные и сложные программы, которые, в свою очередь, поднимали новые

проблемы в области управления программами, а также их сопровождения.

В 70-е годы такие языки программирования, как С и Pascal, помогли войти в эру структурного программирования, принесшего порядок в ту область, которая сильно нуждалась в этом. Язык С предоставил в распоряжение программиста инструменты, необходимые для структурного программирования, а также обеспечил создание компактных, быстро работающих программ и возможность адресации аппаратных средств, например, возможность управления портами связи и накопителями на магнитных дисках. Эти качества помогли языку С стать господствующим языком программирования в 80-е годы. Вместе с тем в эти годы появилась новая модель программирования: объектно-ориентированное программирование, или ООП, воплощенное в таких языках, как SmallTalk и С++. Давайте рассмотрим теперь язык С и объектно-ориентированное программирование чуть подробнее.

Язык С

В начале 70-х годов Денис Ритчи (Dennis Ritchie) из компании Bell Laboratories занимался разработкой операционной системы UNIX. (Операционная система — это совокупность программ, управляющих ресурсами компьютера и его взаимодействием с пользователями. Например, именно операционная система выводит на экран системные сообщения-подсказки и управляет выполнением ваших программ.) Для выполнения этой работы Ритчи нуждался в таком языке программирования, который был бы кратким, а также мог бы обеспечивать эффективное управление аппаратными средствами и создание компактных, быстро работающих программ. Традиционно такие потребности программистов удовлетворял язык ассемблера, который тесно связан с внутренним машинным языком компьютера. Однако язык ассемблера — это язык *низкого уровня*, т.е. он привязан к определенному типу процессора (или компьютера). Поэтому если программу на языке ассемблера необходимо перенести на компьютер другого типа, то ее приходится переписывать заново на другом языке ассемблера. Это чуть-чуть похоже на то, как если бы при покупке нового автомобиля вы каждый раз обнаруживали, что конструкторы решили изменить расположение и назначение органов управления, вынуждая вас переучиваться вождению. Операционная система UNIX предназначалась для работы на разнообразных типах компьютеров (или платформах). А это предполагало использование языка высокого уровня. Язык *высокого уровня* ориентирован на решение задач, а не на конкретное аппаратное обеспечение. Специальные программы, которые называются *компиляторами*, транслируют программу на языке высокого уровня в программу на внутреннем языке конкретного компьютера. Таким образом, используя отдельный компилятор для каждой платфор-

мы, одну и ту же программу на языке высокого уровня можно выполнять на разных платформах. Ритчи нуждался в языке, который бы объединял эффективность и возможность доступа к аппаратным средствам, имеющиеся у языка низкого уровня, с более общим характером и переносимостью, присущими языку высокого уровня. Поэтому на основе имевшихся в то время более старых языков программирования Ритчи разработал язык С.

Философия программирования, заложенная в языке С

Поскольку язык C++ вносит в язык С новую философию программирования, нам следует сначала рассмотреть более старую философию языка С. Вообще, языки программирования имеют дело с двумя основными понятиями: данными и алгоритмами. Данные представляют собой информацию, которую программа обрабатывает. А алгоритмы — это методы, которые программа использует (для обработки данных) (рис. 1.1). Язык С, как и большинство основных языков программирования нашего времени, является *процедурным*. Это означает, что основное внимание в нем уделяется алгоритмам. Теоретически процедурное программирование заключается в том, что сначала определяется последовательность действий, которая должна быть выполнена компьютером, а затем эти действия реализуются с помощью языка программирования. Программа содержит набор процедур, которые компьютер должен выполнить, чтобы получить требуемый результат. Это во многом похоже на то, как кулинарный рецепт предписывает последовательность действий (процедур), следуя которым повар печет пирог.

По мере того как программы становились все больше, первые процедурные языки, такие как FORTRAN и BASIC, столкнулись с проблемами организационного плана. Например, в программах часто используются инструкции ветвления, которые направляют ход выполнения программы в сторону того или иного набора операторов в зависимости от результатов некоторой проверки. У многих старых программ такой запутанный ход выполнения (их называют "программы-спагетти"), что понять их, читая, по существу, невозможно, а модификация такой программы может привести к настоящей катастрофе. В ответ на это ученые- "компьютерщики" разработали более упорядоченный стиль программирования, который называется *структурное программирование*. Язык С включает ряд элементов, облегчающих применение структурного программирования. Например, структурное программирование ограничивает возможности ветвления (выбора следующего выполняемого оператора) небольшим набором хорошо функционирующих конструкций. Эти конструкции (циклы `for`, `while`, `do while` и оператор `if else`) входят в словарь языка С.

Еще одним из новых принципов программирования было проектирование программ *сверху вниз*.

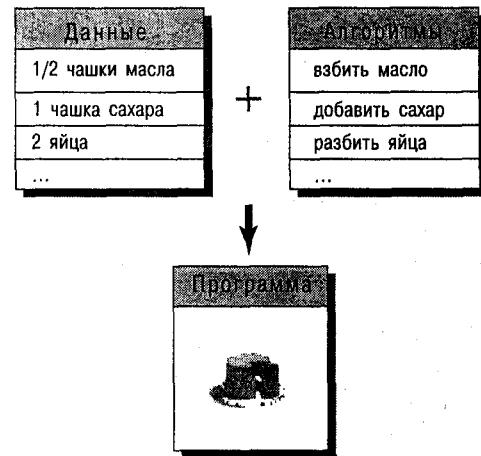


РИСУНОК 1.1 *Данные + алгоритм = программа.*

Идея заключается в разбиении большой программы на меньшие, легче программируемые задачи. Если одна из этих задач по-прежнему остается слишком обширной, разделите ее также на более мелкие задачи. Продолжайте этот процесс до тех пор, пока программа не будет разделена на маленькие, легко программируемые модули. (Наведите порядок в своем кабинете. Ой! Хорошо, наведите порядок в письменном столе, на столе и на своих книжных полках. Ох! Хорошо, начните с письменного стола и наведите порядок в каждом выдвижном ящике, начиная со среднего. Гм, пожалуй я могу справиться с этой задачей.) Язык С облегчает такой подход, поощряя программистов разрабатывать программные единицы (элементы), называемые *функциями*, которые представляют собой модули отдельных задач. Как вы могли заметить, методика структурного программирования отражает процедурный подход, при котором программа рассматривается с точки зрения выполняемых ею действий.

Объектно-ориентированное программирование

Хотя принципы структурного программирования улучшили понятность и надежность программ, а также облегчили их сопровождение, создание программ больших размеров по-прежнему оставалось трудной задачей. *Объектно-ориентированное программирование* (ООП) приносит с собой новый подход к решению этой задачи. В отличие от процедурного программирования, где главное внимание уделяется алгоритмам, в ООП основное внимание направлено на данные. При использовании ООП проблему не решают с помощью процедурного подхода, заложенного в языке, а приспособливают язык для решения этой проблемы. Идея заключается в создании таких форм данных, которые соответствовали бы основным характерным чертам проблемы.

В языке C++ *класс* является спецификацией, описывающей такую новую форму данных, а *объект* — конк-

ретной структурой данных, созданной в соответствии с этой спецификацией. Например, класс может описывать общие свойства руководящего работника корпорации (имя, должность, жалование и, например, необычные способности), тогда как объект представляет конкретного руководителя (Гилфорд Шипблэт (Guilford Sheepblat), вице-президент, оклад \$325 тыс. в год, знает как пользоваться файлом CONFIG.SYS). В общем, класс определяет, какие данные будут образовывать объект и какие операции могут выполняться над этими данными. Например, предположим, что вы разрабатываете графическую программу, способную рисовать прямоугольники. Можно создать класс, описывающий прямоугольник. Данными в спецификации этого класса может быть следующее: местоположение углов, высота и ширина, цвет и стиль ограничивающей линии, а также цвет и шаблон, используемые для заполнения прямоугольника. Часть спецификации этого класса, описывающая операции, может включать методы перемещения прямоугольника, изменения его размеров, вращения треугольника, изменения цветов и шаблонов, а также копирования прямоугольника в другое место. Если затем использовать эту программу, чтобы нарисовать прямоугольник, то она создаст объект в соответствии со спецификацией класса. Этот объект будет содержать все значения данных, описывающие прямоугольник, а с помощью методов класса можно будет этот прямоугольник модифицировать. Если необходимо нарисовать два прямоугольника, то программа создаст два объекта, по одному для каждого прямоугольника.

Объектно-ориентированный подход к разработке программы состоит в том, что сначала разрабатываются классы, точно представляющие те вещи, с которыми имеет дело программа. В графической программе, например, можно определить классы для представления прямоугольников, линий, окружностей, кистей, ручек и т.п. Вспомните, что определение класса включает описание допустимых операций для каждого класса, таких как перемещение окружности или вращение линии. После этого можно приступить к разработке самой программы, используя объекты этих классов. Этот процесс продвижения от более низкого уровня организации, такого как классы, к более высокому уровню, такому как программа, называется программированием *снизу вверх*.

Объектно-ориентированное программирование — это не только связывание данных и методов в единое целое: определение класса. ООП, например, облегчает создание повторно используемого кода программы, что в конечном итоге освобождает от большого объема работы. Скрытие информации позволяет предохранить данные от нежелательного доступа. Полиморфизм дает возможность создавать множественные определения для операций и функций, а то, какое определение будет использоваться, зависит от контекста программы. Наследование позволяет создавать новые классы из старых. Как види-

те, с объектно-ориентированным программированием позволяет много новых идей и используется иной подход к созданию программ, чем при процедурном программировании. Вместо того чтобы сосредоточить свое внимание на задачах, вы фокусируете его на представлении различных понятий. Вместо того чтобы использовать программирование "сверх-вниз", вам иногда придется использовать программирование "снизу-вверх". В данной книге рассматриваются все эти вопросы вместе со множеством наглядных примеров.

Разработка полезного и надежного класса может быть трудной задачей. К счастью, объектно-ориентированные языки дают возможность без особого труда включать существующие классы в создаваемые программы. Поставщики программного обеспечения разработали различные библиотеки классов, включая библиотеки классов, предназначенные для упрощения создания программ в таких средах, как Windows или Macintosh. Одним из реальных преимуществ языка C++ является то, что он позволяет легко адаптировать и повторно использовать хорошо проверенные коды программ.

Обобщенное программирование

Обобщенное программирование — это еще одна парадигма программирования, поддерживаемая языком C++. Оно имеет общую с ООП цель — упростить повторное использование кодов программ и методов абстрагирования общих понятий. Однако, в то время как в ООП основное внимание уделяется данным, в обобщенном программировании упор делается на алгоритмы. И у него другая область применения. ООП — это инструмент для разработки больших проектов, тогда как обобщенное программирование предоставляет инструменты для выполнения задач общего характера, таких как сортировка данных или объединение списков. Термин *обобщенный* означает создание кода программы, независимого от типа данных. В языке C++ имеются данные различных типов — целые числа, числа с дробной частью, символы, строки символов, определяемые пользователем сложные структуры, состоящие из данных нескольких типов. Если, например, требуется сортировать данные различных типов, то обычно для каждого типа создается отдельная функция сортировки. Обобщенное программирование расширяет язык таким образом, что позволяет один раз написать функцию для обобщенного (т.е. неопределенного) типа данных и затем использовать ее для разнообразных реальных типов данных. Это обеспечивается с помощью шаблонов языка C++.

Язык C++

Язык C++, так же как и язык C, является детищем компании Bell Laboratories. Как уже говорилось, Бэрни Страуструп разработал этот язык в начале 80-х годов. По его собственным словам, "язык C++ был спроектирован главным образом так, чтобы мне и моим друзьям не

приходилось программировать на ассемблере, С или различных современных языках высокого уровня. Его главная цель состояла в следующем: сделать так, чтобы отдельным программистам было легче и приятнее писать хорошие программы" (Bjarne Stroustrup, *The C++ Programming Language*. Third Edition. Reading, MA: Addison-Wesley Publishing Company, 1997).

Страуструп был больше озабочен тем, чтобы язык C++ был полезным, а не являлся носителем какой-либо философии или стиля программирования. Реальные потребности программирования более важны, чем теоретическая чистота определения свойств языка. Страуструп создал C++ на основе языка С, так как язык С был кратким, хорошо подходил для системного программирования, был широко доступен и тесно связан с операционной системой UNIX. Объектно-ориентированная часть языка C++ возникла под влиянием языка моделирования Simula67. Страуструп добавил элементы ООП в язык С, не изменяя при этом существенно сам язык С. Таким образом, язык C++ является расширением языка С, а это означает, что любая правильная программа С является также правильной программой C++. Имеются лишь некоторые незначительные различия, но они не столь существенны. Программы C++ могут использовать существующие библиотеки языка С. Библиотеки — это совокупности программных модулей, которые могут вызываться из программ. Они предоставляют готовые решения многих широко распространенных задач программирования, экономя таким образом много времени и усилий. Это помогло распространению языка C++.

Название C++ происходит от обозначения оператора инкремента ++ в языке С, который добавляет единицу к значению переменной. Название C++ подразумевает, что этот язык является усовершенствованной (++) версией языка С.

Задачу, взятую из реальной жизни, компьютерная программа переводит в последовательность действий, которая должна быть выполнена компьютером. Компонент ООП в языке C++ дает возможность устанавливать связи между понятиями, относящимися к данной проблеме, а элементы языка С обеспечивают более тесное взаимодействие с аппаратными средствами (рис. 1.2). Эта комбинация возможностей помогла распространению языка C++. Она также может привести к необходимости изменять методику программирования при переходе от одного аспекта программы к другому. (В самом деле, некоторые сторонники чистого ООП считают, что добавлять элементы ООП к языку С — все равно, что придавать крылья свинье, пусть даже продуктивной и симпатичной.) Кроме того, поскольку C++ является языком С, к которому добавили элементы ООП, можно просто игнорировать эти новые возможности. Но если вы так сделаете, то многое потеряете.

Только после того, как язык C++ получил некоторое признание, Страуструп добавил в него шаблоны,

Элементы ООП обеспечивают высокий уровень абстрагирования



...
north_america.show();
....

Элементы языка С обеспечивают доступ к аппаратным средствам на низком уровне



set byte at
address
01000 to 0

РИСУНОК 1.2 Двойственность языка C++.

обеспечивая тем самым возможность обобщенного программирования. И только после того, как шаблоны были использованы на практике и усовершенствованы, он стал понимать, что они имеют такое же значение, как и ООП, или даже большее. Тот факт, что язык C++ включает в себя как ООП, так и обобщенное программирование, показывает, что в C++ упор делается на утилитарный, а не идеологический подход, и это одна из причин успеха этого языка.

Переносимость и стандарты

Предположим, что на работе вы написали удобную программу для старого компьютера 286 РС АТ, и тут руководство решило заменить этот компьютер на рабочую станцию Sun — компьютер с другим процессором и другой операционной системой. Сможете ли вы выполнять свою программу на новой платформе? Конечно, вам придется повторно скомпилировать программу с помощью компилятора, разработанного для новой платформы. Но придется ли вам делать изменения в написанном коде программы? Если вы можете перекомпилировать программу, не внося в нее изменений, и выполнить ее без сучка и задоринки, значит, эта программа является *переносимой*.

Существует два препятствия, мешающих переносимости программ, одним из которых является аппаратное обеспечение. Программа, в которой используются особенности аппаратных средств, вряд ли будет переносимой. Например, программа, непосредственно управляющая видеокартой VGA на компьютере IBM PC, будет отображать бессмыслицу при выполнении на рабочей станции Sun. (Проблемы, связанные с переносимостью, можно минимизировать, размещая аппаратно зависимые части программы в модулях функций; тогда вам придется разработать только эти конкретные модули.) В настоящей книге этот вопрос не рассматривается.

Вторым препятствием для переносимости является дивергенция языка (различные отклонения в языке). Конечно, в обычных языках это может быть проблемой. Описание событий дня, сделанное жителем Йоркшира, может быть "непереносимым" для жителя Бруклина, хотя и тут, и там говорят по-английски. В языках программирования также могут развиваться диалекты. Является ли реализация языка C++ для компьютеров IBM PC такой же самой, как и реализация для рабочих станций Sun? Многие разработчики реализаций языка C++ хотели бы, чтобы их версии были совместимыми с другими, но без опубликованного стандарта, точно описывавшего язык, сделать это трудно. Поэтому Американский институт национальных стандартов (American National Standards Institute, сокращенно ANSI) в 1990 году создал комитет (ANSI X3J16) для разработки стандарта языка C++. (ANSI уже разработал стандарт языка C.) Международная организация стандартов (International Standards Organization, сокращенно ISO) вскоре присоединилась к этому процессу со своим собственным комитетом (ISO-WG-21). Эти комитеты, ANSI и ISO, заседают совместно три раза в год, и мы будем просто называть их одним комитетом ANSI/ISO. В своем решении о создании стандарта языка C++ комитет ANSI/ISO подчеркивает, что C++ стал широко распространенным языком, имеющим важное значение. В нем также указывается, что C++ достиг определенного уровня зрелости, поскольку нецелесообразно вводить стандарты в то время, когда язык быстро развивается. Тем не менее, язык C++ претерпел значительные изменения со времени начала работы комитета.

Работа комитета ANSI/ISO над стандартом языка C++ началась в 1990 году. В последующие годы комитет выпустил несколько рабочих документов, носящих временный характер. В апреле 1995 года он выпустил черновой вариант стандарта (Committee Draft, сокращенно CD) для широкого обсуждения. В декабре 1996 года он выпустил вторую версию чернового варианта (CD2) для дальнейшего обсуждения общественностью. В этих документах было не только доработано и слажено описание существующих элементов языка C++, но язык был также расширен (дополнен) исключениями, RTTI, шаблонами и стандартной библиотекой шаблонов (Standard Template Library). Временные рабочие документы и черновые варианты комитета часто называют "черновым стандартом", но первым документом, который был озаглавлен так официально, является Окончательный черновой международный стандарт (Final Draft International Standard, сокращенно EDIS), выпущенный в ноябре 1997 года. Окончательный Международный стандарт (IS — International Standard) был утвержден в июне 1998 года. Настоящая книга придерживается главным образом чернового варианта CD2 и очень похожего на него чернового стандарта FDIS. В свою очередь, различия между стандартами FDIS и IS являются весьма незначительны-

ми и представляют собой небольшие изменения, а не замену существующих элементов языка или добавление новых.

Кроме того, стандарт ANSI/ISO C++ придерживается стандарта ANSI C, так как предполагается, что язык C++, насколько это возможно, является расширением языка C. В идеале это означает, что любая работоспособная программа C должна быть также работоспособной программой C++. Между стандартом ANSI C и соответствующими правилами языка C++ имеется ряд различий, но они незначительные. На самом деле, стандарт ANSI C содержит некоторые элементы, впервые появившиеся в языке C++, например, прототипы функций и спецификаторы типа `const`.

До появления стандарта ANSI C те, кто имел отношение к языку C, руководствовались книгой *The C Programming Language* Кернигана и Ритчи (Addison-Wesley Publishing Company Reading, MA. 1978), которая являлась стандартом "де facto". Этот стандарт часто называют стандартом K&R C; с появлением стандарта ANSI C более простой стандарт K&R C стали иногда называть классическим языком C.

Стандарт ANSI C определяет не только язык C, но и стандартную библиотеку C, которая должна быть включена во все реализации языка C, соответствующие стандарту ANSI C. В языке C++ также используется эта библиотека; в настоящей книге она называется стандартной библиотекой C или просто стандартной библиотекой. В дополнение к этому комитет ANSI/ISO должен утвердить стандартную библиотеку классов языка C++.

До того как комитет ANSI/ISO начал свою работу, многие рассматривали как стандарт языка C++ самую последнюю версию языка, выпущенную компанией Bell Labs. Например, в описании компилятора может указываться, что он совместим с версией 2.0 или 3.0 языка C++.

Прежде чем мы возьмемся за язык C++ по-настоящему, давайте обсудим некоторые базовые вопросы, связанные с созданием программ; кроме того, в конце главы кратко объясняется, как пользоваться настоящей книгой.

Методика создания программ

Предположим, что вы написали программу на языке C++. Как ее выполнить? Конкретные действия зависят от программной среды на вашем компьютере и от используемого компилятора C++. Но в общем вам будет необходимо выполнить следующее (рис. 1.3):

- Использовать какой-нибудь текстовый редактор для ввода программы в компьютер и сохранения ее в виде файла. Это будет *исходный код* вашей программы.
- Скомпилировать исходный код. Это означает выполнение программы, которая транслирует исходный код во внутренний язык компьютера, называемый *машинным языком*. Файл, содержащий оттранслиро-

ванную программу, является *объектным кодом* программы.

- Связать объектный код с дополнительным кодом и скомпоновать из них единую программу. Например, программы C++ обычно используют *библиотеки*. Библиотека C++ содержит совокупность объектных кодов компьютерных программ (подпрограмм), называемых *функциями*, которые служат для выполнения таких задач, как отображение информации на экране или вычисление квадратного корня числа. При компоновке объектный код программы объединяется с объектными кодами функций, используемых программой, и определенным стандартным кодом начальной загрузки, в результате чего создается выполняемая версия программы. Этот файл, содержащий окончательный продукт, называется *исполняемым кодом*.

На протяжении всей книги вы будете постоянно сталкиваться с термином *исходный код*, так что запомните его хорошенько в своей "личной памяти с произвольным доступом".

Программы в этой книге являются обобщенными и должны выполняться в любой системе с реализацией языка C++. (Однако ко времени написания этой книги во многих компиляторах присутствовали не все элементы языка.)

Например, только в некоторых компиляторах имелись области имен и новейшие элементы шаблонов.) Действия по получению исполняемого кода программы могут различаться. Давайте рассмотрим эти действия немного подробнее.

Создание исходного кода

Некоторые реализации языка C++, такие как Microsoft Visual C++, Borland C++ (различные версии), Watcom C++, Symantec C++ и Metrowerks CodeWarrior реализованы в виде *интегрированных сред разработки* (*integrated development environments*, сокращенно IDE), которые позволяют вам выполнять все действия по разработке программы, включая редактирование, из одной главной программы. Другие реализации, такие как AT&T C++ или GNU C++ в операционных системах UNIX и Linux, позволяют выполнять только стадии компиляции и компоновки, а команды в этих реализациях вводятся в командной строке системы. В таких случаях для создания и модификации исходного кода можно использовать любой доступный текстовый редактор. Например, в операционной системе UNIX можно использовать редакторы vi, ed, ex или emacs. В DOS можно использовать редакторы edlin, edit или любой другой из доступных редакторов. Можно даже использовать текстовые процессоры при условии, что полученный файл сохраняется в формате стандартного текстового файла DOS, а не в каком-нибудь специальном формате текстового процессора.

Присваивая имя исходному файлу, необходимо использовать соответствующий суффикс, чтобы идентифицировать этот файл как файл исходного кода на языке C++. Этот суффикс информирует не только программистов, но и компилятор о том, что данный файл представляет собой исходный код программы C++. (Если в операционной системе UNIX компилятор выдает сообщение о "неправильном волшебном числе" ("bad magic number"), то это просто любезный способ информирования о том, что использован неверный суффикс.) Суффикс состоит из точки, за которой следует символ или группа символов, называемая расширением (рис. 1.4).

В различных реализациях языка C++ используются разные расширения. В табл. 1.1 приведены некоторые распространенные расширения. Например, spiffy.C – это правильное имя файла исходного кода в реализации AT&T C++. Обратите внимание на то, что в операционной системе UNIX символы верхнего и нижнего регистра различаются; это означает, что в данном случае необходимо использовать символ С верхнего регистра. Расширение c (символ нижнего регистра) также имеется в этой реализации, но оно соответствует исходному коду на стандартном языке С. Поэтому, чтобы избежать путаницы, в операционной системе UNIX расширение c следует использовать для программ С, а расширение C – для программ C++. Если вы не имеете ничего против одного или двух дополнительных символов, то в некоторых системах UNIX можно также использовать расширения cc и cxx.

ОС DOS устроена проще, чем UNIX, и символы верхнего и нижнего регистров в этой системе не различа-

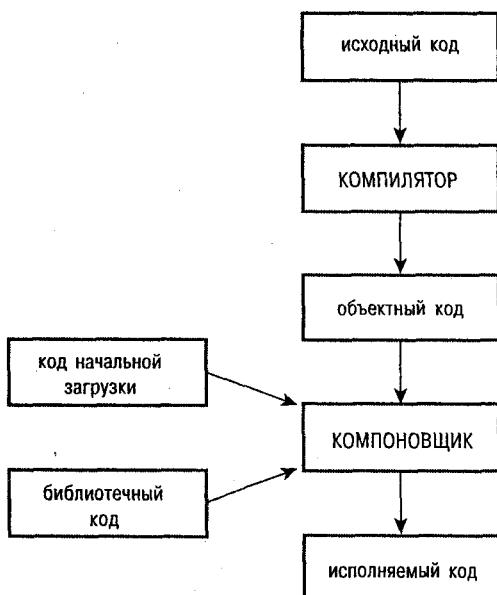


РИСУНОК 1.3 Этапы создания исполняемого кода программы.

ются. Поэтому, как показано в табл. 1.1, чтобы различать программы С и С++, в реализациях С++ для DOS используются дополнительные буквы.

Таблица 1.1 Расширения исходного кода.

Реализация С++	Расширения исходного кода
UNIX AT&T	C, cc, cxx, c
GNU C++	C, cc, cxx, c
Symantec	cpp, cp
Borland C++	cpp
Watcom	cpp
Microsoft Visual C++	cpp, cxx
Metrowerks CodeWarrior	cp, cpp

Компиляция и компоновка

Первоначально Страуструп реализовал язык С++⁶ не в виде компилятора, преобразующего исходный код программы С++ в объектный код, а в виде компилирующей программы, преобразующей программу С++ в программу С. Эта программа, которая называлась *cfront* (for C front end), транслировала исходный код С++ в исходный код С, который мог затем компилироваться стандартным компилятором С. Этот подход упростил работу с языком С++ тем, кто уже работал с языком С. В различных реализациях для других платформ также использовался этот подход. Когда язык С++ стал развитым языком и его популярность выросла, все больше и больше разработчиков стали создавать компиляторы С++, которые создавали объектный код непосредственно из исходного кода С++. Такой подход ускоряет процесс компиляции и подчеркивает тот факт, что С++ является отдельным языком программирования, хотя и похож на язык С.

Часто различия между транслятором *cfront* и компилятором почти невидимы для пользователя.

Например, в операционной системе UNIX с помощью команды **CC** программа может сначала обрабатываться транслятором *cfront*, а затем выходные данные транслятора автоматически передаются компилятору С, который называется **cc**.

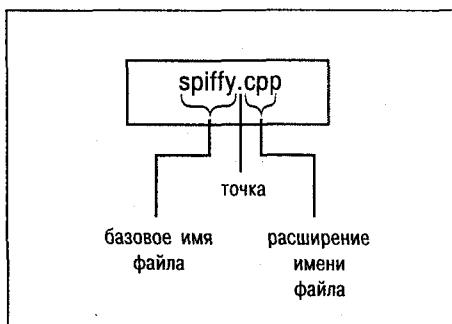


РИСУНОК 1.4 Расширение исходного файла.

Для подобной комбинации транслятора и компилятора мы будем далее использовать термин "компилятор". Методика компилирования зависит от реализации языка, и в последующих разделах описываются несколько широко распространенных видов компиляторов. В этих кратких описаниях рассматриваются основные действия по компиляции программы, но это не заменяет документацию по используемой системе.

Компиляция и компоновка в операционной системе UNIX

Предположим, например, что вы работаете с реализацией AT&T Release 3.0 С++ в операционной системе UNIX. Чтобы скомпилировать программу, используйте команду **CC**. Чтобы это имя отличалось от имени компилятора **cc** (стандартного компилятора С), оно составлено из буквы верхнего регистра. Компилятор **CC** является компилятором командной строки; это означает, что команды компиляции вводятся на командной строке операционной системы UNIX. Например, чтобы скомпилировать файл исходного кода *spiffy.C*, необходимо в командной строке UNIX ввести следующую команду:

```
CC spiffy.C
```

Если благодаря вашим знаниям, опыту или просто везению программа не содержит ошибок, компилятор создает файл объектного кода с расширением **o**. В данном случае компилятор создаст файл *spiffy.o*.

Затем компилятор автоматически передает файл объектного кода системному компоновщику — программе, которая объединяет этот код с библиотечным кодом и создает исполняемый файл. По умолчанию исполняемый файл называется **a.out**. Если был только один исходный файл, то компоновщик, помимо всего прочего, удаляет файл *spiffy.o*, так как он больше не нужен. Чтобы выполнить эту программу, просто введите имя исполняемого файла:

```
a.out
```

Обратите внимание на то, что если скомпилировать новую программу, то новый исполняемый файл **a.out** заменяет предыдущий файл **a.out**. (Это связано с тем, что исполняемые файлы занимают много места. Поэтому запись нового исполняемого файла на место старого файла позволяет снизить требования к памяти.) Но если вы создаете исполняемую программу, которую требуется сохранить, просто используйте команду **mv**, чтобы изменить имя исполняемого файла.

Программа С++, так же как и программа С, может состоять из нескольких файлов. (Таковыми являются многие программы в главах 8–16 настоящей книги.) В этом случае программу можно скомпилировать, указав в командной строке имена всех этих файлов:

```
CC my.C precious.C
```

Если имеется несколько файлов исходного кода, компилятор не удаляет файлы объектного кода. Тогда, если вы изменяете только файл **my.C**, то можете перекомпилировать эту программу с помощью следующей команды:

```
cc my.C precious.o
```

В результате выполнения этой команды файл **my.C** повторно компилируется и связывается с ранее скомпилированным файлом **precious.o**. Некоторые библиотеки вам, возможно, придется идентифицировать явно. Например, для доступа к функциям математической библиотеки вам, возможно, придется добавить в командной строке флагок **-lm**.

```
cc usingmath.C -lm
```

Компания Free Software Foundation поставляет с реализацией GNU C++ компилятор **g++**, который функционирует во многом подобно стандартному компилятору системы UNIX:

```
g++ spiffy.C
```

В некоторых версиях может потребоваться указать библиотеку C++:

```
g++ spiffy.C -lg++
```

Компилятор **g++** реализации GNU C++ выпускается для многих платформ, включая операционную систему Linux, работающую на компьютерах, совместимых с IBM PC.

Реализации Turbo C++ 2.0 и Borland C++ 3.1 (DOS)

В версиях интегрированных сред Turbo C++ и Borland C++ для DOS, включающих встроенный редактор, для выполнения требуемых действий используется панель меню, доступ к которой осуществляется посредством мыши или путем сочетания некоторых клавиш с клавишей Alt. Например, меню File (Файл) позволяет создавать, сохранять и открывать файлы. Меню Edit (Правка) помогает редактировать исходный файл. Меню Compile (Компилировать) дает возможность выбирать разные режимы компиляции, а меню Run (Выполнить) — разные режимы выполнения программ. После того как с помощью встроенного редактора вы ввели текст программы, проще всего выбрать команду Run из меню Run. В результате выполнения этой команды осуществляется компиляция, компоновка и выполнение программы. Если компилятор обнаружит ошибки, то он, конечно, не будет выполнять программу, а отобразит на экране список ошибок и выделит в исходном коде строки с ошибками. Кроме того, интегрированная среда включает в себя отладчик, который позволяет выполнять программу в пошаговом режиме и наблюдать, как изменяются данные.

Если вы разрабатываете программу, состоящую из нескольких файлов исходного кода, то откройте новый проект с помощью команд меню Project (Проект). Затем можно использовать другие команды меню, чтобы добав-

лять в список проекта соответствующие файлы. Файл проекта дает возможность интегрированной среде Borland C++ или Turbo C++ отслеживать все, что происходит с проектом. Если вы изменяете один из файлов в списке проекта, то компилятор обновляет исполняемую программу. В интегрированной среде Borland C++ или Turbo C++ несколько файлов исходного кода могут быть открыты одновременно, причем каждый файл открывается в своем собственном окне и можно легко переключаться от одного файла к другому.

Интегрированные среды Borland C++ и Turbo C++ поставляются с обучающими программами, в которых описаны все элементы этих сред. И конечно же вы можете читать соответствующие учебники.

Компиляторы для операционной системы Windows

Компиляторов для Windows так много, и они так часто обновляются, что было бы неразумно описывать их все по отдельности. Однако у них есть ряд общих черт.

Как правило, необходимо создавать проект программы и добавлять в него один или несколько файлов, составляющих программу. Все поставщики выпускают интегрированные среды разработки с набором меню и, возможно, с программой автоматизированного помощника, которая очень полезна при создании проекта. Вам необходимо решить один очень важный вопрос: какого вида программу вы создаете. Как правило, компилятор допускает много вариантов выбора, таких как приложение Windows, приложение Windows MFC, библиотека динамических связей, элемент управления Active X, исполняемый файл DOS, статическая библиотека или консольное приложение. Для некоторых из них имеются 16- и 32-разрядные версии.

Поскольку программы в этой книге являются обобщенными, вам не следует выбирать варианты, приводящие к созданию платформенно-зависимых кодов программ, например, приложение Windows. Вместо этого нужно работать в символьном режиме. Выбор требуемого режима будет зависеть от компилятора. В одних версиях фирмы Microsoft имеется режим QuickWin, который эмулирует сеанс DOS; в других версиях имеется режим Console. Одни версии Borland включают режим EasyWin, который эмулирует сеанс DOS; в других версиях предлагается режим Console. В компиляторах Metrowerks предлагается вариант Console. Вообще, ищите в меню такие названия, как Console, character-mode или DOS executable, и попробуйте выбрать их. После того как проект будет создан, вам придется компилировать и компоновать свою программу. Для этого в интегрированной среде разработки (IDE) предлагается, как правило, несколько вариантов, например: Compile, Build, Make, Build All, Link, Execute и Run (но не обязательно все эти наименования имеются в одной IDE!).

- *Compile* обычно означает: компилировать код программы в том файле, который открыт в настоящее время.

- *Build or Make* обычно означает: компилировать коды программ во всех файлах исходного кода, входящих в данный проект. Этот процесс носит, как правило, инкрементный характер, т.е. если проект имеет три файла и вы изменили только один, то перекомпилироваться будет один этот файл.
- *Build All* обычно означает: компилировать с самого начала все файлы исходного кода.
- *Link* означает (как описывалось ранее): объединить скомпилированный исходный код с необходимым библиотечным кодом.
- *Run* или *Execute* означает: выполнить программу. Если при этом вы еще не выполнили более ранние действия, то команда Run перед выполнением программы позволит сделать и это.

Когда вы нарушаете правила языка, компилятор выдает сообщение об ошибке и идентифицирует строку с ошибкой. К сожалению, новичку может быть трудно понять данное сообщение. Иногда может оказаться, что на самом деле ошибка произошла не в той строке, которая указана как ошибочная, а в предыдущей. А иногда одна ошибка может породить целую цепочку сообщений об ошибках.

СОВЕТ

Всегда сначала устранийте первую ошибку. Если вы не можете найти ее в строке, указанной как ошибочная, то проверьте предыдущую строку.

СОВЕТ

Иногда компиляторы, не закончив создание программы, выдают бессмысленные сообщения об ошибках, которые нельзя устраниить. В таких случаях ситуацию можно прояснить, выбирая команду *Build All*, чтобы начать весь процесс компиляции с самого начала. К сожалению, эту ситуацию трудно отличить от другой, более распространенной ситуации, когда сообщение об ошибке лишь только кажется бессмысленным.

В IDE программа обычно выполняется во вспомогательном окне. Когда программа завершает свое выполнение, в некоторых IDE это окно сразу закрывается, а в некоторых остается открытым. Если компилятор закрывает данное окно, то у вас не будет достаточно времени, чтобы прочитать выходные данные программы, разве что вы умеете очень быстро читать и у вас феноменальная память. Чтобы увидеть результат, в конце программы необходимо поместить дополнительный код:

```
cin.get(); // добавить этот оператор
cin.get(); // и этот, возможно, тоже
return 0;
}
```

Оператор *cin.get()* считывает информацию с клавиатуры, поэтому его действие приводит к тому, что программа ожидает нажатия клавиши Enter. (Программа не

получит информацию ни от одной нажатой клавиши до тех пор, пока не будет нажата клавиша Enter, так что не пробуйте нажимать какую-нибудь другую клавишу.) Второй оператор необходим на тот случай, если программа оставила необработанным нажатие клавиши после предыдущего ввода информации в программе. Например, при вводе числа вы нажимаете на соответствующую клавишу, а затем на клавишу Enter. Программа может считать число, но оставить необработанным нажатие клавиши Enter. Тогда оно будет обработано первым оператором *cin.get()*.

Компилятор Borland C++Builder конструктивно немного отличается от других, более традиционных компиляторов. Он предназначен для программирования в операционной системе Windows. Для создания с его помощью обобщенных программ в меню File (Файл) выберите команду New (Создать). Затем выберите команду Console App (Консоль приложения). Откроется окно, содержащее макет функции *main()*. Некоторые элементы можно удалить, но две следующие нестандартные строки необходимо оставить:

```
#include <vc1\condefs.h>
#pragma hdrstop
```

Компиляторы для компьютеров Macintosh

Двумя наиболее известными компиляторами для реализации Macintosh C++ являются Metrowerks Code Warriort и Symantec C++ для компьютеров Macintosh. Оба представляют собой интегрированные среды разработки (IDE), главным рабочим элементом которых является проект. В основных чертах эти IDE подобны компиляторам для операционной системы Windows. (Действительно, обе компании предлагают версии своих компиляторов, предназначенные для Windows.) Работу в любом из этих программных продуктов начинайте с выбора команды New Project в меню File. Вам будет предложено на выбор несколько типов проектов. В более ранних версиях Code Warriort выберите MacOS:C/C++:ANSI C++ Console, а в более поздних — MacOS:C/C++:Standard Console:Std C++ Console; В компиляторе Symantec выберите ANSI C++ (iostreams). Возможно, вам также придется выбирать между версией 68K (для процессоров серии Motorola 680X0) или PPC (для процессоров Power PC).

В обоих программных продуктах новый проект содержит в качестве составной части маленький файл исходного кода. Можно попробовать скомпилировать и выполнить эту программу, чтобы посмотреть, соответствующим ли образом настроена ваша система. Однако если вы вводите свой код программы, то предварительно следует удалить из проекта этот файл. Для этого выделите файл в окне проекта и затем выберите команду Remove (Удалить) в меню Project (Проект).

Затем необходимо добавить в проект свой исходный код. Для этого в меню File можно выбрать команду New

(чтобы создать новый файл) или команду Open (чтобы открыть существующий файл). Используйте соответствующий суффикс, например, `ср` или `cpr`. Добавьте этот файл в список проекта с помощью меню Project. Для некоторых программ из этой книги потребуется вводить несколько файлов исходного кода. Когда закончите ввод, выберите команду Run (Выполнить) в меню Project (Проект).



СОВЕТ

Вы сможете лучше освоить материал, если будете выполнять приводимые примеры и экспериментировать с ними.

Между прочим, вы только что прочитали реальный и важный совет, а не просто пример того, как выглядит в книге правило или совет.

И наконец, когда вы вводите в программу данные, то в конце обычно нажимаете клавишу Return или Enter. На разных клавиатурах используется либо одно, либо другое название. В настоящей книге эта клавиша называется Enter.

Наша система

В данной книге описывается черновой вариант стандарта ISO/ANSI CD2 языка C++, который применительно к материалу данной книги не отличается, по существу, от окончательного стандарта. Поэтому примеры программ из этой книги должны работать в любой реализации языка C++, совместимой с данным стандартом. (По крайней мере, так должно быть с точки зрения переносимости, и мы на это надеемся.) Однако стандарт языка C++ появился недавно, и вы можете найти в компиляторах несколько расхождений со стандартом. Например, к моменту написания данной книги во многих компиляторах C++ отсутствовали области имен и новейшие свойства шаблонов. А поддержка стандартной библиотеки шаблонов, описанной в главе 15, была неполной. В тех вычислительных системах, где используется версия 2.0 (или более поздняя) транслятора cfront, отранслированный код может передаваться компилятору C, который не полностью совместим со стандартом ANSI, в результате чего в нем остались нереализованными некоторые элементы языка и не поддерживаются некоторые функции и заголовочные файлы стандартной библиотеки ANSI. Кроме того, некоторые моменты и факторы, например, величина целого числа, зависят от реализации.

Примеры программ в этой книге были разработаны с применением компиляторов Microsoft Visual C++ 5.0 и Metrowerks Code Warrior Professional Release 2 на ПК Pentium, работающем под управлением Windows 95. Программы были проверены с помощью компилятора GNU g++ 2.7.1 на ПК 486, совместимом с IBM PC и работающем под управлением Linux, компилятора Watcom 10.6 на ПК Pentium, компилятора Metrowerks CodeWarrior Professional Release 2 на ПК Macintosh G3, работающем под управлением System 8.0. О расхождениях, являющихся результатом несоответствия стандарту, в книге сообщается в обобщенном виде, например: "в более старых реализациях используется `ios::fixed` вместо `ios_base::fixed`". В книге сообщается об имеющихся (в реализациях языка C++) некоторых ошибках и особенностях, которые могут приводить к затруднениям или путанице; однако в последующих версиях они уже могут быть устранены.



Чтобы сэкономить время, для всех учебных программ можно использовать один проект. Удалите предыдущий учебный файл исходного кода из списка проекта и добавьте текущий исходный код. Это позволит сэкономить место на диске.

В состав обоих компиляторов входит отладчик, который помогает локализовать ошибки выполнения.

Соглашения, используемые в этой книге

Для выделения различных частей текста мы использовали несколько типографских соглашений. Курсив используется для выделения важных слов и фраз, когда они встречаются в тексте впервые, например: *структурное программирование*. Полужирное начертание используется для выделения:

- Имен и значений, используемых в программах, например: `x`, `starship` и `3.14`
- Ключевых слов языка, например: `int` и `if else`
- Имен файлов, например: `iostream`
- Функций, например: `main()` и `puts()`

Исходный код программ C++ представлен в книге следующим образом:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "What's up, Doc!\n";
    return 0;
}
```

Информация, выводимая на экран учебной программы, представлена точно так же, а информация, вводимая пользователем, — выделена подчеркиванием:

Please enter your name:
Plato

Поскольку эта книга посвящена объектно-ориентированному программированию, мы использовали в ней геометрические фигуры, чтобы помочь вам идентифицировать различные элементы этой книги. Советы, правила и примечания обозначаются пиктограммами в виде электролампочки, указывающей кисти руки и карандаша соответственно.

Встречающиеся повсеместно правила или советы имеют следующий вид:

Приступаем к изучению языка C++

В этой главе рассматривается следующее:

- Как создавать программы C++
- Директива #include
- Общий формат программ C++
- Функция main()
- Как использовать объект cout для вывода данных
- Как размещать комментарии в программах C++
- Символ новой строки \n
- Как объявлять и применять переменные
- Как использовать объект cin для ввода данных
- Как определять и применять простые функции

Строительство дома начинается с того, что закладывается фундамент и возводятся стены. Если эта структура с самого начала не будет прочной, то потом могут возникнуть проблемы с более мелкими деталями, такими как окна, двери и паркетные полы. Аналогично этому изучение языка программирования следует начинать с общей структуры программы. Только тогда можно переходить к деталям, таким как циклы и объекты. В этой главедается общее представление о структуре программы, написанной на языке C++, и вкратце описываются функции и классы, которые позднее будут рассматриваться более подробно. (Идея заключается в том, чтобы вводить, по крайней мере, некоторые основные понятия постепенно по пути к последующим "открытиям".)

Начальные сведения о языке C++

Давайте начнем с простой программы C++, которая выводит на экран сообщение. Для вывода символов в программе из листинга 2.1 используется конструкция cout (произносится си-аут). Исходный код содержит несколько комментариев для читателя; они начинаются с символов //, и компилятор их игнорирует. Язык C++ чувствителен к выбору регистра, т.е. в этом языке различаются символы верхнего и нижнего регистров. Это означает, что нужно быть аккуратным и использовать тот же самый регистр, что и в примерах. Например, в программе используется слово cout. Если вы замените его на Cout или COUT, то компилятор отвергнет ваше предложение и сообщит, что вы используете неизвестный идентификатор. (Компилятор также требует соблюдения правописания, так что не пытайтесь вводить kout или coot.)

Листинг 2.1 Программа myfirst.cpp.

// программа myfirst.cpp – отображает на экране сообщение

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Come up and C++ me some time.";
    cout << "\n";
    return 0;
} // директива препроцессора
   // включает в программу определения
   // заголовок функции
   // начало тела функции
   // сообщение
   // начать новую строку
   // завершить функцию main()
   // конец тела функции
```

Для обозначения программы C++ обычно используется расширение имени файла `cpp`; однако, как указано в главе 1, вам, возможно, придется использовать другое расширение.

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если у вас компилятор старой версии, то вместо `#include <iostream>` вам, возможно, придется использовать `#include <iostream.h>`; в этом случае вам также пришлось бы опустить строку `using namespace std;` т.е.

```
#include <iostream> //это способ из будущего
using namespace std; //то же
```

следует заменить на

```
#include <iostream.h> //в том случае, если
//будущее еще не наступило
```

(Для некоторых компиляторов старых версий вместо `#include <iostream.h>` используйте `#include <stream.h>`; если у вас устаревший компилятор, то следует приобрести либо более новый компилятор, либо более старую книгу.) Переход от `iostream.h` к `iostream` произошел совсем недавно, и к моменту написания этой книги многие поставщики его еще не реализовали. В некоторых средах, работающих под управлением Windows, программа выполняется в отдельном окне, которое автоматически закрывается после окончания выполнения программы. В главе 1 уже говорилось, что, добавляя в программу перед оператором `return` следующую строку, можно сделать так, чтобы окно оставалось открытым до тех пор, пока не будет нажата клавиша:

```
cin.get();
```

В некоторых программах необходимо добавлять две такие строки. Этот код приводит к тому, что программа будет ожидать нажатия клавиши. Более подробно это будет рассмотрено в главе 4.

АДАПТАЦИЯ ПРОГРАММЫ

Вы можете обнаружить, что для выполнения примеров из настоящей книги потребуется вносить изменения в эти примеры. О двух наиболее распространенных изменениях упоминается в первом замечании по совместимости в данной главе. Одно из них касается стандартов языка; если у вас устаревший компилятор, то вместо `iostream` необходимо указывать `iostream.h` и опускать строку `namespace`. Второе изменение связано со средой программирования; вам, возможно, придется добавить в программу один или два оператора `cin.get()`, чтобы выходные данные программы были видны на экране. Поскольку эти изменения однаково справедливы для всех примеров настоящей книги, то данное замечание по совместимости является единственным предупреждением по этому поводу. Не забывайте о нем! В последующих замечаниях по совместимости будут содержаться предупреждения о других изменениях, которые, возможно, потребуется выполнить.

После того как с помощью выбранного редактора вы скопируете эту программу (или загрузите файлы исходного кода с Web-узла издательства Macmillan: www.mcp.com/info), с помощью своего компилятора C++

создайте, как описывается в главе 1, исполняемый код программы. Вот результат выполнения скомпилированной программы:

```
Come up and C++ me some time.
```

ВВОД И ВЫВОД ДАННЫХ В ЯЗЫКЕ С

Если вы раньше программировали на С, то, увидев `cout` вместо функции `printf()`, можете удивиться. Конечно, в языке C++ можно использовать функции `printf()`, `scanf()` и все остальные функции ввода и вывода стандартного языка С при условии, что будет включен обычный файл `stdio.h` этого языка. Но это книга посвящена языку C++, так что используйте новые средства ввода языка C++, которые во многих отношениях лучше, чем их аналоги в языке С.

Программа C++ строится из отдельных блоков, называемых *функциями*. Как правило, программа разделяется на ряд крупных задач, а затем для выполнения этих задач разрабатываются отдельные функции. Программа из листинга 2.1 достаточно проста и состоит из одной функции с именем `main()`. Эта программа (`myfirst.cpp`) содержит следующие элементы:

- Комментарии, на которые указывает префикс `//`
- Директиву препроцессора `#include`
- Директиву `using namespaces`
- Заголовок функции: `int main()`
- Тело функции, ограниченное символами `{` и `}`
- Оператор, в котором для вывода сообщения на экран используется объект `cout`
- Оператор `return`, завершающий выполнение функции `main()`

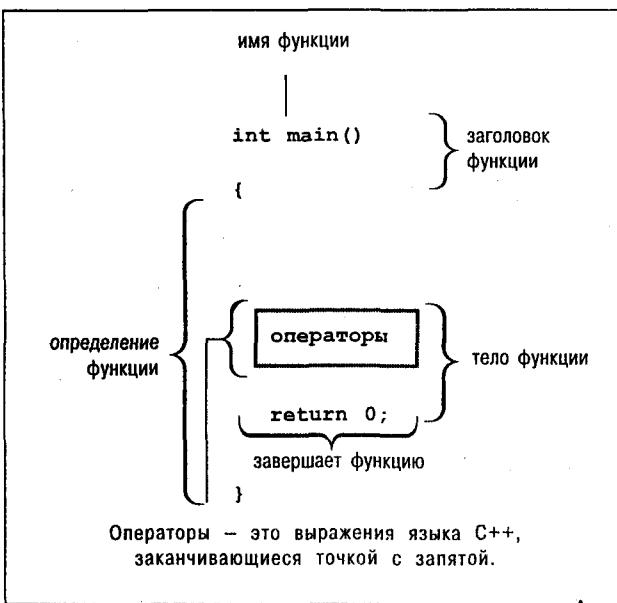
Рассмотрим теперь эти разнообразные элементы языка более подробно. Будет правильно начать с функции `main()`, так как понять некоторые предшествующие ей элементы, например директиву препроцессора, будет проще после знакомства с функцией `main()`.

Функция `main()`

В своей основе учебная программа, показанная в листинге 2.1, имеет следующую общую структуру:

```
int main()
{
    операторы
    return 0;
}
```

Эти строки означают, что перед вами функция по имени `main()`, и они описывают, как она работает. Все вместе они составляют *определение функции*. Это определение состоит из двух частей: первой строки `int main()`, которая называется *заголовком функции*, и остальной части, заключенной в фигурные скобки `{` и `}`, которая является *телом функции*. На рис. 2.1 изображена функция `main()`.

РИСУНОК 2.1 Функция *main()*.

Заголовок функции в краткой форме определяет интерфейс между функцией и остальной частью программы, а тело функции содержит инструкции для компьютера, т.е. определяет то, что собственно делает функция. В языке C++ каждая законченная инструкция называется *оператором*. Каждый оператор должен заканчиваться точкой с запятой, так что не опускайте точки с запятыми при вводе примеров программ.

Заключительный оператор в функции *main()*, называемый *оператором возврата*, завершает функцию. Далее в этой главе вы познакомитесь ближе с оператором возврата.

ОПЕРАТОРЫ И ТОЧКИ С ЗАПЯТЫМИ

Оператор представляет собой законченную инструкцию для компьютера. Чтобы понять исходный код, компилятор должен знать, когда заканчивается один оператор и начинается другой. В некоторых языках программирования используются разделители операторов. Например, в языке FORTRAN символ конца строки отделяет один оператор от следующего. В языке Pascal один оператор от следующего отделяется точкой с запятой. В некоторых случаях точку с запятой в языке Pascal можно опускать, например, после оператора перед словом END, когда фактически не происходит разделение двух операторов. (Прагматики и сторонники краткого стиля будут спорить о том, подразумевается ли под словом *можно* слово *следует*.) Но в языке C++, так же как и в языке C, применяется скорее признак (указатель) конца, чем разделитель. Признак конца – это точка с запятой, которая отмечает конец оператора; она является скорее частью оператора, чем разделителем между операторами. Практический результат заключается в том, что в языке C++ никогда нельзя опускать точку с запятой.

Заголовок функции в роли интерфейса

Сейчас самое главное — это помнить, что синтаксис языка C++ требует, чтобы определение функции *main()* начиналось со следующего заголовка: *int main()*. Более подробно этот вопрос освещается далее в настоящей главе, но для тех, кто не может сдержать своего любопытства, приводим несколько предварительных замечаний.

В общем случае функция C++ активизируется или вызывается другой функцией, а заголовок некоторой функции описывает интерфейс между этой функцией и той, которая ее вызывает. Слово, стоящее перед именем функции, называется *возвращаемым типом функции*; оно описывает информацию, передаваемую из функции назад в ту функцию, которая ее вызвала. Информация в круглых скобках, следующих за именем функции, называется *списком аргументов* или *списком параметров*; этот список описывает информацию, передаваемую из вызывающей в вызываемую функцию. Этот общий формат применительно к функции *main()* немного сбивает с толку, так как обычно функция *main()* не вызывается из других частей программы. Как правило, функция *main()* вызывается кодом начальной загрузки, который добавляется в программу компилятором для связи программы с операционной системой (UNIX, Windows 95 и т.п.). По существу, заголовок функции описывает интерфейс между функцией *main()* и операционной системой.

Рассмотрим этот интерфейс для функции *main()*, начиная со слова *int*. Функция C++, вызываемая другой функцией, может возвращать в вызывающую функцию некоторое значение. Это значение называется *возвращаемым*. В данном случае функция *main()* может возвращать целое число (целочисленное значение), на что указывает ключевое слово *int*. Затем обратите внимание на пустые круглые скобки. В общем случае, когда функция C++ вызывает другую функцию, она может передавать в нее информацию. Эту информацию описывают аргументы функции, заключенные в круглые скобки. В данном случае пустые скобки означают, что функция *main()* не принимает никакой информации или, по обычной терминологии, — что функция *main()* не принимает аргументов. (Фраза "функция *main()* не принимает аргументов" не означает, что *main()* является какой-то неразумной, властной функцией. Термин *аргумент* используется поклонниками компьютеров для обозначения информации, передаваемой от одной функции к другой.)

Итак, заголовок

```
int main()
```

означает, что функция *main()* может возвращать целочисленное значение функции, которая ее вызвала, и не принимает от нее никакой информации.

Во многих существующих программах используется заголовок из классического языка C:

```
main() //первоначальный стиль языка С
```

Если в языке С возвращаемый тип в заголовке функции опускается, то это означает, что функция имеет возвращаемый тип `int`. Однако в языке C++ такая форма выводится из употребления.

Можно также использовать следующий вариант:

```
int main(void)      //излишне явный стиль
```

Использование в круглых скобках ключевого слова `void` — это явный способ указания на то, что функция не принимает аргументов. Пустые круглые скобки в языке C++ (но не в языке С) означают то же самое, что и слово `void` в круглых скобках. (В языке С пустые скобки означают, что программист ничего не сообщает о том, имеются ли у функции аргументы или нет.)

Некоторые программисты используют такой заголовок и опускают оператор возврата:

```
void main()
```

Логически это правильно, так как возвращаемый тип `void` означает, что функция не возвращает никакого значения. Этот вариант используется во многих системах; но поскольку он не требуется существующими стандартами, то в некоторых системах он отсутствует.

Наконец, стандарт ANSI/ISO C++ гласит, что

```
return 0;
```

неявно подразумевается в конце функции `main()` (но не другой функции), если вы не указали его явно.

Когда функция `main()` не соответствует своему названию?

В программе C++ должна присутствовать функция с именем `main()`. (Не `Main()`, `MAIN()` или `mane()`! Помните, что здесь учитывается и регистр клавиатуры, и правописание.) Поскольку программа `myfirst.cpp` содержит только одну функцию, эта функция должна называться именно `main()`. Выполнение программы C++ всегда начинается с вызова этой функции. Поэтому, если в программе нет такой функции, то у вас нет и законченной программы; компилятор в этом случае указывает, что функция `main()` не была определена.

Из этого правила имеются исключения. Например, в операционной системе Windows можно написать программу, которая является модулем библиотеки динамических связей (`DLL`). Эта программа может использоваться другими программами в системе Windows. Поскольку модуль `DLL` не является отдельной программой, он не нуждается в функции `main()`. Программы специального назначения, например для микросхемы контроллера робота, могут не нуждаться в функции `main()`. Но обычная отдельная программа должна иметь функцию `main()`; в настоящей книге рассматриваются именно такие программы.

Комментарии в языке C++

В языке C++ комментарии обозначаются двойной наклонной чертой `//`. Комментарий — это написанное программистом замечание, которое предназначается для читателя; обычно он идентифицирует раздел программы или поясняет некоторые аспекты кода программы. Компилятор игнорирует комментарии. В конце концов, он знает язык C++ не хуже вас, и в любом случае он просто не понимает комментарии. Что касается компилятора, то для него листинг 2.1 выглядит так, как если бы он был написан без комментариев:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Come up and C++ me some time.";
    cout << "\n";
    return 0;
}
```

Комментарии языка C++ занимают место от символов `//` до конца строки. Комментарий может находиться в отдельной строке или в той же строке, что и код программы. Между прочим, обратите внимание на первую строку листинга 2.1:

```
// программа myfirst.cpp — отображает
// на экране сообщение
```

Все программы в этой книге начинаются с комментария, который содержит имя файла исходного кода и краткие сведения о программе. Как упоминалось в главе 1, расширение имени файла исходного кода зависит от реализации C++. В других системах вам, возможно, придется использовать имя `myfirst.C` или `myfirst.cxx`.

СОВЕТ

В своих программах обязательно применяйте комментарии; они являются своего рода документом, поясняющим программу. Чем сложнее программа, тем более ценными становятся комментарии. Они помогут понять то, что вы написали, не только другим, но и вам, особенно по прошествии некоторого времени.

КОММЕНТАРИИ В СТИЛЕ ЯЗЫКА С

В программах C++ можно использовать комментарии из языка С, которые заключены между символами `/*` и `*/`:

```
#include <iostream> /* комментарий из
языка С */
```

Поскольку комментарий из языка С заканчивается не символом конца строки, а символом `*/`, его можно продолжать на несколько строк. В программах можно использовать любой из этих двух видов комментариев или оба вместе. Однако старайтесь придерживаться языка C++. Тогда заглядывающий через ваше плечо программист, который пишет на языке С, будет знать, что вы продвинулись на более высокий уровень программирования.

Препроцессор C++ и файл *iostream*

Здесь кратко освещается то, что вам необходимо знать по данному вопросу. Если в программе должны использоваться обычные средства ввода или вывода языка C++, включите в нее две строки:

```
#include <iostream>
using namespace std;
```

Если компилятору не понравятся эти строки (например, он сообщит, что не может найти файл *iostream*), попробуйте заменить их на одну следующую строку:

```
#include <iostream.h> // совместимо с
                     // компиляторами более
                     // ранних версий
```

Фактически это все, что необходимо знать, чтобы программы работали; однако мы рассмотрим этот вопрос немного подробнее.

При создании исполняемого кода программ C++, так же как и в случае с программами C, используется *препроцессор*. Это программа, которая обрабатывает исходный файл перед основной компиляцией. (Как вы помните из главы 1, в некоторых реализациях C++ используется транслирующая программа, которая преобразует программу C++ в программу C. Хотя транслятор — это тоже вид препроцессора, мы его здесь не рассматриваем; мы рассматриваем препроцессор, который обрабатывает директивы, начинающиеся с символа #.) Чтобы вызвать этот препроцессор, не надо делать ничего особенного. Он запускается автоматически при компиляции программы.

В листинге 2.1 используется директива `#include`:

```
#include <iostream> // директива
                  // ПРЕПРОЦЕССОРА
```

Эта директива приводит к тому, что препроцессор добавляет в программу содержимое файла *iostream*. Это типичное для препроцессора действие: добавление или изменение текста в исходном коде перед компиляцией.

Отсюда возникает вопрос: почему в программу необходимо добавлять содержимое файла *iostream*? Это связано с передачей данных между программой и внешним миром. Буквы *io* в слове *iostream* означают *ввод* (передачу данных в программу) и *вывод* (передачу данных из программы). Схема ввода/вывода в языке C++ включает в себя несколько определений, которые находятся в файле *iostream*. Вашей первой программе необходимы эти определения, чтобы с помощью конструкции `cout` отобразить на экране сообщение. Директива `#include` приводит к тому, что содержимое файла *iostream* передается в компилятор вместе с содержимым исходного файла. В сущности, содержимое файла *iostream* заменяется в программе строку `#include <iostream>`. Исходный файл не изменяется, а объединенный файл, созданный из исходного файла и файла *iostream*, обрабатывается на следующем этапе компиляции.

ПОМНИТЕ

Программы, в которых для ввода и вывода используются конструкции `cin` и `cout`, должны включать файл *iostream*.

Имена заголовочных файлов

Такие файлы, как *iostream*, называются *файлами включения* (поскольку они включаются в другие файлы) или *заголовочными файлами* (поскольку они включаются в начале файла). Компиляторы C++ поставляются со многими заголовочными файлами, каждый из которых поддерживает отдельное семейство программных средств. Заголовочные файлы в языке C по традиции имеют расширение *h*, это самый простой способ идентификации типа файла по его имени. Например, заголовочный файл *math.h* поддерживает различные математические функции языка C. В языке C++ первоначально была принята такая же система наименования. Например, заголовочный файл, поддерживающий ввод и вывод, был назван *iostream.h*. Однако позднее способ именования заголовочных файлов в языке C++ изменился. Теперь расширение *h* зарезервировано для старых заголовочных файлов языка C (которые, по-прежнему, можно использовать в программах C++), а заголовочные файлы языка C++ вообще не имеют никакого расширения. Ряд заголовочных файлов языка C был преобразован в заголовочные файлы языка C++. Эти файлы были переименованы: у них убрали расширение *h* (чтобы имена были в стиле языка C++) и добавили префикс *c* (указывая, что они поддерживают элементы языка C). Например, заголовочный файл *math.h* языка C превратился в заголовочный файл *cmath* языка C++. Эти версии C и C++ заголовочных файлов языка C иногда могут быть идентичными, а иногда имеют некоторые различия. Для заголовочных файлов языка C++, таких, как *iostream*, удаление расширения *h* — это не просто косметическое изменение, поскольку эти заголовочные файлы включают в себя также области имен, которые являются следующей темой нашего краткого обзора. В табл. 2.1 приведены соглашения по именованию заголовочных файлов.

В связи с тем что существует традиция в языке C указывать тип файла с помощью расширения его имени, кажется разумным, чтобы заголовочным файлам C++ также присваивалось специальное расширение. Первоначально так и было задумано. Проблема заключалась лишь в том, чтобы договориться, каким должно быть это расширение. Однако так получилось, что комитетом ANSI/ISO было принято решение использовать заголовочные файлы без расширения.

Области имен

Если вместо файла *iostream.h* вы используете файл *iostream*, то, для того чтобы определения в этом файле были доступны вашей программе, необходимо также использовать директиву для области имен:

```
using namespace std;
```

Таблица 2.1 Соглашения по именованию заголовочных файлов.

Вид заголовочного файла	Соглашение	Пример	Комментарии
Файл C++ (старый стиль)	Оканчивается на .h	iostream.h	Используется в программах C++
Файл C (старый стиль)	Оканчивается на .h	math.h	Используется в программах C и C++
Файл C++ (новый стиль)	Расширение отсутствует	iostream	Используется в программах C++, содержит namespace std
Файл C, преобразованный в файл C++	Имеет префикс с, расширение отсутствует	cmath	Используется в программах C++, может содержать элементы, не относящиеся к языку C, например, namespace std

Она называется *директивой using*. Проще всего принять это как данное и пока об этом не думать (например, до главы 8). Но, чтобы вы не оставались в полном неведении, прочтите краткое пояснение.

Область имен — это новая особенность языка C++, предназначенная для упрощения создания программ, в которых объединяются готовые коды программ от разных поставщиков. Одна из возможных проблем заключается в том, что в двух используемых пакетах программных продуктов могут находиться функции с одинаковыми именами, например, с именем **wanda()**. Если затем в своей программе вы используете функцию **wanda()**, то компилятор не будет знать, какую именно версию вы имеете в виду. Такое средство, как область имен, дает возможность поставщику помещать свои продукты в программную единицу (блок, элемент), называемую *областью имен*, а программисту — использовать имена из этой области, чтобы указать, какой именно программный продукт ему требуется. Таким образом, компания Microflop Industries может поместить свои определения в область имен **Microflop**. Тогда имя **Microflop::wanda()** станет полным именем для функции **wanda()** этой компании. Аналогично этому имя **Piscine::wanda()** будет обозначать функцию **wanda()** компании Piscine Corporation. Тогда в программе можно использовать эти области имен, чтобы отличать друг от друга разные функции с одинаковыми именами:

```
// использовать версию из области
// имен Microflop
Microflop::wanda("go dancing?");

// использовать версию из области
// имен Piscine
Piscine::wanda("a fish named Desire")
```

Точно так же классы, функции и переменные, являющиеся стандартными компонентами компиляторов C++, помещаются в область имен, которая называется **std**. Так делается в файлах без расширения **h**. Это означает, например, что переменная **cout**, которая используется для вывода данных и определение которой находится в файле **iostream**, в действительности называется **std::cout**. Однако большинство пользователей не хотели бы преобразовывать свои коды программ без областей имен, в которых используются файл **iostream.h** и имя

cout, в коды программ с областями имен, в которых используются файл **iostream** и имя **std::cout**, если только они не могут сделать это без особого труда. Вот тут-то и появляется директива *using*. Имена, определенные в области имен **std**, можно использовать без префикса **std::** — вот смысл следующей директивы:

```
using namespace std;
```

Вывод данных в языке C++ с использованием объекта cout

Посмотрим теперь, каким образом выводится на экран сообщение. В программе **myfirst.cpp** для этого используется следующий оператор C++:

```
cout << "Come up and C++ me some time.";
```

Информация, заключенная в двойные кавычки, является сообщением, которое должно быть выведено на экран. В языке C++ любая последовательность символов, заключенная в двойные кавычки, называется *строкой* — главным образом потому, что она состоит из нескольких символов, соединяемых вместе в более крупный блок (элемент). Символы **<<** указывают на то, что этот оператор отправляет данную строку в объект **cout**; эти символы указывают направление потока информации. А что это за объект **cout**? Это предопределенный объект, который знает, как отображать на экране разнообразные данные, включая строки, числа и отдельные символы. (Вспомните, что объект, как говорилось в главе 1, является экземпляром класса, а класс определяет способ хранения и использования данных.)

М-да... получается немного неловкая ситуация. Вы будете готовы изучать объекты только через несколько глав, а уже сейчас вам приходится использовать в программе объект. На самом деле это обнаруживает одну из сильных сторон объектов. Чтобы применять объекты, вам не требуется знать их внутреннее строение. Все, что вы должны знать, — это интерфейс объекта или, другими словами, как пользоваться объектом. Объект **cout** имеет простой интерфейс. Если переменная **string** представляет собой строку, то, чтобы отобразить эту строку на экране, выполните следующее:

```
cout << string;
```

Вот и все. Но давайте посмотрим, что представляет собой этот процесс теоретически. С точки зрения теории выходные данные являются потоком, т.е. последовательностью символов, выходящих из программы. Объект `cout`, свойства которого определены в файле `iostream`, представляет этот поток. К свойствам объекта `cout` относится операция вставки (`<<`), в результате которой информация вставляется в поток, начиная с правого края. Итак, оператор (обратите внимание на заключительную точку с запятой)

```
cout << "Come up and C++ me some time.";
```

вставляет в выходной поток строку "Come up and C++ me some time." Таким образом, скорее можно сказать, что программа вставляет строку в выходной поток, чем то, что оно отображает на экране сообщение. В некотором отношении это звучит более впечатляюще (рис. 2.2).

Если вы переходите на язык C++ с языка С, то, вероятно, заметили, что знак операции вставки (`<<`) выглядит точно так же, как и знак поразрядной операции сдвиг влево (left-shift). Это пример *перегрузки операции*, которая заключается в том, что один и тот же знак операции может иметь разные значения. Компилятор определяет значение знака операции исходя из контекста программы. В языке С также имеется перегрузка некоторых операций. Например, знак `&` обозначает как оператор адресации, так и поразрядную операцию AND. Знак `*` обозначает как умножение, так и разыменование указателя. Здесь важно не то, что именно представляют собой эти операции, а то, что один и тот же знак

операции имеет несколько значений и компилятор определяет соответствующее значение исходя из контекста. (Вы делаете примерно то же самое, когда определяете значение слова "shoot" в выражениях "shoot the breeze" (дыхание бриза) и "shoot the piano player" (игра на фортепиано).) В языке C++ по сравнению с языком С понятие перегрузки операций становится шире: разрешается переопределять значения операций для определяемых пользователем типов данных, называемых классами.

Символ новой строки (\n)

Теперь более внимательно рассмотрим странного вида запись во втором операторе вывода:

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

В языке C++ (и в языке С тоже) комбинация символов `\n` является специальной формой записи важного понятия, которому присвоено название *символ новой строки*. Хотя символ новой строки состоит из двух символов (`\` и `n`), они считаются единым символом. Обратите внимание на то, что здесь используется не наклонная черта (/), а обратная наклонная черта (\). Если вы выводите символ новой строки на экран, то курсор экрана перемещается на начало новой строки, если — на принтер, то печатающая головка перемещается на начало следующей строки. Символ новой строки заслуженно носит свое имя.

Обратите внимание, что при выводе строки с помощью конструкции `cout` автоматический переход на начало следующей строки не происходит, поэтому после первого оператора `cout` курсор остается на позиции, следующей за точкой в конце выведенной строки. Чтобы установить курсор на начало следующей строки, необходимо вывести символ новой строки, или, пользуясь жаргоном, принятым в языке C++, можно сказать, что необходимо вставить символ новой строки в выходной поток.

Символ новой строки можно применять так же, как и любой другой обычный символ. В программе из листинга 2.1 этот символ выводится в виде отдельной строки, но его можно было бы вывести и в составе первой строки. Так, два оператора вывода можно заменить следующим оператором:

```
cout << "Come up and C++ me some time.\n";
```

Символ новой строки можно поместить даже в середину строки. Например, рассмотрим следующий оператор:

```
cout << "I am a mighty stream\nof lucid  
    \nclarify.\n";
```

Каждый символ новой строки приводит к перемещению курсора на начало следующей строки, в результате чего выходные данные будут выглядеть на экране следующим образом:

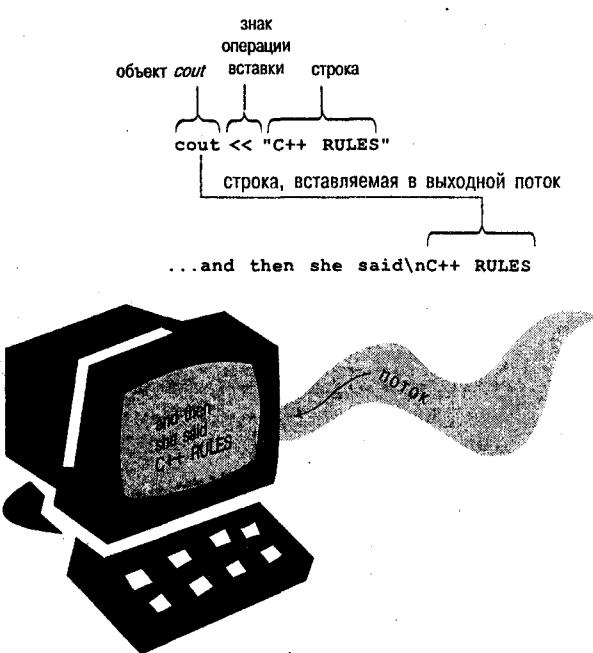


РИСУНОК 2.2 Отображение на экране строки с помощью объекта `cout`.

I am a mighty stream
of lucid
clarity.

Если не использовать символы новой строки, то можно сделать так, чтобы последовательные операторы `cout` осуществляли вывод в одной строке. Например, операторы

```
cout << "The Good, the";
cout << "Bad, ";
cout << "and the Ukulele\n";
```

отображают на экране следующие выходные данные:

```
The Good, thebad, and the Ukulele
```

Обратите внимание, что каждая очередная строка начинается непосредственно после окончания предыдущей строки. Если на стыке двух строк требуется пробел, то его необходимо включить в одну из строк. (Помните, чтобы попробовать выполнить эти примеры, их необходимо поместить в законченную программу с функцией `main()`, которая состоит из заголовка и тела, расположенного внутри фигурных скобок.)

Чтобы в выходных данных указать переход на новую строку, в языке С имеется еще один способ — использование слова `endl`:

```
cout << "What's next?" << endl;
// endl означает начало новой строки
```

Этот термин определен в файле `iostream`. Его немного легче вводить, чем "\n", но использовать этот термин можно лишь отдельно, а не как часть строки. Например, строка "What's next?\n" включает в себя символ новой строки, а строка "What's next?endl" — это просто строка, оканчивающаяся на четыре буквы: e, n, d и l.

Форматирование исходного кода C++

В некоторых языках программирования, например в языке FORTRAN, в одной строке может находиться только один оператор. В таких языках символ возврата каретки служит для разделения операторов. Однако в языке C++ конец каждого оператора обозначает точка с запятой. Поэтому в языке C++ можно трактовать символ возврата каретки таким же образом, как и символ пробела или табуляции. Иначе говоря, в языке C++ на месте символа возврата каретки можно использовать пробел и наоборот. Это означает, что один оператор может занимать несколько строк, а в одной строке может находиться несколько операторов. Например, программу `myfirst.cpp` можно переформатировать следующим образом:

```
#include <iostream>
using
namespace
std;
int
main
() { cout
<<
```

```
"Come up and C++ me some time.";
cout << "\n"; return 0; }
```

Это немного некрасивый, но действительный код. Конечно, необходимо соблюдать некоторые правила. В частности, в языках С и C++ нельзя ставить символ пробела, табуляции или символ возврата каретки в середине такого элемента, как имя; нельзя также помещать символ возврата каретки в середину строки.

```
int ma in() // НЕДЕЙСТВИТЕЛЬНЫЙ -
// пробел внутри имени
re
turn 0; // НЕДЕЙСТВИТЕЛЬНЫЙ -
// возврат каретки
// в середине слова
cout << "Behold the Beans
of Beauty!"; // НЕДЕЙСТВИТЕЛЬНЫЙ -
// возврат каретки в
// середине строки
```

Неделимые элементы кода программы называются лексемами (рис. 2.3). Обычно одна лексема должна отделяться от другой символом пробела, табуляции или возврата каретки, у которых есть общее название — *пробельные символы (white space)*. Некоторые одиночные символы, например круглые скобки и запятые, являются лексемами, которые не требуется отделять пробельными символами.

```
return0; // НЕДЕЙСТВИТЕЛЬНЫЙ,
// должно быть return 0;
return(0); // ДЕЙСТВИТЕЛЬНЫЙ,
// пробельные символы опущены
return (0); // ДЕЙСТВИТЕЛЬНЫЙ, хотя
// используется символ пробела
int main() // ДЕЙСТВИТЕЛЬНЫЙ,
// пробельные символы опущены
int main () // ТАКЖЕ ДЕЙСТВИТЕЛЬНЫЙ, хотя
// используется 2 пробела
```

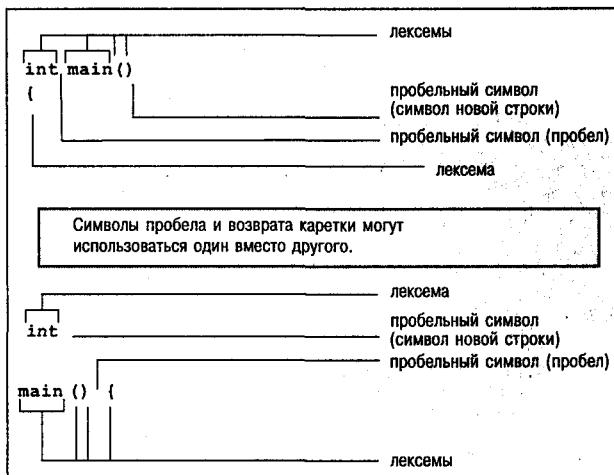


РИСУНОК 2.3 Лексемы и пробельные символы.

Формат исходного кода программ C++

Хотя язык C++ дает большую свободу в отношении форматирования кода программы, читать программу будет легче, если у нее определенный, разумный формат. Если у вас будет правильный, но некрасивый код программы, то вряд ли вы останетесь довольны. Большинство программистов придерживаются формата, который используется в листинге 2.1 и подчиняется следующим правилам:

- В строке присутствует один оператор
- Открывающая и закрывающая фигурные скобки для функции располагаются каждая в отдельной строке
- Операторы функции располагаются с отступом по отношению к этим фигурным скобкам
- Символы пробела не ставятся с обеих сторон круглых скобок, следующих за именем функции.

Цель первых трех правил проста — код программы должен быть аккуратным и легко читаемым. Четвертое правило помогает отличать функции от других, встроенных структур языка C++, например, циклов, которые также содержат круглые скобки. Далее по ходу книги вы познакомитесь и с другими правилами форматирования кода программы.

Краткие сведения об операторах языка C++

Программа C++ представляет собой совокупность функций, а каждая функция — это совокупность операторов. В языке C++ имеется несколько видов операторов; рассмотрим некоторые из них. В листинге 2.2 содержатся два новых вида операторов. Во-первых, оператор объявления создает переменную. Во-вторых, оператор присваивания присваивает этой переменной некоторое значение. Кроме того, в программе показаны новые возможности объекта cout.

Листинг 2.2 Программа fleas.cpp.

```
// fleas.cpp - отображает на экране
// значение переменной
#include <iostream>
using namespace std;
int main()
{
    int fleas;           //создает целочисленную
                        //переменную
    fleas = 38;          //присваивает значение
                        //этой переменной
    cout << "My cat has ";
    cout << fleas;      //отображает на экране
                        //значение переменной
fleas
    cout << " fleas.\n";
    return 0;
}
```

Пустая строка отделяет объявление от остальной части программы. В программах C это общепринятая практика, но в программах C++ она распространена не так широко. В результате выполнения программы получается следующий результат:

```
My cat has 3 fleas.
```

На нескольких последующих страницах рассматривается, как работает эта программа.

Операторы объявления и переменные

Компьютеры — это точные, подчиняющиеся строгому порядку машины. Чтобы запомнить в компьютере какой-нибудь элемент информации, необходимо указать, в каком месте памяти он будет храниться, а также объем области памяти, требуемой для хранения этого элемента информации. В языке C++ это можно сделать относительно безболезненно с помощью *оператора объявления*, в котором указываются тип запоминаемой информации и идентификатор для выделяемой области памяти. Например, в программе имеется такой оператор объявления (обратите внимание на точку с запятой):

```
int fleas;
```

Этот оператор выделяет в памяти достаточно места для хранения информации, тип которой обозначается словом *int*. Всю работу по выделению области памяти и присвоению ей идентификатора выполняет компилятор. В языке C++ можно обрабатывать данные нескольких видов (типов), а *int* — это самый основной тип данных. Он соответствует целому числу, т.е. числу без дробной части. Данные типа *int* в языке C++ могут быть положительными и отрицательными, а диапазон значений зависит от реализации языка. В главе 3 подробно рассматриваются данные типа *int* и данные других основных типов.

Помимо типа данных оператор объявления также говорит о том, что для идентификации значения в этой области памяти программа, начиная с этого момента, будет использовать имя *fleas*. Поскольку это значение может изменяться, *fleas* называется *переменной*. В языке C++ требуется объявлять все переменные. Если в программе *fleas.cpp* опустить объявление этой переменной, то при попытке использовать далее в программе переменную *fleas* компилятор выдаст сообщение об ошибке. (Вы действительно могли бы попробовать опустить это объявление переменной, чтобы увидеть каким сообщением отреагирует на это ваш компилятор. Тогда, если в будущем вы увидите это сообщение, то будете знать, что необходимо проверить, не пропущены ли объявления переменных.)

ПОЧЕМУ НЕОБХОДИМО ОБЪЯВЛЯТЬ ПЕРЕМЕННЫЕ

В некоторых языках, например в BASIC, новая переменная создается без каких-либо явных объявлений каждый раз, когда в программе используется новое имя. Может показаться, что для пользователя так удобнее, и это действительно так, но в краткосрочной перспективе. Проблема заключается в том, что, совершив ошибку в написании имени переменной, можно непреднамеренно создать новую переменную, не сознавая этого. Так, например, в языке BASIC можно сделать примерно следующее:

```
CastleDark = 34
CastleDank = CastleDank + MoreGhosts
PRINT CastleDark
```

Поскольку изменения осуществляются над записанной по ошибке переменной **CastleDank** (вместо *r* написали *n*), переменная **CastleDark** остается неизменной. Ошибки такого типа тяжело обнаруживать, так как правила языка BASIC при этом не нарушаются. Однако аналогичный код, написанный на языке C++, нарушает правило, требующее объявлять переменную перед ее использованием, поэтому компилятор перехватывает эту ошибку, не позволяя ей оставаться в программе.

Таким образом, в общем случае объявление содержит тип данных, которые будут храниться в выделяемой области памяти, и имя, используемое программой для обозначения этих данных. В данном конкретном случае в программе создается переменная с именем **fleas**, в которой может запоминаться целое число (рис. 2.4).

Оператор объявления в данной программе называется оператором *определяющего объявления* (*a defining declaration statement*) или, для краткости, *определением*. Это означает, что компилятор выделяет для переменной область памяти. В более сложных ситуациях могут применяться *ссыльные объявления* (*reference declarations*). Они сообщают компьютеру, чтобы он использовал переменную, которая уже была определена в другом месте. Вообще, объявление не обязательно является определением, но в данном примере это именно так.

Если вы знакомы с языком С или Pascal, значит, имеете представление и об объявлениях переменных. Однако вас может ожидать сюрприз.



РИСУНОК 2.4 Объявление переменной.

В языках С или Pascal все объявления переменных обычно располагаются в самом начале функции или процедуры. В языке C++ такое ограничение отсутствует. Действительно, в языке C++ принято объявлять переменные непосредственно перед тем, как они впервые будут использоваться. В этом случае не требуется рыться в программе в обратном направлении, чтобы увидеть тип переменной. Пример этого вы увидите далее в настоящей главе. Данный стиль имеет тот недостаток, что объявления всех переменных не располагаются в одном месте. Таким образом, вы не можете с первого взгляда сказать, какие переменные используются в функции.



COBET

В языке C++ принято объявлять переменные как можно ближе к тому месту, где они впервые используются.

Оператор присваивания

Оператор присваивания присваивает значение некоторой области памяти. Например, оператор

```
fleas = 38;
```

присваивает целое число 38 области памяти, выделенной для переменной **fleas**. Знак **=** обозначает *операцию присваивания*. В языке C++ (и в языке С) имеется одна необычная особенность: операцию присваивания можно использовать последовательно, т.е. следующий код программы будет действительным:

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

Присваивание происходит в направлении справа налево. Сначала число 88 присваивается переменной **steinway**; затем значение переменной **steinway**, которое теперь равно 88, присваивается переменной **baldwin**; потом значение переменной **baldwin**, равное 88, присваивается переменной **yamaha**. (Язык C++, так же как и язык С, склонен допускать причудливые коды программ.)

Важный момент, связанный с объектом cout

В приводимых до сих пор примерах объекту **cout** для вывода на печать передавались строки. Помимо этого, в листинге 2.2 в объект **cout** передавалась переменная, значением которой является целое число:

```
cout << fleas;
```

Программа не выводит на печать слово **fleas**; вместо этого на печать выводится целочисленное значение, хранимое в переменной **fleas** (оно равно 38). Фактически здесь присутствует даже не один, а два важных момента. Во-первых, в объекте **cout** имя переменной **fleas** замещается ее текущим числовым значением 38. Во-вторых, это значение преобразуется в соответствующие выходные символы.

Как видите, объект `cout` может работать как со строками, так и с целыми числами. Возможно, для вас в этом нет ничего особенного, но имейте в виду, что целое число 38 — это нечто совсем иное, чем строка "38". Стока содержит символы, с помощью которых записывается это число: символ 3 и символ 8. И в памяти компьютера хранится код для символов 3 и 8. Чтобы вывести на печать строку, объект `cout` просто выводит на печать каждый символ строки. Однако целое число 38 хранится как числовое значение, т.е. каждая цифра числа не хранится в компьютере отдельно, а число 38 хранится в виде двоичного числа. (Двоичные числа рассматриваются в приложении А.) Главное здесь заключается в том, что в объекте `cout` перед выводом на печать целое число должно быть преобразовано в символьную форму. Более того, объект `cout` обладает способностью определять, что переменная `fleas` представляет собой целое число, требующее преобразования.

Возможно, сопоставление с языком С покажет вам, насколько объект `cout` "умнее" функции `printf()`. Чтобы вывести на печать строку "38" и целое число 38, в языке С используется многоцелевая функция вывода `printf()`:

```
printf("Printing a string: %s\n", "38");
printf("Printing an integer: %d\n", 38);
```

Не вдаваясь во все сложности работы функции `printf()`, заметим, что необходимо использовать специальные коды (`%s` и `%d`), чтобы указать, печатаете ли вы строку или целое число. И если вы дали команду функции `printf()` печатать строку, а вместо строки по ошибке передали ей целое число, то функция `printf()` не заметит эту ошибку. Она просто выполняет свою работу и отображает на экране информационный мусор.

Объект `cout` работает более разумно, что является следствием объектно-ориентированных особенностей языка C++. По сути, операция вставки в языке C++ выполняется по-разному, в зависимости от типа данных, следующих за знаком этой операции (`<<`). Это пример перегрузки операций. В последующих главах, когда вы приметесь за изучение перегрузки функций и перегрузки операций, сами научитесь выполнять подобные действия.

ОБЪЕКТ `cout` И ФУНКЦИЯ `printf()`

Если вы привыкли к языку С и функции `printf()`, то объект `cout` может показаться вам немного сложным. Возможно, вы даже предпочтете работать с функцией `printf()`, чтобы иметь возможность пользоваться своим тяжело приобретенным мастерством. Но на самом деле объект `cout` можно использовать с таким же успехом, как и функцию `printf()`. Здесь важно то, что объект `cout` обладает значительными преимуществами. Его способность распознавать типы данных отражает тот факт, что объект `cout` более сложен, "разумен" и защищен от неумелого использования. Кроме того, он является расширяемым. Это значит, что операцию вставки (`<<`) можно переопределить таким образом, что объект `cout` сможет распоз-

навать и выводить на экран создаваемые вами данные новых типов. И если вам нравится, что функция `printf()` обеспечивает хорошее управление, вы можете осуществить то же самое, используя объект `cout` более совершенным способом (см. главу 16).

Еще несколько операторов языка C++

Рассмотрим еще пару примеров операторов. Программа из листинга 2.3 является расширением предыдущего примера, она позволяет вводить данные во время выполнения программы. Для этого в ней используется объект `cin` (произносится си-ин) — противоположность объекта `cout`, осуществляющий ввод данных. Кроме того, в этой программе показан еще один способ применения объекта `cout` — этого мастера перевоплощений.

Листинг 2.3 Программа `yourcat.cpp`.

```
// yourcat.cpp - ввод и вывод
#include <iostream>
using namespace std;
int main()
{
    int fleas;

    cout <<
        "How many fleas does your cat have?\n";
    cin >> fleas; // ввод C++
// следующая строка конкатенирует вывод
    cout << "Well, that's " << fleas
        << " fleas too many!\n";
    return 0;
}
```

В результате выполнения программы получается следующее:

```
How many fleas does your cat have?
112
Well, that's 112 fleas too many!
```

В этой программе имеется две новые особенности: для ввода данных с клавиатуры применяется объект `cin`, и три оператора вывода объединяются в один. Рассмотрим эти особенности.

Применение объекта `cin`

Из выходных данных видно, что значение, вводимое с клавиатуры (112), в конце концов, присваивается переменной `fleas`. Вот оператор, который совершает это чудо:

```
cin >> fleas;
```

Глядя на этот оператор, можно без труда представить себе поток информации, текущий из объекта `cin` в переменную `fleas`. Конечно, имеется и более формальное описание этого процесса. Подобно тому как вывод данных рассматривается в языке C++ как поток символов, вытекающих из программы, ввод данных рассматривается как поток символов, втекающих в программу. Согласно определению, находящемуся в файле `iostream`,

объект `cin` представляет собой этот поток. С целью выполнения вывода данных на печать операция (`<<`) вставляет символы в выходной поток. Для ввода данных объект `cout` использует операцию (`>>`), чтобы извлечь символы из входного потока. Как правило, переменная, принимающая извлекаемую информацию, располагается справа от знака этой операции. (Символы `<<` и `>>` были выбраны, чтобы визуально показать направление, в котором передается информация.)

Подобно объекту `cout`, объект `cin` — довольно "разумный". Он преобразует входные данные, которые представляют собой просто последовательность символов, вводимых с клавиатуры, в форму, приемлемую для переменной, принимающей информацию. В данном случае переменная `fleas` объявлена в программе как целочисленная, поэтому входные данные преобразуются в числовую форму, используемую компьютером для хранения целых чисел.

И снова объект `cout`

Вторая новая особенность программы `yourcat.cpp` — это объединение трех операторов вывода в один. Операция `<<` определена в файле `iostream` так, что вывод данных можно конкатенировать следующим образом:

```
cout << "Well, that's " << fleas
     << " fleas too many.\n";
```

Это позволяет объединять в одном операторе вывод строк и вывод целых чисел. Результатирующие выходные данные будут такими же, как и выходные данные следующих трех операторов:

```
cout << "Well, that's ";
cout << fleas;
cout << " fleas too many.\n";
```

Пока вы в настроении, вот вам еще один совет относительно оператора вывода: конкатенированную версию оператора можно записать следующим образом, размещая один оператор в трех строках:

```
cout << "Well, that's "
     << fleas
     << " fleas too many.\n";
```

Это возможно потому, что, согласно правилам языка C++, допускающим свободный формат программ, символы пробела и новой строки между лексемами являются взаимозаменяемыми. Такая форма записи удобна тогда, когда программист ограничивает ширину строки.

Несколько слов о классах

Вы уже знаете кое-что об объектах `cin` и `cout`, и теперь вам пора поближе познакомиться с объектами и, в частности, с классами. Классы — это одно из центральных понятий объектно-ориентированного программирования.

Класс — это тип данных, определяемый пользователем. Чтобы определить класс, вы описываете, какого вида информацию он представляет и какого вида действия можно совершать над этими данными. Класс имеет такое же отношение к объекту, как тип данных — к переменной. Другими словами, в определении класса описывается формат данных и то, как они могут использоваться, тогда как объект представляет собой реальный экземпляр данных, созданный в соответствии со спецификацией данных. Если отвлечься от компьютерной терминологии, то можно сказать, что класс — это аналог некоторой категории, например, "известные актеры", а объект является аналогом отдельного экземпляра этой категории, например, лягушонка Кермита (Kermit the Frog). Если развивать эту аналогию дальше, то класс, представляющий актеров, должен включать определения возможных действий, относящихся к этому классу, таких как чтение роли, выражение горя, изображение угрозы, получение наград и т.п. Если вы уже знакомы с другой терминологией ООП, то это поможет вам узнать, что классы языка C++ соответствуют тому, что в некоторых других языках называется объектным типом, а объекты языка C++ соответствуют объектным экземплярам или экземплярам переменных.

Теперь давайте рассуждать более конкретно. Вспомним объявление переменной:

```
int fleas;
```

Данный оператор создает некоторую переменную (`fleas`), обладающую свойствами данных типа `int`. Это значит, что переменная `fleas` может содержать целое число и использоваться для определенных целей, например, для сложения и вычитания. Теперь рассмотрим объект `cout`. Этот объект создается таким образом, чтобы он обладал свойствами класса `ostream`. В определении класса `ostream` (еще один обитатель файла `iostream`) описывается некоторый вид данных и операции, которые можно выполнять над этими данными, такие как вставка числа или строки в выходной поток. Конкретным представителем этого вида данных является объект класса `ostream`. Аналогично этому объект `cin` обладает свойствами класса `istream`, который также определен в файле `iostream`.

ПОМНИТЕ

Класс представляет собой описание всех свойств данных некоторого типа, а объект является конкретным экземпляром данных, созданных в соответствии с этим описанием.

Вы узнали, что классы — это определяемые пользователем типы данных, но вы как пользователь, конечно, не разрабатывали классы `ostream` и `istream`. Подобно тому, как функции поставляются в виде библиотек функций, классы поставляются в виде библиотек классов. Именно так обстоит дело с классами `ostream` и

istream. Технически они не являются частью языка C++, а представляют собой примеры классов, которые поставляются вместе с реализацией языка C++. Определения классов находятся в файле *iostream*, а не встроены в компилятор. Вы можете даже модифицировать определения этих классов, но это не очень хорошая идея. (Если быть более точным, то это плохая идея.) Семейство классов *iostream* и родственное ему семейство *fstream* (для ввода/вывода файлов) — это единственные наборы классов, которые поставлялись с первыми реализациями языка C++. Однако комитет ANSI/ISO по языку C++ добавил в стандарт еще несколько библиотек классов. Кроме того, в пакеты большинства реализаций языка входят дополнительные определения классов. Действительно, нынешняя привлекательность языка C++ в значительной мере объясняется наличием обширных и полезных библиотек классов, поддерживающих программирование в операционных системах UNIX, Macintosh и Windows.

В описании класса точно определены все операции, которые могут выполняться над объектами этого класса. Чтобы произвести над некоторым объектом какое-либо допустимое действие, вы посыпаете этому объекту сообщение. Например, если вы хотите, чтобы объект *cout* отобразил на экране строку, вы посыпаете ему сообщение, которое фактически имеет следующий смысл: "Объект! Отобразить это на экране!" В языке C++ посыпать сообщения можно двумя способами. Первый способ, который называется использование метода класса, представляет собой, по существу, вызов функции, подобный тем, какие вы уже видели. Второй способ, использовавшийся при работе с объектами *cin* и *cout*, заключается в переопределении операции. Таким образом, чтобы отправить объекту *cout* сообщение "отобразить на экране строку данных", в операторе

```
cout << "I am not a crook."
```

используется переопределенная операция *<<*. В данном случае сообщение поступает вместе с аргументом, представляющим собой строку, которую необходимо отобразить на экран (рис. 2.5).

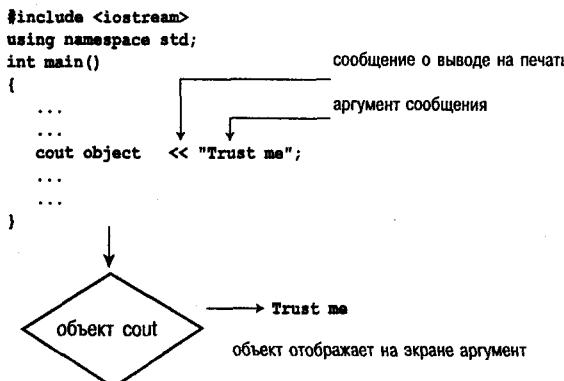


РИСУНОК 2.5 Отправка сообщения объекту.

Функции

Поскольку функции — это модули, из которых строятся программы C++, и поскольку они важны для элементов ООП в языке C++, необходимо изучить их самым тщательным образом. Некоторые аспекты этой темы довольно сложны, и в полном объеме функции будут рассматриваться в главах 7 и 8. Однако если вы познакомитесь с основными характеристиками функций сейчас, то впоследствии вам будет легче их изучать. В оставшейся части этой главы излагаются основные сведения о функциях.

В языке C++ имеется две разновидности функций: функции с возвращаемыми значениями и без них. В стандартной библиотеке функций языка C++ можно найти примеры функций обоих видов и можно также создавать свои собственные функции любого вида. Давайте рассмотрим библиотечные функции, имеющие возвращаемое значение, а также научимся создавать свои собственные простые функции.

Применение функций с возвращаемым значением

Функция, имеющая возвращаемое значение, в результате своей работы выдает значение, которое можно присвоить переменной. Например, в стандартную библиотеку C/C++ входит функция *sqrt()*, возвращающая квадратный корень числа. Предположим, что вам требуется вычислить квадратный корень числа 6.25 и присвоить полученное значение переменной *x*. Для этого в программе можно использовать такой оператор:

```
x = sqrt(6.25); //возвращает значение 2.5 и
//присваивает его переменной x
```

Выражение *sqrt(6.25)* вызывает функцию *sqrt()*. Выражение *sqrt(6.25)* называется *вызовом функции*, сама функция называется *вызываемой*, а функция, в которой находится вызов функции, называется *вызывающей* (рис. 2.6).

Значение в круглых скобках (в данном случае 6.25) представляет собой информацию, посыпанную функции; говорят, что эта информация *передается* в функцию. Значение, посыпанное в функцию таким образом, называется *аргументом* или *параметром* (рис. 2.7). Функция *sqrt()* вычисляет результат, который будет равен 2.5, и посыпает это значение назад в вызывающую функцию; это значение, посыпанное назад, называется *возвращаемым значением* функции. Можно представить себе, что возвращаемое значение замещает в операторе вызов функции после того, как функция завершит свою работу. Таким образом, в данном примере возвращаемое значение присваивается переменной *x*. Коротко говоря, аргумент — это информация, посыпанная в функцию, а возвращаемое значение — это значение, передаваемое из функции назад.

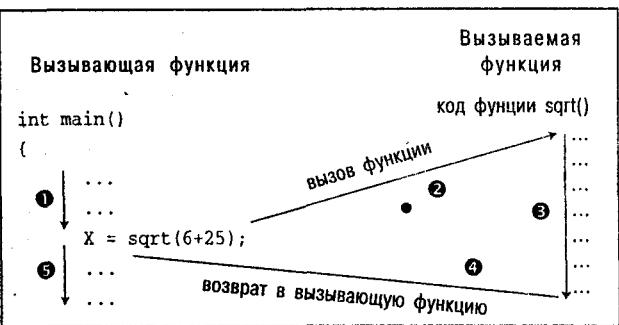


РИСУНОК 2.6 Как осуществляется вызов функции.

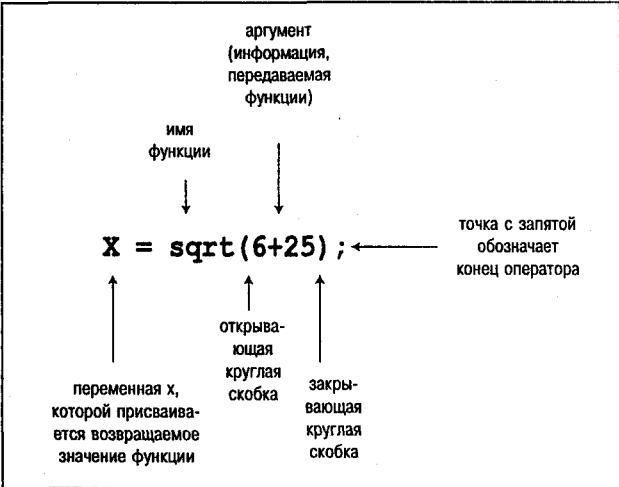


РИСУНОК 2.7 Синтаксис вызова функции.

Это практически все, что касается вызова функции. Необходимо только добавить, что, перед тем как использовать функцию, компилятор C++ должен знать, какого типа у функции аргументы и возвращаемое значение. Другими словами, что возвращает функция? Целое число? Символ? Число с дробной частью? Приговор "виновен"? Или что-нибудь еще? Если у компилятора отсутствует эта информация, он не будет знать, как интерпретировать возвращаемое значение. В языке C++ эта информация передается компилятору с помощью прототипа функции.

ПОМНИТЕ

В программе C++ для каждой используемой функции должен быть прототип.

Прототип функции является для функции тем же, чем для переменной является объявление переменной, — он информирует программу о типах данных. Например, в библиотеке C++ определено, что функция `sqrt()` принимает в качестве аргумента число с дробной частью (например, 6.25) и возвращает число того же самого типа. В некоторых языках такие числа называются вещественными, но в языке C++ они называются данны-

ми типа `double` или числами удвоенной точности. (Более подробную информацию о данных типа `double` см. в главе 3.) Прототип функции `sqrt()` выглядит так:

`double sqrt(double); // прототип функции`

Первое ключевое слово `double` означает, что функция `sqrt()` возвращает значение типа `double`. Ключевое слово `double` в круглых скобках означает, что функции `sqrt()` требуется аргумент типа `double`, т.е. этот прототип описывает функцию `sqrt()` точно так, как она используется в следующем выражении:

`x = sqrt(6.25);`

Между прочим, заключительная точка с запятой идентифицирует прототип как оператор, проводя различие между прототипом и заголовком функции. Если опустить точку с запятой, то компилятор интерпретирует эту строку как заголовок функции и ожидает, что за ней последует тело функции, являющееся ее определением. Когда в программе применяется функция `sqrt()`, до нее должен стоять прототип этой функции. Это можно обеспечить двумя способами:

- Программист сам может ввести прототип функции в исходный код программы.
- Программист может включить в программу заголовочный файл `cmath` (в более ранних реализациях — `math.h`), который содержит прототип этой функции.

Второй способ лучше, так как в этом случае более вероятно, что прототип будет правильным. Прототип любой функции из библиотеки C++ обязательно имеется в одном или нескольких заголовочных файлах. Просто посмотрите описание функции в руководстве к программе или в справочном файле, если он имеется, и вы узнаете, какой заголовочный файл требуется. Например, из описания функции `sqrt()` вы узнаете, что вам необходимо использовать заголовочный файл `cmath`. (Или, возможно, предшествовавший ему заголовочный файл `math.h`, который можно использовать как в программах C, так и в программах C++.)

Не путайте прототип функции с определением функции. Прототип, как вы видели, лишь только описывает интерфейс функции, т.е. он описывает информацию, посылаемую в функцию и из функции. А определение содержит код функции — например, код для вычисления квадратного корня числа. В языках C и C++ прототипы и определения функций хранятся раздельно. Библиотечные файлы содержат скомпилированные коды функций, тогда как заголовочные файлы содержат прототипы функций. Прототип функции должен находиться в программе до того места, где функция применяется впервые. Обычно прототип функции помещается непосредственно перед определением функции `main()`. В листинге 2.4 демонстрируется применение библиотечной функции `sqrt()`; прототип функции помещается в программу путем включения файла `cmath`.

Листинг 2.4 Программа sqrt.cpp.

```
// sqrt.cpp - использование функции sqrt()
#include <iostream>
using namespace std;
#include <cmath> // или math.h
int main()
{
    // использовать вещественные числа типа
    // double, т.е. удвоенной точности
    double cover;

    cout << "How many square feet of
        sheets do you have?\n";
    cin >> cover;
    double side; //создать еще одну переменную

    // вызвать функцию и присвоить
    // возвращаемое значение переменной
    side = sqrt(cover);

    cout << "You can cover a square with
        sides of " << side;
    cout << " feet\nwith your sheets.\n";
    return 0;
}
```

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если у вас компилятор более ранних выпусков, то в листинге 2.4 вместо директивы `#include <cmath>` вам, возможно, придется использовать директиву `#include <math.h>`.

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕЧНЫХ ФУНКЦИЙ

Библиотечные функции C++ хранятся в библиотечных файлах. Когда компилятор компилирует программу, ему приходится отыскивать библиотечные файлы для используемых функций. Компиляторы различаются тем, поиск каких библиотечных файлов они осуществляют автоматически. Если вы попытаетесь выполнить программу из листинга 2.4 и получите сообщение, что функция `_sqrt()` является неопределенным внешним модулем, то можно предположить, что ваш компилятор не осуществляет автоматический поиск библиотеки `math` (математической библиотеки). (Компиляторы любят добавлять к именам функций префикс в виде символа подчеркивания — деликатное напоминание о том, что последнее слово по поводу вашей программы за ними.) Если вы получите такое сообщение, то в документации по компилятору посмотрите, как сделать, чтобы компилятор обращался к требуемой библиотеке. Например, в реализациях языка C++ для системы UNIX требуется в конце командной строки ставить параметр `-lmath` (для библиотеки `math`):

CC sqrt.C -lmath

Простое включение заголовочного файла `cmath` обеспечивает доступ к прототипу функции, но не обязательно приводит к тому, что компилятор будет обращаться к требуемому библиотечному файлу.

Вот пример выполнения программы:

```
How many square feet of sheets do you
have?
123.21
```

You can cover a square with sides of 11.1
feet with your sheets.

Поскольку функция `sqrt()` имеет дело со значениями типа `double`, в примере используются переменные этого типа. Обратите внимание, что для объявления переменной типа `double` применяется та же самая форма записи, или синтаксис, что и для объявления переменной типа `int`:

typename variablename;

То, что переменные `cover` и `side` являются переменными типа `double`, означает, что они могут содержать числа с десятичной дробной частью, такие как 123.21 и 1.1. Как вы увидите в главе 3, данные типа `double` имеют намного больший диапазон допустимых значений, чем данные типа `int`.

Поскольку новые переменные можно объявлять в любом месте программы C++, переменная `side` в программе `sqrt.cpp` объявляется непосредственно перед ее использованием. Кроме того, при объявлении переменной ей можно сразу же присваивать значение, поэтому можно было бы сделать так:

double side = sqrt(cover);

К этому процессу, называемому *инициализацией*, мы вернемся в главе 3.

Обратите внимание, что объект `cin` знает, как преобразовывать информацию входного потока в данные типа `double`, а объект `cout` знает, как вставлять данные типа `double` в выходной поток. Как уже отмечалось, эти объекты довольно "разумны".

Разновидности функций

Некоторым функциям требуется более одного элемента информации. Эти функции имеют несколько аргументов, разделяемых запятыми. Например, математическая функция `pow()` принимает два аргумента и возвращает значение, равное первому аргументу, возведенному в степень, определяемую вторым аргументом. Она имеет такой прототип:

double pow(double, double); //прототип
//функции с двумя аргументами

Если, скажем, вам требуется найти 5 в 8-й степени, то следует использовать такой оператор:

answer = pow(5.0, 8.0); //вызов функции со
//списком аргументов

Имеются функции, которые не принимают аргументы. Например, одна из библиотек C (связанная с заголовочным файлом `cstdlib` или `stdlib.h`) содержит функцию `rand()`, которая не имеет аргументов и возвращает случайное целое число. Ее прототип выглядит таким образом:

int rand(void); //прототип функции,
//не принимающей аргументы

Ключевое слово **void** явно указывает на то, что функция не принимает аргументы. Если опустить слово **void** и оставить скобки пустыми, то это интерпретируется компилятором как неявное объявление того, что аргументы отсутствуют. Такую функцию можно применять следующим образом:

```
myGuess = rand(); // вызов функции, не
                  // имеющей аргументов
```

Обратите внимание, что, в отличие от некоторых языков программирования, скобки при вызове функции должны присутствовать даже тогда, когда аргументов нет.

Имеются также функции, у которых отсутствует возвращаемое значение. Предположим, например, что была создана функция, которая отображает на экране число в формате "доллары-центы" (в денежном формате). Вы можете послать в функцию аргумент, скажем, 23.5, и она отобразит на экране \$23.50. Поскольку функция посылает это значение на экран, а не в вызывающую программу, она не имеет возвращаемого значения. Это указывается в прототипе функции с помощью ключевого слова **void** на месте типа возвращаемого значения:

```
void bucks(double); //прототип функции,
                    //не имеющей возвращаемого значения
```

Поскольку эта функция не имеет возвращаемого значения, ее нельзя применять в операторе присваивания или в каком-нибудь другом выражении. Вместо этого вы имеете оператор чистого вызова функции:

```
bucks(1234.56) //вызов функции, возвращающее
                 //значение отсутствует
```

В некоторых языках программирования термин **функция** зарезервирован для функций с возвращаемым значением, а для функций без возвращаемого значения используется термин *процедура* или *подпрограмма*. Но в языке C++, так же как и в языке С, для обеих разновидностей применяется термин **функция**.

Функции, определяемые пользователем

В стандартной библиотеке С имеется свыше 140 предопределенных функций. Если одна из них соответствует вашим потребностям, постарайтесь любыми средствами использовать ее. Но часто программисту приходится создавать свои собственные функции, особенно при разработке классов. В любом случае разрабатывать свои собственные функции очень интересно, так что давайте рассмотрим этот процесс. Вы уже применяли несколько определяемых пользователем функций, и все они назывались **main()**. В любой программе C++ должна присутствовать функция **main()**, определяемая пользователем. Допустим, вам требуется добавить в программу вторую, определяемую пользователем функцию. Как и в случае с библиотечными функциями, для вызова определяемой пользователем функции в программе используется ее

имя. И так же, как в случае с библиотечными функциями, следует в программе поместить прототип функции до ее вызова; обычно прототип помещается непосредственно перед определением функции **main()**. Новым здесь является то, что необходимо привести также и исходный код определяемой функции. Самый простой способ — поместить этот код после кода функции **main()** в том же файле. Это иллюстрируется в листинге 2.5.

Листинг 2.5 Программа ourfunc.cpp.

```
// ourfunc.cpp - определяет вашу
// собственную функцию
#include <iostream>
using namespace std;
void simon(int); // прототип функции
simon()
int main()
{
    simon(3); // вызов функции simon()
    cout << "Pick an integer: ";
    int count;
    cin >> count;
    simon(count); // еще один вызов этой функции
    return 0;
}

void simon(int n) //определение функции simon()
{
    cout << "Simon says touch your toes "
        << n << " times.\n";
} // в функциях без возвращаемого значения
// не требуются операторы return
```

Функция **main()** вызывает функцию **simon()** дважды. Первый раз аргументом является число 3, а второй раз — переменная **count**. Между этими вызовами пользователь вводит целое число, которое заносится в переменную **count**. В сообщении-подсказке, выводимом на экран объектом **cout**, не используется символ новой строки. В результате данные, вводимые пользователем, оказываются в той же строке, что и сообщение-подсказка. Вот пример выходных данных программы:

```
Simon says touch your toes 3 times.
Pick an integer: 512
Simon says touch your toes 512 times.
```

Формат определения функции

Определение функции **simon()** имеет тот же самый формат, что и определение функции **main()**. Сначала идет заголовок функции, затем следует тело функции, заключенное в фигурные скобки. В общем случае определение функции имеет следующий формат:

```
type имя_функции(список_аргументов)
{
    операторы
}
```

Обратите внимание, что определение функции **simon()** следует сразу же за закрывающей скобкой функции **main()**. В отличие от языка Pascal и подобно языку С, в языке C++ одно определение функции нельзя

вкладывать в другое. Каждое определение функции располагается отдельно от всех остальных; все функции являются равнозначными (рис. 2.8).

Заголовки функций

Заголовок функции `simon()` имеет следующий вид:

```
void simon(int n)
```

Начальное ключевое слово `void` означает, что функция `simon()` не имеет возвращаемого значения. Следовательно, в результате вызова функции `simon()` не будет получено число, которое можно было бы присвоить какой-нибудь переменной в функции `main()`. Итак, первый вызов этой функции выглядит следующим образом:

```
simon(3); // правильно для функций без
           // возвращаемого значения
```

Поскольку у функции `simon()` отсутствует возвращаемое значение, ее нельзя использовать таким образом:

```
simple = simon(3); // недопустимо для функций
                   // без возвращаемого значения
```

Выражение `int n` в круглых скобках означает, что функция `simon()` имеет один аргумент типа `int`. Переменная `n` — это новая переменная, которой присваивается значение, передаваемое в функцию во время ее вызова. Таким образом, при вызове функции

```
simon(3);
```

переменной `n`, определенной в заголовке этой функции, присваивается значение 3. Переменная `n` в операторе `cout` означает, что используется ее значение, передаваемое в функцию при вызове. Вот почему в выходных данных функции `simon(3)` отображается число 3. В приме-

```
#include <iostream>
using namespace std;

Прототипы
функций { void simon(int);
            double taxes(double);

функция #1 { int main()
              {
                  ...
                  return 0;
              }

функция #2 { void simon(int n)
              {
                  ...
              }

функция #3 { double taxes(double t)
              {
                  ...
                  return 2 * t;
              }
}
```

РИСУНОК 2.8 Определения функций располагаются в файле программы последовательно.

ре выходных данных при втором вызове функции `simon(count)` отображается число 512, так как это значение было присвоено переменной `count`. Иными словами, заголовок функции `simon()` информирует программу о том, что эта функция принимает один аргумент типа `int` и у нее отсутствует возвращаемое значение.

Давайте теперь рассмотрим заголовок функции `main()`:

```
int main()
```

Ключевое слово `int` означает, что функция `main()` возвращает целочисленное значение. Пустые круглые скобки (которые могли бы содержать слово `void`) означают, что функция `main()` не имеет аргументов. В функции с возвращаемым значением должен присутствовать оператор `return`, завершающий выполнение функции, и переслать возвращаемое значение в вызывающую функцию. Вот почему в конце функции `main()` стоит следующий оператор:

```
return 0;
```

Логически это правильно: функция `main()` должна возвращать значение типа `int`, и данный оператор посылает целое число 0. Но у вас может возникнуть вопрос: куда возвращается это значение? В конце концов, в вашей программе функция `main()` не вызывалась:

```
squeeze = main(); // в наших программах
                   // отсутствует
```

Нетрудно догадаться, что программу вызывает операционная система (скажем, UNIX или DOS). И следовательно, функция `main()` возвращает значение не в другую часть этой же программы, а в операционную систему. Возвращаемые значения программ применяются во многих операционных системах. Например, были разработаны скрипты интерпретатора команд UNIX и пакетные файлы DOS, чтобы запускать программы и проверять их возвращаемые значения, обычно называемые выходными значениями выхода. Обычно принято, что нулевое возвращаемое значение означает успешное выполнение программы, тогда как ненулевое значение означает, что при выполнении программы были какие-то проблемы. Таким образом, можно написать программу C++, которая будет возвращать ненулевое значение в случае, если ей, скажем, не удалось открыть файл. Скрипт интерпретатора команд или пакетный файл можно спроектировать таким образом, что если после запуска программа сигнализирует о неуспешном выполнении (аварийном окончании), то они предпринимают какие-либо альтернативные действия.

КЛЮЧЕВЫЕ СЛОВА

Ключевые слова — это словарь языка программирования. В этой главе использовались четыре ключевых слова языка C++: `int`, `void`, `return` и `double`. Поскольку эти слова имеют в языке C++ специальное значение, их нельзя ис-

пользовать для других целей, т.е. слово `return` нельзя использовать как имя переменной или слово `double` — как имя функции. Но они могут быть составной частью имени, например, `painter` (содержит в себе слово `int`) или `return_aces`. В приложении B приводится полный список ключевых слов языка C++. Между прочим, слово `main` не является ключевым, так как оно не является частью языка C++. Это имя обязательной для применения функции. Слово `main` можно использовать как имя переменной. (При некоторых обстоятельствах, слишком экзотических, чтобы описывать их здесь, это может вызвать проблемы. В любом случае это может привести к путанице и лучше так не делать.) Подобно этому, имена других функций и объектов не являются ключевыми словами. Однако, если использовать в программе одно и то же имя, скажем `cout`, как для объекта, так и для переменной, то это может сбить с толку компилятор. Другими словами, можно применять `cout` как имя переменной в той функции, которая для вывода данных не использует объект `cout`; но нельзя в одной и той же функции применять имя `cout` и для переменной, и для объекта.

Определяемая пользователем функция с возвращаемым значением

Давайте теперь сделаем еще один шаг вперед и напишем функцию, в которой используется оператор `return` (оператор возврата). Как это сделать, иллюстрирует функция `main()`: в заголовке функции необходимо указать тип возвращаемого значения, а в конце тела функции поставить оператор `return`. С помощью такой функции можно решить проблему взвешивания для тех, кто посещает Великобританию. Там многие напольные весы имеют шкалу, откалиброванную в *стоунах*, а не в фунтах США или килограммах. Один стоун равен 14 фунтам, и программа из листинга 2.6 содержит функцию, которая осуществляет такое преобразование.

Листинг 2.6 Программа convert.cpp.

```
// convert.cpp - преобразование стоунов
// в фунты
#include <iostream>
using namespace std;
int stonetolb(int); // прототип функции
int main()
{
    int stone;
    cout << "Enter the weight in stone: ";
    cin >> stone;
    int pounds = stonetolb(stone);
    cout << stone << " stone are ";
    cout << pounds << " pounds.\n";
    return 0;
}

int stonetolb(int sts)
{
    return 14 * sts;
}
```

В результате выполнения программы получаются следующие результаты:

Enter the weight in stone: 14
14 stone are 196 pounds.

С помощью объекта `cin` функция `main()` вводит значение для целочисленной переменной `stone`. Это значение передается функции `stonetolb()` как аргумент и в ней присваивается переменной `sts`. Затем функция `stonetolb()`, используя ключевое слово `return`, возвращает в функцию `main()` значение $14 * sts$. Это иллюстрирует тот факт, что за ключевым словом `return` не обязательно должно следовать простое число. Используя здесь более сложное выражение, вы избегаете необходимости создавать новую переменную и присваивать ей возвращаемое значение перед передачей вызывающей функции. Программа вычисляет значение этого выражения (в данном примере 196) и возвращает результирующее значение. Если вам не нравится возвращать значение выражения, можно избрать более длинный способ (метод):

```
int stonetolb(int sts)
{
    int pounds = 14 * sts;
    return pounds;
}
```

Оба варианта дают один и тот же результат, но второй вариант длиннее.

Вообще, функцию с возвращаемым значением можно использовать везде, где можно использовать простую константу того же самого типа. Например, функция `stonetolb()` возвращает значение типа `int`. Это означает, что функцию можно использовать следующими способами:

```
int aunt = stonetolb(20);
int aunt = aunt + stonetolb(10);
cout << "Ferdie weighs " << stonetolb(16)
     << " pounds.\n";
```

Здесь программа вычисляет возвращаемое значение и затем использует его в операторе.

Как видно из этих примеров, прототип функции описывает интерфейс функции, т.е. ее взаимодействие с остальной частью программы. Список аргументов показывает, какого вида информация передается в функцию, а тип функции — какого типа значение возвращает функция. Иногда программисты представляют функцию как *черный ящик* (этот термин взят из электроники), который точно описывается потоком информации, входящим в этот ящик и выходящим из него. Прототип функции идеально соответствует такому подходу (рис. 2.9).

Хотя функция `stonetolb()` является короткой и простой, в ней реализованы все функциональные особенности, присущие функциям:

- у нее есть заголовок функции и тело функции
- она принимает один аргумент
- она возвращает значение
- у нее должен быть прототип

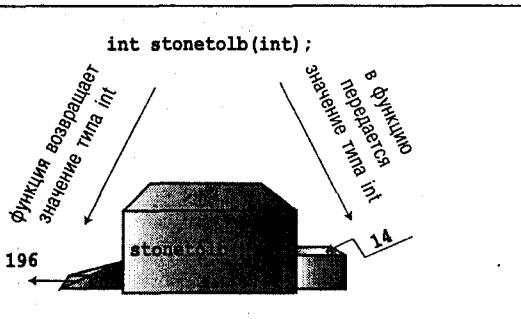


РИСУНОК 2.9 Функция в виде черного ящика и ее прототип.

Можно считать, что функция **stonetolb()** имеет стандартный формат функции. В главах 7 и 8 будет продолжено рассмотрение функций. А до того времени знания, полученные в настоящей главе, позволят вам чувствовать себя более спокойно и уверенно по отношению к функциям языка C++.

Итак, операторы...

В этой главе вы узнали о некоторых видах операторов C++ и их назначении

- Оператор объявления объявляет имя и тип переменной, используемой в функции.
- Оператор присваивания присваивает переменной некоторое значение.
- Оператор сообщения посылает в объект сообщение, инициируя тем самым некоторое действие.
- При вызове функции активизируется соответствующая функция. После того как выполнение функции завершается, программа возвращается в вызывающую функцию к оператору, следующему за вызовом функции.
- Прототип функции объявляет тип возвращаемого значения функции, а также число и тип аргументов, принимаемых функцией.
- Оператор возврата посылает возвращаемое значение из вызываемой функции назад в вызывающую функцию.

Резюме

Программа C++ состоит из одного или более модулей, называемых функциями. Выполнение программы начинается с функции, имеющей имя **main()**, поэтому в программе обязательно должна присутствовать функция с таким именем. Функция, в свою очередь, состоит из заголовка и тела функции. Заголовок функции содержит

информацию о том, есть ли у функции возвращаемое значение, и если есть, то какого оно типа, а также какого вида информация передается в функцию с помощью аргументов. Тело функции состоит из последовательности операторов C++, заключенных между парой фигурных скобок: {}.

Существуют следующие виды операторов C++: операторы объявления, операторы присваивания, операторы вызова функции, операторы сообщений для объектов и операторы возврата. Оператор объявления объявляет имя переменной и устанавливает, какого типа данные она может содержать. Оператор присваивания присваивает переменной некоторое значение. При вызове функции управление передается вызываемой функции. Когда функция завершает свою работу, управление передается оператору, непосредственно следующему за вызовом функции. Сообщение, посылаемое объекту, дает ему команду выполнить определенное действие. Оператор возврата — это механизм, с помощью которого функция возвращает значение в вызывающую функцию. Класс представляет собой определяемую пользователем спецификацию некоторого типа данных. В спецификации подробно описывается, каким образом должны быть представлены эти данные и какие операции могут выполняться над ними. Объект — это реальный экземпляр класса, создаваемый в соответствии со спецификацией класса, подобно тому как простая переменная создается в соответствии с описанием типа данных.

В языке C++ имеется два предопределенных объекта (**cin** и **cout**) для обработки входных и выходных данных. Они являются экземплярами соответственно классов **istream** и **ostream**, определения которых содержатся в файле **iostream**. В этих классах входные и выходные данные рассматриваются как потоки символов. Операция вставки (**<<**), определенная для класса **ostream**, позволяет вставлять данные в выходной поток, а операция извлечения (**>>**), определенная для класса **istream**, позволяет извлекать информацию из входного потока. Объекты **cin** и **cout** являются довольно "разумными" и способны преобразовывать информацию из одного вида в другой, в соответствии с контекстом программы.

В программах C++ может использоваться обширный набор библиотечных функций языка С. Чтобы использовать в программе библиотечную функцию, в программу необходимо включить заголовочный файл, содержащий прототип этой функции.

Теперь, когда вы имеете общее представление о простых программах C++, можно переходить к следующей главе, чтобы расширить горизонты своих знаний и детализировать их.

Вопросы для повторения

Ответы на вопросы для повторения из этой и последующих глав приведены в приложении J.

1. Как называются модули, из которых состоит программа C++?
2. Что делает следующая директива препроцессора:

```
#include <iostream>
```

3. Что делает следующий оператор:

```
using namespace std;
```

4. Какой оператор следует использовать, чтобы напечатать фразу "Hello, world" и перейти на начало следующей строки?
5. Какой оператор следует использовать, чтобы создать целочисленную переменную с именем `cheeses`?
6. Какой оператор следует использовать, чтобы присвоить значение 32 переменной `cheeses`?
7. Какой оператор следует использовать для ввода значения с клавиатуры и присвоения его переменной `cheeses`?
8. Какой оператор следует использовать, чтобы напечатать предложение "We have X varieties of cheese", где буква X будет заменяться текущим значением переменной `cheeses`?
9. Какую информацию о функции дает следующий заголовок функции:

```
int froop(double t)
```

10. В каком случае при определении функции не используется ключевое слово `return`?

Упражнения по программированию

1. Напишите программу C++, которая отображает на экране ваше имя и адрес.
2. Напишите программу C++, которая запрашивает расстояние в фарлонгах и преобразует его в ярды (один фарлонг равен 220 ярдам).
3. Напишите программу C++, которая состоит из трех определяемых пользователем функций (включая функцию `main()`) и выводит на экран следующие данные:

```
Three blind mice
Three blind mice
See how they run
See how they run
```

Одна функция, вызываемая два раза, должна отображать две первые строки, а другая, также вызываемая два раза, должна отображать остальные выходные данные.

4. Напишите программу, в которой функция `main()` вызывает определяемую пользователем функцию, принимающую в качестве аргумента температуру в градусах Цельсия и возвращающую эквивалентное значение в градусах Фаренгейта. По запросу программы температуру в градусах Цельсия вводит пользователь и затем программа отображает результат. Данные, выводимые на экран, имеют следующий вид:

```
Please enter a Celsius value: 20
20 degrees Celsius is 68 degrees
Fahrenheit.
```

Для справок: вот формула для выполнения преобразования:

```
Fahrenheit = 1.8 * Celsius + 32.0
```

Представление данных

В этой главе рассматривается следующее:

- Правила именования переменных языка C++
- Типы целочисленных данных языка C++: `unsigned long`, `long`, `unsigned int`, `int`, `unsigned short`, `short`, `char`, `unsigned char`, `signed char` и `bool`
- Файл `climits`, представляющий предельные значения для различных типов целочисленных данных в вычислительной системе
- Числовые константы различных типов целочисленных данных
- Применение квалификатора `const` для создания символических констант
- Встроенные типы данных с плавающей точкой языка C++: `float`, `double` и `long double`
- Файл `cfloat`, представляющий предельные значения для различных типов данных с плавающей точкой в вычислительной системе
- Числовые константы различных типов данных с плавающей точкой
- Арифметические операции языка C++
- Автоматическое преобразование типов данных
- Принудительное преобразование типов данных (приведение типов)

Сутью объектно-ориентированного программирования является разработка и модификация программистом собственных типов данных. Разработка нового типа данных представляет собой попытку создания такого типа данных, который бы лучше соответствовал требованиям конкретной задачи программирования. Выполнив эту задачу, вы убедитесь, что работать с данными стало намного проще. Но прежде чем вы сможете создавать свои собственные типы данных, необходимо изучить и понять готовые типы данных, имеющиеся в языке C++, поскольку вы будете создавать программы, используя именно их.

В языке C++ имеется две группы типов данных: основные и производные. В настоящей главе вы познакомитесь с основными типами данных, которые служат для представления целых чисел и чисел с плавающей точкой. Сначала создается такое впечатление, как будто имеется всего лишь два основных типа данных; однако было признано, что ни один тип целочисленных данных и ни один тип данных с плавающей точкой не смогут удовлетворить все потребности, возникающие при разработке программ. Поэтому в языке C++ существует несколько типов целочисленных данных и несколько типов данных с плавающей точкой. Далее в главе 4 рассматривается несколько типов данных, которые являются производными от основных типов данных; к этим дополнитель-

ным производным типам относятся массивы, строки, указатели и структуры.

Конечно, программе также необходимы средства для идентификации хранимых данных. Мы рассмотрим один метод выполнения этой задачи — применение переменных. Затем будет рассмотрено, как в языке C++ выполняются арифметические действия, и наконец, — как в языке C++ осуществляется преобразование данных одного типа в данные другого типа.

Простые переменные

Как правило, программы должны хранить какую-либо информацию, касающуюся, например, текущей цены акций компании IBM, средней влажности воздуха в Нью-Йорке в августе, наиболее часто встречающейся буквы в конституции и относительной частоты ее употребления или количества имеющихся двойников Элвиса Пресли. Для хранения в компьютере элемента информации программа должна отслеживать три основных свойства этого элемента информации, в частности, она определяет:

- где хранится информация
- какое значение там хранится
- какой вид информации хранится

Ранее в примерах программ использовался следующий способ: объявление переменной. Используемый в объявлении тип данных описывает вид информации, а имя переменной является символическим представлением значения переменной. Предположим, например, что старший лаборант Игорь использует следующие операторы:

```
int braincount;
braincount = 5;
```

Эти операторы означают, что программа запоминает целое число и что имя `braincount` представляет это целое число, в данном случае 5. В сущности, программа выделяет область памяти, достаточно большую, чтобы хранить целое число, отмечает ее, присваивает этой области метку `braincount` и копирует в нее значение 5. Эти операторы не сообщают вам (или Игорю), где в памяти хранится значение, но программе известно местоположение этой информации. Действительно, используя операцию `&`, можно извлечь адрес переменной `braincount` из памяти. Вы столкнетесь с этой операцией в следующей главе, когда будете изучать второй способ идентификации данных — с помощью указателей.

Имена переменных

Для переменных рекомендуется выбирать имена, имеющие смысл. Если переменная представляет стоимость поездки, назовите ее `costoftrip` или `costOfTrip`, а не просто `x` или `cot`. Вы должны следовать некоторым простым правилам именования переменных в языке C++:

- В именах можно использовать только следующие символы: буквы алфавита, цифры и символ подчеркивания (`_`).
- Первый символ имени не может быть цифрой.
- Символы верхнего и нижнего регистров рассматриваются как разные.
- В качестве имен нельзя использовать ключевые слова языка C++.
- Имена, начинающиеся с двух символов подчеркивания или с символа подчеркивания и следующей за ним буквы верхнего регистра, зарезервированы для использования реализацией языка. Имена, начинающиеся с символа подчеркивания, зарезервированы для использования реализацией языка в качестве глобальных идентификаторов.
- В языке C++ на длину имени не накладывается никаких ограничений и учитывается значение каждого символа имени.

Предпоследний пункт стоит в стороне от предшествующих пунктов; здесь говорится о том, что использование имени, подобного `_time_stop` или `Donut`, не приводит к появлению ошибки компилятора; поведение компилятора в этом случае не определено. Другими словами, ничего не говорится о том, каким должен быть правильный результат. Отсутствие ошибки компилятора связано с тем, что эти имена не являются недействительными, а зарезервированы для использования будущей реализацией языка. Что же касается глобальных имен, то здесь имеется в виду то, где объявляются имена; эта тема подробно рассматривается в главе 4. Последний пункт означает, что язык C++ не соответствует стандарту ANSI C, устанавливающему, что только первые 31 символ имеют значение. (Согласно стандарту ANSI C, два имени считаются идентичными, если у них одинаковые 31 символ, даже если 32-е символы различаются.)

Вот несколько действительных и недействительных имен языка C++:

```
int poodle;           // действительное
int Poodle;          // действительное и отличное
                     // от имени poodle
int POODLE;          // действительное и еще более
                     // отличное от имени poodle
Int terrier;         // недействительное -
                     // должно быть ключевое
                     // слово int, а не Int
int my_stars3;        // действительное
int _Mystars3;       // действительное, но
                     // зарезервированное -
                     // начинается с символа
                     // подчеркивания
int 4ever;           // недействительное, так как
                     // начинается с цифры
int double;          // недействительное - double
                     // является ключевым словом
                     // языка C++
int begin;           // действительное - begin
                     // является ключевым словом
                     // языка Pascal
int __fools;          // действительное, но
                     // зарезервированное -
                     // начинается с двух
                     // символов подчеркивания
int the_very_best_variable_i_can_be_version_112;
                     // действительное
int honky-tonk;       // недействительное -
                     // дефис не допускается
```

Если требуется образовать имя из двух или более слов, то обычно слова разделяются символами подчеркивания, например: `my_onions`, или каждое слово, кроме первого, пишется с заглавной буквы, например: `myEyeTooth`. (Ветераны программирования на языке C склонны использовать символы подчеркивания, что яв-

ляется традицией языка C, тогда как программисты, пишущие на языке Pascal, предпочитают использовать заглавные буквы.) Каждый из этих методов позволяет выделять отдельные слова и отличать, скажем, `carDrip` от `cardRip` или `boat_sport` от `boats_port`.

Целочисленные типы данных

Целые числа — это числа без дробной части, например, 2, 98, -5286 и 0. В природе существует огромное количество целых чисел (это количество является бесконечным), поэтому ни в какой самой большой оперативной памяти компьютера не могут быть представлены все возможные целые числа. Следовательно, в любом языке программирования может быть представлена только часть всех целых чисел. В некоторых языках программирования, например в языке Standard Pascal, существует только один тип целочисленных данных (один тип данных служит для представления всех целых чисел!), но в языке C++ имеется несколько типов целочисленных данных. Это позволяет программистам выбирать такой тип целочисленных данных, который лучше всего соответствует требованиям конкретной программы. Подобная забота о том, чтобы тип данных соответствовал конкретным требованиям, облегчает создание новых типов данных в ООП.

В языке C++ различные типы целочисленных данных различаются объемом памяти, используемым для их хранения. Большой блок памяти может представлять больший диапазон целых чисел. Кроме того, одни типы данных (типы данных со знаком) могут представлять и положительные, и отрицательные значения, тогда как другие типы данных (типы данных без знака) не могут представлять отрицательные значения. Основные типы целочисленных данных языка C++ (в порядке возрастания размера) именуются `char`, `short`, `int` и `long`. Каждый из этих типов данных подразделяется на две разновидности: со знаком и без знака. В результате вы имеете на выбор восемь различных типов целочисленных данных! Давайте рассмотрим эти типы целочисленных данных более подробно. Поскольку тип данных `char` обладает некоторыми особыми свойствами (чаще всего он используется для представления символов, а не чисел), в настоящей главе сначала рассматриваются другие типы целочисленных данных.

Типы данных `short`, `int` и `long`

Оперативная память компьютера состоит из элементов, называемых **битами**. (См. примечание "Биты и байты".) Располагая различным числом битов для хранения значений, типы данных `short`, `int` и `long` могут представлять целые числа трех разных величин. Было бы удобно, если бы данные каждого типа во всех системах были одной

определенной величины; например, величина данных типа `short` всегда была бы равна 16 битам, типа `int` — 32 битам и т.д. Но не все так просто. А все потому, что ни один вариант не подходит для всех типов компьютеров. В языке C++ предлагается гибкий стандарт с некоторыми гарантированными минимальными размерами:

- Величина данных типа `short` не меньше 16 битов.
- Величина данных типа `int` не меньше размера данных типа `short`.
- Величина данных типа `long` не меньше 32 битов и не меньше размера данных типа `int`.

БИТЫ И БАЙТЫ

Основным элементом памяти компьютера является **бит**. Представьте себе бит в виде электронного переключателя, который можно либо включить, либо выключить. Если переключатель отключен, то он представляет значение 0, а если включен — то значение 1. Элемент памяти из 8 битов может принимать 256 различных значений. Число 256 получается потому, что каждый бит имеет два возможных значения, и тогда общее число возможных значений для 8 битов равно 2^8 , или 256. Таким образом, 8-битовый элемент памяти может представлять, скажем, значения в диапазоне от 0 до 255 или в диапазоне от -128 до 127. Каждый дополнительный бит удваивает число возможных значений. Это означает, что 16-битовый элемент памяти может принимать 65 536 различных значений, а 32-битовый элемент — 4 294 672 296 различных значений. **Байт** — это обычно 8-битовый элемент памяти. В этом смысле байт является единицей измерения, описывающей объем памяти компьютера; при этом один килобайт равен 1024 битам, а один мегабайт — 1024 Кб. Однако в языке C++ байт определяется иначе. В языке C++ байт состоит из такого числа смежных битов, которого достаточно для того, чтобы вместить основной набор символов для данной реализации языка. Другими словами, число возможных значений должно быть равно или превышать число различных символов. В США основными наборами символов являются, как правило, наборы символов ASCII или EBCDIC, для представления которых достаточно 8 битов. Поэтому в системах, где используются эти наборы символов, один байт языка C++ равен 8 битам. Однако при программировании для других стран могут потребоваться расширенные наборы символов, такие как Unicode, поэтому в некоторых реализациях могут использоваться 16-битовые байты.

В настоящее время во многих системах используются минимальные гарантированные размеры данных и данные типа `short` имеют величину, равную 16 битам, а данные типа `long` — величину, равную 32 битам. Для данных типа `int` остается возможность выбора. Их величина может быть равна 16, 24 или 32 битам и при этом соответствовать стандарту. В более ранних реализациях языка C++ на компьютерах IBM PC данные типа `int` имеют, как правило, величину 16 битов (такую же, как и у данных типа `short`); в Windows 95, Windows 98, Windows NT, Macintosh, VAX и во многих других реа-

лизациях языка C++ для мини-компьютеров данные типа `int` имеют величину 32 бита (такую же, как и данные типа `long`). В некоторых реализациях вам предоставляется возможность выбирать величину данных типа `int`. (Какая величина используется в вашей реализации? В следующем примере показано, как определить предельные значения величин данных для вашей системы, не заглядывая в учебник.) При переносе программы C++ из одной среды в другую могут возникнуть проблемы, связанные с разными величинами типов данных в различных реализациях. Но, как будет показано далее в этой главе, небольшие меры предосторожности могут свести эти проблемы к минимуму.

При объявлении переменных имена данных этих типов используются точно так же, как имена данных типа `int`:

```
short score;           //создает переменную
                      //типа short int
int temperature;      //создает переменную
                      //типа int
long position;        //создает переменную
                      //типа long int
```

В действительности, `short` — это сокращение от `short int`, а `long` — сокращение от `long int`, но вряд ли кто-нибудь использует расширенные формы записи.

Данные этих трех типов: `int`, `short` и `long` — являются данными со знаком, а это означает, что диапазон их возможных значений разделяется приблизительно поровну между положительными и отрицательными значениями. Например, 16-битовые данные типа `int` могут лежать в диапазоне от -32768 до +32767.

Если вам требуется узнать величины целых чисел в конкретной системе, то это можно сделать с помощью

Листинг 3.1 Программа `limits.cpp`.

```
// limits.cpp — некоторые предельные значения для целых чисел
#include <iostream>
using namespace std;
#include <climits>           // в устаревших системах используйте заголовочный файл limits.h
int main()
{
    int n_int = INT_MAX;      // инициализация переменной n_int максимальным значением типа int
    short n_short = SHRT_MAX; // символы определены в файле limits.h
    long n_long = LONG_MAX;

    // операция sizeof выдает размер данных какого-либо типа или переменной
    cout << "int is " << sizeof (int) << " bytes.\n";
    cout << "short is " << sizeof n_short << " bytes.\n";
    cout << "long is " << sizeof n_long << " bytes.\n\n";

    cout << "Maximum values:\n";
    cout << "int: " << n_int << "\n";
    cout << "short: " << n_short << "\n";
    cout << "long: " << n_long << "\n\n";

    cout << "Minimum int value = " << INT_MIN << "\n";
    return 0;
}
```

программных инструментов, имеющихся в языке C++. Во-первых, операция `sizeof` возвращает размер данных какого-либо типа или размер переменной (в байтах). (Оператор (операция) — это элемент языка, с помощью которого выполняются определенные действия над одним или несколькими элементами данных и в результате получается некоторое значение. Например, операция сложения, представленная знаком +, вызывает сложение двух чисел. Обратите внимание, что значение термина "байт" зависит от реализации, поэтому двухбайтовые данные типа `int` могут иметь величину 16 битов в одной системе и 32 бита — в другой). Во-вторых, заголовочный файл `climits` (в более ранних реализациях — заголовочный файл `limits.h`) содержит информацию о предельных значениях целочисленных данных различных типов. В частности, в нем определены символические имена для представления различных предельных значений. Например, в этом файле определена константа `INT_MAX`, которая является самым большим возможным значением данных типа `int`. В программе из листинга 3.1 демонстрируется, как использовать описанные средства. Там же иллюстрируется использование инициализации, которая заключается в присваивании переменной некоторого значения.

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Заголовочный файл `climits` — это используемая в языке C++ версия заголовочного файла `limits.h`, определенного в стандарте ANSI C. Некоторые устаревшие платформы C++ не использовали заголовочные файлы вообще. Если вы пользуетесь такой системой, то вам придется ограничиться мысленным выполнением этого примера.

В результате выполнения программы на платформе Microsoft Visual C++ 5.0 получились следующие результаты:

```
int is 4 bytes.
short is 2 bytes.
long is 4 bytes.

Maximum values:
int: 2147483647
short: 32767
long: 2147483647

Minimum int value = -2147483648
```

Результаты при выполнении программы на другой платформе (Borland C++ 3.1 для DOS):

```
int is 2 bytes.
short is 2 bytes.
long is 4 bytes.

Maximum values:
int: 32767
short: 32767
long: 2147483647

Minimum int value = -32768
```

Примечания к программе

В данном разделе перечислены главные особенности данной программы.

Оператор `sizeof` выдает информацию о том, что тип данных `int` имеет величину 4 байта в базовой системе, в которой используется 8-битовый байт. Оператор `sizeof` можно использовать как с именем типа данных, так и с именем переменной. Когда оператор `sizeof` используется с именем типа данных, например с `int`, имя заключается в круглые скобки. Но, если оператор используется с именем переменной, например с `n_short`, круглые скобки необязательны:

```
cout << "int is " << sizeof (int)
     << " bytes.\n";
cout << "short is " << sizeof n_short
     << " bytes.\n";
```

В заголовочном файле `climits` определены символические константы (как указано в примечании "Символические константы"), представляющие предельные значения для данных различных типов. Как уже упоминалось, константа `INT_MAX` представляет максимально возможное значение, которое может принимать тип данных `int`; в нашей системе DOS это будет 32767. Изготовитель поставляет с компилятором такой файл `climits`, в котором отражены значения, соответствующие данному компилятору. Например, для системы с 32-битовыми данными типа `int` в файле `climits` определена константа `INT_MAX`, равная 2147483647. В табл. 3.1 приведены символические константы, определенные в этом файле; некоторые из них относятся к типам данных, которые вы еще не изучали.

Таблица 3.1 Символические константы из файла `climits`.

Символическая константа	Представляет
<code>CHAR_BIT</code>	Число битов для типа данных <code>char</code>
<code>CHAR_MAX</code>	Максимальное значение типа данных <code>char</code>
<code>CHAR_MIN</code>	Минимальное значение типа данных <code>char</code>
<code>SCHAR_MAX</code>	Максимальное значение типа данных <code>signed char</code>
<code>SCHAR_MIN</code>	Минимальное значение типа данных <code>signed char</code>
<code>UCHAR_MAX</code>	Максимальное значение типа данных <code>unsigned char</code>
<code>SHRT_MAX</code>	Максимальное значение типа данных <code>short</code>
<code>SHRT_MIN</code>	Минимальное значение типа данных <code>short</code>
<code>USHRT_MAX</code>	Максимальное значение типа данных <code>unsigned short</code>
<code>INT_MAX</code>	Максимальное значение типа данных <code>int</code>
<code>INT_MIN</code>	Минимальное значение типа данных <code>int</code>
<code>UINT_MAX</code>	Максимальное значение типа данных <code>unsigned int</code>
<code>LONG_MAX</code>	Максимальное значение типа данных <code>long</code>
<code>LONG_MIN</code>	Минимальное значение типа данных <code>long</code>
<code>ULONG_MAX</code>	Максимальное значение типа данных <code>unsigned long</code>

Инициализация представляет собой объединение присваивания значения с объявлением. Например, оператор

```
int n_int = INT_MAX;
```

объявляет переменную `n_int` и присваивает ей максимальное значение, возможное для типа данных `int`. Для инициализации значения переменных можно также использовать стандартные константы. Инициализировать переменную можно, присваивая ей значение другой переменной, при условии, что эта переменная была определена раньше. Можно даже инициализировать переменную значением выражения при условии, что значения всех компонентов выражения известны во время компиляции:

```
int uncles = 5; //инициализирует переменную
                  //uncles значением 5
int aunts = uncles; //инициализирует
                  //переменную aunts значением 5
int chairs = aunts + uncles + 4;
                  //инициализирует переменную
                  //chairs значением 14
```

Если объявление переменной `uncles` переместить в конец этого списка операторов, то инициализации переменных `aunts` и `chairs` станут недействительными, поскольку значение переменной `uncles` не будет известно в то время, когда компилятор будет пытаться проинициализировать эти две переменные.

ПОМНИТЕ

Если вы не инициализируете переменную, определяемую внутри функции, то значение переменной будет **неопределенным**. Это означает, что значением переменной будет то, что находилось в этой области памяти до объявления переменной.

Если вы знаете, каким должно быть начальное значение переменной, инициализируйте ее. Правда, разделение объявления переменной и присваивания ей значения может вызвать кратковременную неопределенность:

```
short year; //какое значение может
//быть у этой переменной?
year = 1492; //а-а, теперь ясно
```

Кроме того, инициализируя переменную при ее объявлении, вы уже не забудете проинициализировать переменную в дальнейшем.

СОЗДАНИЕ СИМВОЛИЧЕСКИХ КОНСТАНТ С ПОМОЩЬЮ ДИРЕКТИВЫ ПРЕПРОЦЕССОРА

Файл `climits` содержит строки, подобные следующей:

```
#define INT_MAX 32767
```

Вспомните, что в процессе компиляции программ C++ исходный код сначала обрабатывается препроцессором. Здесь строка `#define`, так же как и `#include`, является директивой препроцессора. Она дает препроцессору следующую команду: все имена `INT_MAX` в программе заменить числами `32767`. Следовательно, директива `#define` функционирует как глобальная команда "найти-и-заменить" в текстовом редакторе. После того как будут осуществлены эти замены, происходит компиляция измененной программы. Препроцессор отыскивает независимые лексемы (отдельные слова), пропуская встроенные слова, т.е. он не заменяет `PINT_MAXIM` на `P32767IM`. Директиву `#define` можно также использовать для определения своих собственных констант (листинг 3.2). Однако директива `#define` – это пережиток прошлого, доставшийся в наследство от языка С. Язык C++ предоставляет лучший

способ создания символьических констант (ключевое слово `const`, рассматриваемое в настоящей главе), так что вам не придется часто пользоваться директивой `#define`. Но она применяется в некоторых заголовочных файлах, предназначенных как для С, так и для C++.

Типы данных без знака

Для каждого из трех только что рассмотренных типов данных имеется его разновидность: тип данных без знака. Данные этих типов не могут принимать отрицательные значения. Преимуществом здесь является то, что увеличивается максимально возможное значение данных. Например, тип данных `short` представляет данные, лежащие в диапазоне от -32768 до +32767, тогда как разновидность этого типа данных без знака может представлять данные, лежащие в диапазоне от 0 до 65535. Конечно, типы данных без знака можно использовать только для величин, которые никогда не бывают отрицательными и которые касаются, например, населения, чисел в инвентаризационной ведомости и количества осадков в месяц. Чтобы создать разновидность целочисленной переменной без знака, просто модифицируйте объявление с помощью ключевого слова `unsigned`:

```
unsigned short change; //переменная типа
//unsigned short
unsigned int rovert; //переменная типа
//unsigned int
unsigned quarterback; //также переменная
//типа unsigned int
unsigned long gone; //переменная типа
//unsigned long
```

Обратите внимание, что ключевое слово `unsigned` является сокращением от `unsigned int`.

В листинге 3.2 иллюстрируется использование типов данных без знака.

Листинг 3.2 Программа exceed.cpp.

```
// exceed.cpp – предпринимается попытка выйти за границы диапазонов значений целых чисел
#include <iostream>
using namespace std;
#define ZERO 0 // определяет, что символ ZERO имеет значение 0
#include <climits> // определяет константу INT_MAX как самое большое значение типа int
int main()
{
    short sam = SHRT_MAX; // инициализирует переменную максимально возможным значением
    unsigned short sue = sam; // все в порядке, если переменная sam уже определена
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nAdd $1 to each account.\nNow ";
    sam = sam + 1;
    sue = sue + 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nPoor Sam!\n";
    sam = ZERO;
    sue = ZERO;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\n";
    cout << "Take $1 from each account.\nNow ";
    sam = sam - 1;
    sue = sue - 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nLucky Sue!\n";
    return 0;
}
```

В нем также показано, что может произойти, если в программе будет предпринята попытка выйти за границы диапазона возможных значений для целочисленных данных какого-либо типа. И наконец, в этом листинге вы можете в последний раз увидеть директиву препроцессора `#define`.

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В программе из листинга 3.2, так же как и в программе из листинга 3.1, используется заголовочный файл `climits`; в компиляторах более старых выпусков вместо него может быть использован файл `limits.h`, а в компиляторах очень старых выпусков может не быть ни того, ни другого файла.

В результате выполнения программы выводятся следующие данные:

```
Sam has 32767 dollars and Sue has 32767
→dollars deposited.
Add $1 to each account.
Now Sam has -32768 dollars and Sue has
→32768 dollars deposited.
Poor Sam!
Sam has 0 dollars and Sue has 0 dollars
→deposited.
Take $1 from each account.
Now Sam has -1 dollars and Sue has 65535
→dollars deposited.
Lucky Sue!
```

В этой программе переменной `sam` типа `short` и переменной `sue` типа `unsigned short` присваивается самое большое значение для данных типа `short`, которое в нашей системе равно 32767. Затем значение каждой переменной увеличивается на единицу. Для переменной `sue` это не вызывает никаких проблем, поскольку новое значение намного меньше максимально допустимого для целого числа без знака. Но значение переменной `sam` изменяется с 32767 на -32768! Аналогично этому вычитание 1 из 0 не вызывает никаких проблем в отношении переменной `sam`, но значение беззнаковой переменной `sue` изменяется с 0 на 65535. Как видите, это во многом подобно тому, как изменяются значения спидометра автомобиля или счетчика видеомагнитофона. Если вы выходите за границу диапазона, то одно граничное значение заменяется другим граничным значением (с противоположного конца диапазона) (рис. 3.1). В языке C++ гарантируется, что данные без знака ведут себя именно таким образом. Хотя в языке C++ не гарантируется, что выход за границы диапазона (переполнение или потеря значимости) для целочисленных данных со знаком будет происходить без выдачи сообщения об ошибке, именно так наиболее часто работают современные реализации языка C++.

Какой тип выбрать?

В языке C++ имеется такое богатство целочисленных типов данных, что может возникнуть вопрос: какой тип данных следует использовать в данном конкретном случае? Обычно для данных типа `int` выбирается величина, которая является наиболее "естественной" для компьютеров, для которых предназначается реализация языка C++. *Естественная величина* означает, что целые числа подобного типа обрабатываются компьютером наиболее эффективно. Если отсутствуют причины, побуждающие выбирать какой-либо иной тип данных, то используйте тип данных `int`.

А теперь рассмотрим причины, по которым может потребоваться использовать какой-либо другой тип данных. Если переменная представляет величину, которая никогда не бывает отрицательной, например число слов в документе, то можно использовать тип данных без знака; таким образом, переменная может иметь больший диапазон возможных значений.

Если вы знаете, что переменная может принимать значения, превышающие 16-битовое целое число, то используйте тип данных `long`. Поступайте так даже в том случае, если в вашей системе данные типа `int` имеют величину 32 бита. Тогда при переносе программы в систему, в которой заданы 16-битовые данные типа `int`, она может выполняться некорректно (рис. 3.2).

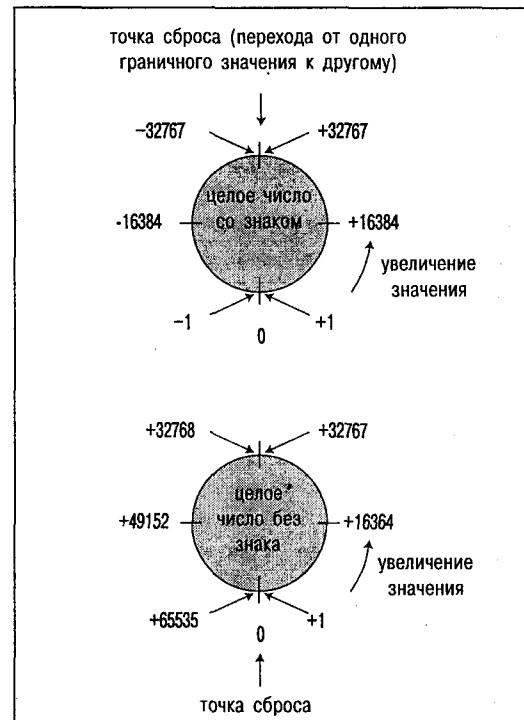
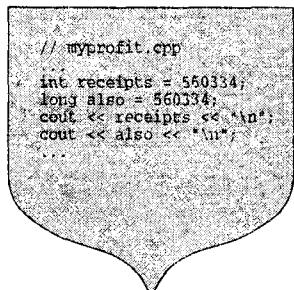
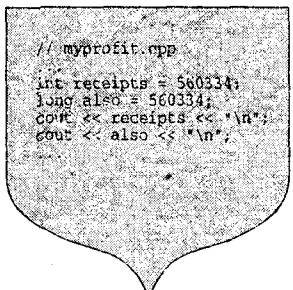


РИСУНОК 3.1 Типичные действия реализации языка C++ в случае возникновения переполнения для целочисленных данных.



560334
560334

На этом компьютере программа, использующая тип данных `int`, работает правильно.



-29490
560334

На этом компьютере программа, использующая тип данных `int`, не работает (выполняется некорректно).

РИСУНОК 3.2 Чтобы программа была переносимой, для представления больших чисел используйте данные типа `long`.

Если тип данных `short` имеет меньшую величину, чем тип данных `int`, то использование типа данных `short` позволит сэкономить память. Как правило, это важно только тогда, когда вы работаете с большими массивами целых чисел. (*Массив* — это структура данных, состоящая из нескольких значений одного типа, которые хранятся в памяти последовательно.) Если экономия памяти имеет важное значение, то вместо данных типа `int`, следует использовать данные типа `short`, даже если они имеют один и тот же размер. Предположим, например, что вы переносите программу из операционной системы DOS, которая использует 16-битовый тип данных `int` в операционную систему Windows NT, работающую с 32-битовым типом данных `int`. В этом случае объем памяти, необходимый для хранения массива типа `int`, увеличивается вдвое, а объем памяти, требуемый для хранения массива типа `short`, остается прежним. Помните, что, так же как и в случае с деньгами, сэкономленный бит — это заработанный бит.

Если для представления данных требуется только один байт, то можно использовать тип данных `char`. Эта возможность будет вскоре рассмотрена.

Целочисленные константы

Целочисленная константа — это константа, записываемая явно, например: 212 или 1776. В языке C++, так же как и в языке С, целочисленные константы могут записываться в трех системах счисления: в десятичной системе счисления (самая популярная система), восьмерич-

ной системе счисления (фаворит системы UNIX старого выпуска) и в шестнадцатиричной системе счисления (любимая система хакеров, стремящихся получить доступ к данным на аппаратном уровне). Эти системы счисления описываются в приложении А; здесь мы рассмотрим, как в языке C++ представлены числа для этих трех систем счисления. Чтобы определить, к какой системе счисления относится числовая константа, в языке C++ используется одна или две первые цифры константы. Если первая цифра находится в диапазоне 1–9, то число является десятичным; так, число 93 является десятичным. Если первая цифра равна 0, а вторая находится в диапазоне от 1 до 7, то число является восьмеричным; так, число 042 — это восьмеричное число, эквивалентное десятичному числу 34. Если первыми двумя символами являются 0x или 0X, то речь идет о шестнадцатиричном числе; так, число 0x42 — это шестнадцатиричное число, эквивалентное десятичному числу 66. В шестнадцатиричных числах символы a–f и A–F обозначают соответственно шестнадцатиричные цифры 10–15. Шестнадцатиричное число 0xF эквивалентно десятичному числу 15, а 0xA5 — десятичному числу 165 (10 умножить на 16 и прибавить 5, умноженное на 1). Программа из листинга 3.3 была написана специально, чтобы продемонстрировать использование этих трех систем счисления.

Листинг 3.3 Программа hexost.cpp.

```
// hexost.cpp — демонстрирует использование
// шестнадцатиричных и восьмеричных констант
#include <iostream>
using namespace std;
int main()
{
    int chest = 42; //десятичная целочисленная
                     //константа
    int waist = 0x42; //шестнадцатиричная
                      //целочисленная константа
    int inseam = 042; //восьмеричная
                      //целочисленная константа

    cout << "Monsieur cuts a striking figure!\n";
    cout << "chest = " << chest << "\n";
    cout << "waist = " << waist << "\n";
    cout << "inseam = " << inseam << "\n";
    return 0;
}
```

По умолчанию объект `cout` отображает на экране целые числа в десятичном виде, независимо от того, как они записаны в программе; это демонстрируют следующий результат выполнения программы:

```
Monsieur cuts a striking figure!
chest = 42
waist = 66
inseam = 34
```

Помните, что все эти системы счисления используются только лишь для удобства. Например, если вы про-

читали, что сегмент видеопамяти CGA эквивалентен шестнадцатиричному числу B000, то вам не требуется преобразовывать его в десятичное число 45056 перед использованием в своей программе. Вы можете просто записать 0xB000. Но независимо от того, в каком виде вы записываете число 10: как 10, 012 или 0xA, — оно сохраняется в компьютере одним и тем же способом: как двоичное число.

Между прочим, если вам требуется отобразить число в шестнадцатиричном или восьмеричном виде, то можно использовать специальные особенности объекта `cout`. Мы сейчас не будем рассматривать этот вопрос, но соответствующую информацию вы можете найти в главе 16. Вы можете бегло просмотреть эту главу, прочитать требуемую информацию и пропустить все остальное.)

Определение типа константы в C++

Из объявления переменной в программе компилятор C++ узнает тип целочисленной переменной. А как обстоит дело с константами? Предположим, что вы используете в программе константу (как число):

```
cout << "Year = " << 1492 << "\n";
```

Хранится ли число 1492 как тип данных `int`, `long` или как целочисленные данные какого-то другого типа? Ответ таков: в программах C++ целочисленные константы сохраняются как данные типа `int`, если нет каких-либо причин сделать иначе. Вот две такие причины: если используется специальный суффикс, указывающий конкретный тип данных, или же если число слишком большое, чтобы определяться типом данных `int`.

Сначала рассмотрим суффиксы. Это буквы, которые помещаются в конец числовой константы, чтобы указать тип данных. Суффикс `I` или `L` для целого числа означает, что оно является константой типа `long`, суффикс `u` или `U` указывает на константу типа `unsigned int`, а суффикс из двух букв, `ui` и `LI` (в любом порядке и для любого регистра, верхнего или нижнего), указывает на константу `unsigned long`. (Поскольку буква `I` нижнего регистра очень похожа на цифру `1`, то в суффиксах лучше использовать букву `L` верхнего регистра.) Например, в системе, для которой задан 16-битовый тип данных `int` и 32-битовый тип данных `long`, число 22022 сохраняется как 16-битовое число типа `int`, а число 22022L — как 32-битовое число типа `long`. Аналогично этому числа 22022LU и 22022UL являются константами типа `unsigned long`.

Теперь поговорим о величинах чисел. В языке C++ для десятичных чисел существуют немного иные правила, чем для шестнадцатиричных и восьмеричных. (Здесь термин "десятичный" означает, что основанием данной системы счисления является число 10, так же как основанием шестнадцатиричной системы счисления является

число 16; этот термин не предполагает обязательного наличия десятичной точки.) Десятичное целое число без суффикса будет представлено как тип данных `int`, `long` или `unsigned long`, а именно — как тип данных, который имеет наименьшую, но достаточную величину для представления этого числа. В вычислительной системе с 16-битовыми данными типа `int` и 32-битовыми данными типа `long` число 20000 будет представлено типом данных `int`, число 40000 — типом данных `long`, а число 3000000000 — типом данных `unsigned long`. Шестнадцатиричное или восьмеричное число без суффикса будет представлено типом данных `int`, `unsigned int`, `long` или `unsigned long`, т.е. тем типом, который обладает наименьшей, но достаточной величиной для представления этого числа. В той же самой вычислительной системе, где число 40000 будет представлено типом данных `long`, его шестнадцатиричный эквивалент 0x9C40 будет представлен типом данных `unsigned int`. Это связано с тем, что в шестнадцатиричном виде часто используются адреса памяти, которые по своей природе являются числами без знака. Поэтому тип данных `unsigned int` больше подходит для 16-битовых адресов, в этом случае лучше не применять тип данных `long`.

Тип данных `char`: символы и малые целые числа

Теперь пора вернуться к последнему упомянутому типу целочисленных данных — типу `char`. Как можно судить по его имени, тип данных `char` предназначен для представления символов, например, букв и цифр. Но если хранение чисел не является для компьютеров большой проблемой, то хранение символов — это совсем другое дело. В языках программирования был найден простой выход: для представления символов используется числовой код. Таким образом, тип данных `char` — это еще один тип целочисленных данных. Гарантируется, что данные этого типа имеют достаточные размеры для представления всего диапазона основных символов — всех букв, цифр, знаков препинания и т.п. — вычислительной системы, для которой предназначена реализация языка. На практике в большинстве систем основной набор символов содержит менее 256 символов, поэтому для его представления достаточно одного байта. Поэтому, хотя тип данных `char` чаще всего используется для представления символов, его также можно использовать для представления целых чисел, меньших, чем числа типа `short`.

Наиболее распространенным набором символов в США является набор символов ASCII, описанный в приложении С. Символы этого набора представлены числовым кодом (кодом ASCII). Например, 65 — это код символа A. Для удобства в примерах настоящей книги

используется код ASCII. Однако в реализации C++ используется любой код, который является естественным для данной вычислительной системы, например, код EBCDIC на больших ЭВМ фирмы IBM. Ни код ASCII, ни код EBCDIC не могут обслуживать потребности программистов в мировом масштабе, и в языке C++ имеется расширенный символьный тип данных, который способен представлять больший диапазон значений, например, используемый многоязыковым набором символов Unicode. Этот тип данных (`wchar_t`) мы рассмотрим далее в настоящей главе.

Работа с данными типа `char` демонстрируется в листинге 3.4.

Листинг 3.4 Программа chartype.cpp.

```
// chartype.cpp - применение данных типа char
#include <iostream>
using namespace std;
int main()
{
    char ch; //объявление переменной типа char
    cout << "Enter a character:\n";
    cin >> ch;
    cout << "Holla! ";
    cout << "Thank you for the "
        << ch << " character.\n";
    return 0;
}
```

Как обычно, запись `\n` в языке C++ означает символ новой строки. В результате выполнения программы получается следующий результат:

```
Enter a character:
M
Holla! Thank you for the M character.
```

Интересно здесь то, что вы вводите символ `M`, а не соответствующий код символа `77`. И программа также печатает символ `M`, а не код `77`. Однако, если вы "заглянете" в память, то обнаружите, что в переменной `ch` хранится значение `77`. Это чудесное превращение связано не с данными типа `char`, а с объектами `cin` и `cout`. Эти интересные конструкции выполняют преобразование от вашего имени. При вводе данных объект `cin` преобразует входные данные, являющиеся результатом нажатия клавиши `M`, в значение `77`. При выводе данных объект `cout` преобразует значение `77` в отображаемый символ `M`; функционирование объектов `cin` и `cout` определяет тип переменной. Если то же самое значение `77` присвоить переменной типа `int`, то объект `cout` отобразит его как `77` (т.е. объект `cout` отобразит два символа `7`). Этот момент иллюстрируется в программе из листинга 3.5. В ней также показано, как в языке C++ определяются символьные константы: символ заключается в одинарные кавычки, например, '`M`'. (Обратите внимание, что в этом примере программы не используются двой-

ные кавычки. В языке C++ одинарные кавычки применяются для задания символов, а двойные кавычки — для задания строк. Объект `cout` может обрабатывать и символы, и строки, но, как будет показано в главе 4, это две совершенно разные вещи.) И наконец, в программе применяется новое свойство объекта `cout` — функция `cout.put()`, которая отображает одинарный символ.

Листинг 3.5 Программа morechar.cpp.

```
// morechar.cpp - сопоставление данных
// типа char и типа int
#include <iostream>
using namespace std;
int main()
{
    char c = 'M'; //переменной с присваивается
                  //код ASCII для символа M
    int i = c;     //тот же код присваивается
                  //переменной типа int
    cout << "The ASCII code for " << c
        << " is " << i << "\n";
    cout << "Add one to the character code:\n";
    c = c + 1;
    i = c;
    cout << "The ASCII code for " << c
        << " is " << i << '\n';
    // использование функции cout.put() для
    // отображения константы типа char
    cout << "Displaying char c
          using cout.put(c): ";
    cout.put(c);
    // использование функции cout.put() для
    // отображения константы типа char
    cout.put('!');
    cout << "\nDone\n";
    return 0;
}
```

В ходе выполнения программы получается следующий результат:

```
The ASCII code for M is 77
Add one to the character code:
The ASCII code for N is 78
Displaying char c using cout.put(c): N!
Done
```

Примечания к программе

Запись '`M`' представляет числовой код для символа `M`, поэтому инициализация переменной `c` типа `char` значением '`M`' означает занесение в нее числа `77`. Затем идентичное значение присваивается переменной `i` типа `int`. Теперь обе переменные, `c` и `i`, содержат число `77`. Затем объект `cout` отображает переменную `c` как символ `M` и переменную `i` как число `77`. Как уже говорилось ранее, тип значения определяет функционирование объекта `cout`, когда он решает, каким образом отображать данное значение. Это еще один пример того, что `cout` — "разумный" объект.

Поскольку переменная `c` в действительности содержит целое число, по отношению к ней можно выполнять операции для целых чисел, например, прибавление единицы. В результате переменная `c` будет иметь значение 78. Затем это новое значение присваивается переменной `i`. (Можно было просто добавить 1 к переменной `i`.) Объект `cout` снова отображает это значение типа `char` как символ и значение типа `int` как число.

В языке C++ символы представлены в виде целых чисел, что очень удобно, так как упрощается выполнение действий со значениями символов. Не нужно пользоваться неудобными функциями преобразования символов в код ASCII и обратно.

Наконец, в программе используется функция `cout.put()` для отображения как переменной `c`, так и символьной константы.

Функция-элемент: `cout.put()`

Но что представляет собой функция `cout.put()` и почему в ее названии присутствует точка? Функция `cout.put()` знакомит вас с таким важным понятием объектно-ориентированного программирования, как *функция-элемент*. Класс, как вы помните, определяет способ представления данных и операции, которые можно выполнять над этими данными. Функция-элемент принадлежит классу и описывает метод выполнения действий с данными этого класса. В классе `ostream`, например, существует функция-элемент `put()`, предназначенная для вывода символов. Функции-элементы могут использоваться только отдельными объектами своего класса; в данном случае это объект `cout`. Чтобы использовать функцию-элемент класса при работе с таким объектом, как `cout`, имя объекта (`cout`) и имя функции (`put()`) объединяются с помощью точки. Эта точка называется *оператором принадлежности*. Запись `cout.put()` означает, что функция-элемент `put()` некоторого класса будет использоваться объектом `cout()` этого же класса. Конечно, вы изучите все это подробно, когда дойдете до рассмотрения классов в главе 9. Единственные классы, которые вы знаете пока, — это классы `istream` и `ostream`; вы можете поэкспериментировать с их функциями, чтобы лучше освоиться с новым понятием.

Функция-элемент `cout.put()` представляет собой альтернативу операции `<<`, используемой для отображения символа. В этом месте вы могли бы удивиться: зачем вообще нужна функция `cout.put()`? Ответ по большей части лежит в исторической плоскости. До появления версии 2.0 языка C++ объект `cout` отображал символьные переменные как символы, а символьные константы, например '`M`' и '`\n`' — как числа. Проблема заключалась в том, что в первых версиях языка C++, как и в языке C, символьные константы хранились как данные

типа `int`. Таким образом, код 77 для символа '`M`' хранился в 16- или 32-битовой ячейке памяти. А переменные типа `char` занимали, как правило, 8 битов. Такой оператор, как

```
char c = 'M';
```

копировал 8 битов (8 значимых битов) константы '`M`' в переменную `c`. К сожалению, это означало, что константа '`M`' и переменная `c` были для объекта `cout` совершенно разными величинами, хотя и содержали одно и то же значение. Поэтому такой оператор, как

```
cout << '$';
```

выводил не символ `$`, а его код ASCII. Но оператор

```
cout.put('$');
```

выводил символ так, как требовалось. В последующих версиях языка C++ (появившихся после версии 2.0) символьные константы хранятся как данные типа `char`, а не `int`. Поэтому в них объект `cout` обрабатывает символьные константы правильно. В качестве символа новой строки в языке C++ всегда можно было использовать строку "`\n`". Теперь для этой цели можно также использовать символьную константу '`\n`':

```
cout << "\n"; //используется строка
cout << '\n'; //используется символьная
//константа
```

Строка заключена не в одинарные, а в двойные кавычки и может содержать более одного символа. Строки, даже состоящие из одного символа, не являются данными типа `char`. Мы вернемся к рассмотрению строк в следующей главе.

Объект `cin` может считывать входные данные двумя разными способами. Это легче всего исследовать с помощью программы, в которой используется цикл для чтения нескольких символов. Поэтому мы вернемся к этой теме, когда рассмотрим циклы в главе 5.

Константы типа `char`

В языке C++ символьные константы можно записывать несколькими разными способами. Самый простой способ записи обычных символов, например букв, цифр, знаков препинания, — заключить символ в одинарные кавычки. Эта запись заменяет числовой код символа. Например, в системе, включающей код ASCII, имеются следующие соответствия:

- '`A`' соответствует 65, коду ASCII для буквы "A"
- '`a`' соответствует 97, коду ASCII для буквы "a"
- '`'5`' соответствует 53, коду ASCII для цифры "5"
- '`' '`' соответствует 32, коду ASCII для символа пробела
- '`'!`' соответствует 33, коду ASCII для восклицательного знака

Лучше использовать такую запись, чем явно записывать числовые коды. Она яснее и не привязана к конкретному коду. Если в системе используется код EBCDIC, то 65 не будет кодом для символа A, но запись 'A' всегда будет представлять данный символ.

Некоторые символы нельзя вводить в программу с клавиатуры непосредственно. Например, символ новой строки нельзя ввести, просто нажав клавишу Enter; программа текстового редактора интерпретирует нажатие этой клавиши как требование начать новую строку в вашем файле исходного кода. Трудности с записью других символов происходят потому, что в языке C++ им придается особое значение. Например, символ двойных кавычек ограничивает строки, поэтому его нельзя взять и просто вставить в середину строки. Как показано в табл. 3.2, для нескольких таких символов в языке C++ имеется специальная форма записи, которая называется *управляющей последовательностью*. Например, последовательность \a задает символ предупреждения, который вызывает звуковой сигнал. Последовательность \" трактует двойные кавычки как обычный символ, а не ограничитель строки. Такую форму записи можно использовать в строках или символьных константах:

```
char alarm = '\a';
cout << alarm
    << "Don't do that again!\a\n";
cout << "Ben \"Buggsie\" Hacker was here!\n";
```

Последняя строка дает такие выходные данные:

```
Ben "Buggsie" Hacker was here!
```

Обратите внимание, что управляющая последовательность, например \a, трактуется как обычный символ, например, Q. Таким образом, для создания символьной

константы управляющая последовательность заключается в одинарные кавычки, а внутри строки одинарные кавычки не заключаются.

Наконец, в управляющих последовательностях можно использовать восьмеричные и шестнадцатиричные коды символов. Например, сочетанию клавиш Ctrl+Z соответствует код ASCII 26, который соответствует 032 в восьмеричной системе и 0x1a — в шестнадцатиричной системе счисления. Этот символ может быть представлен любой из следующих управляющих последовательностей: \032 или \0x1a. Заключив эти управляющие последовательности в одинарные кавычки, их можно сделать символьными константами, например, "\032"; можно также включать их в строку, например, "hi\0x1a there".



СОВЕТ

Если вы стоите перед выбором — использовать числовую управляющую последовательность или символьскую, например, \0x8 или \b, — лучше используйте символьскую запись. Числовая запись привязана к конкретному коду, например, к коду ASCII, а символьская запись справедлива при любом коде и более понятна.

В программе из листинга 3.6 демонстрируется несколько управляющих последовательностей. Символ предупреждения используется, чтобы привлечь ваше внимание; символ новой строки вызывает продвижение курсора вперед; а символ возврата приводит к возврату курсора на одну позицию влево. (Гудини однажды нарисовал картину с рекой Гудзон, используя только управляющие последовательности; он, безусловно, был великим мастером в этой области.)

Таблица 3.2 Коды управляющих последовательностей языка C++.

Имя символа	Символ ASCII	Код C++	Десятичный код ASCII	Шестнадцатиричный код ASCII
Новая строка	NL (NF)	\n	10	0xA
Горизонтальная табуляция	HT	\t	9	0x9
Вертикальная табуляция	VT	\v	11	0xB
Возврат на одну позицию	BS	\b	8	0x8
Возврат каретки	CR	\r	13	0xD
Предупреждение	BEL	\a	7	0x7
Обратная наклонная черта	\	\\	92	0x5C
Знак вопроса	?	\?	63	0x3F
Одинарная кавычка	'	\'	39	0x27
Двойная кавычка	"	\"	34	0x22

Листинг 3.6 Программа bondini.cpp.

```
// bondini.cpp – использование управляющих последовательностей
#include <iostream>
using namespace std;
int main()
{
    cout << "\aOperation \"HyperHype\" is now activated!\n";
    cout << "Enter your agent code: _____\b\b\b\b\b\b\b\b\b";
    long code;
    cin >> code;
    cout << "\aYou entered " << code << "... \n";
    cout << "\aCode verified! Proceed with Plan Z3!\n";
    return 0;
}
```

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторых реализациях языка C++ с компиляторами, выпущенными до принятия стандарта ANSI, не распознается последовательность `\a`. В системах, где используется код ASCII, последовательность `\a` можно заменить последовательностью `\007`. В некоторых системах управляющие последовательности могут приводить к иным действиям, например, последовательность `\b` – к отображению маленького треугольника вместо возврата курсора или к стиранию символов при возврате.

Когда вы запускаете эту программу, она выводит на экран следующий текст:

```
Operation "HyperHype" is now activated!
Enter your agent code: _____
```

После печати символов подчеркивания программа с помощью символа возврата перемещает курсор назад на место первого символа подчеркивания. Теперь можно ввести секретный код и продолжить выполнение программы. Вот окончательные результаты:

```
Operation "HyperHype" is now activated!
Enter your agent code: 42007007
You entered 42007007.
Code verified! Proceed with Plan Z3!
```

Типы данных `signed char` и `unsigned char`

В отличие от данных типа `int`, данные типа `char` по умолчанию не являются ни данными со знаком, ни данными без знака. Право выбора остается за разработчиками реализации языка C++ с тем, чтобы этот тип данных лучше соответствовал конкретному аппаратному обеспечению. Если для вас жизненно важно, чтобы данные типа `char` вели себя определенным образом, то можно явно определять их как `signed char` или `unsigned char`:

```
char fodo;      //данные могут быть как
                //со знаком, так и без знака
unsigned char bar; //это определенно
                    //данные без знака
signed char snark; //это определенно
                    //данные со знаком
```

Эти различия особенно важны, если у вас данные типа `char` являются числовыми данными. Тип данных `signed char` имеет диапазон значений от -128 до 127, а тип `unsigned char` – от 0 до 255. Предположим, например, что вам необходимо присвоить переменной типа `char` значение 200. В одних системах эта операция пройдет успешно, а в других – нет. Однако для переменной типа `unsigned char` эта операция пройдет успешно в любой системе. С другой стороны, если переменная типа `char` служит для хранения стандартных символов кода ASCII, то неважно, имеет ли тип данных `char` знак или не имеет; и в одном, и в другом случае можно использовать переменную этого типа.

Тип данных `wchar_t`

Иногда программам приходится работать с символьными данными, величины которых больше, чем определяется 8-битовым байтом; например, японский набор символов (kanji). Справиться с такой ситуацией в языке C++ можно двумя способами. Во-первых, если большой набор символов является основным набором символов для данной реализации, то поставщик компилятора может определить, что размер данных типа `char` равен 16 битам или больше. Во-вторых, реализация языка может допускать работу как с основным, так и с расширенным набором символов. Обычные 8-битовые данные типа `char` могут использоваться для представления основного набора символов, а данные нового типа `wchar_t` – для представления расширенного набора символов. Данные типа `wchar_t` являются целочисленными данными, имеющими достаточную величину для представления самого большого расширенного набора символов, используемого в данной системе. Данные этого типа имеют такую же величину и знак, что и данные одного из целочисленных типов, который называется *базовым типом*. Выбор базового типа зависит от реализации, и в одной системе это может быть тип данных `unsigned short`, а в другой – тип `short` или какой-нибудь другой.

Семейство объектов `cin` и `cout` может осуществлять ввод и вывод потоков данных типа `char`, но тип данных `wchar_t` не воспринимается. В самой последней версии заголовочного файла `iostream` предлагаются параллельные программные конструкции в виде объектов `wcin` и `wcout` для обеспечения ввода и вывода потоков данных типа `wchar_t`. Кроме того, в языке C++ разрешается использовать расширенные символьные константы и строки. Эти конструкции указываются с помощью префикса L.

```
wchar_t bob = L'P'; //расширенная символьная
                     //константа
wcout << L"tall" << endl; //вывод
                            //расширенной символьной строки
```

Эти строки означают, что в системе с двухбайтовыми данными типа `wchar_t` каждый символ сохраняется в двухбайтовой ячейке памяти. В настоящей книге расширенные символьные данные не используются, но нужно иметь о них представление, так как, возможно, вам придется заниматься программированием в международном масштабе или использовать набор кодов Unicode.

НАБОР СИМВОЛОВ UNICODE

Набор символов Unicode решает проблему представления в языке C++ различных наборов символов. Он содержит стандартные числовые коды для большого числа символов и знаков, сгруппированных по типам. Например, код ASCII входит в состав набора символов Unicode как поднабор, так что латинские буквы английского алфавита имеют одинаковые коды в обеих системах. Но в Unicode входят также другие латинские буквы, используемые в европейских языках; буквы других алфавитов, включая греческий, кириллицу, еврейский, арабский, тайский и бенгальский; иероглифы китайского и японского языков. В настоящее время в Unicode входит свыше 38 тыс. символов, и он продолжает расширяться. Если вы хотите узнать об этом больше, обратитесь к Web-узлу консорциума Unicode по адресу www.unicode.org.

Новый тип данных `bool`

Стандарт ANSI/ISO языка C++ добавил новый тип данных `bool` (т.е. новый для языка C++). Он назван в честь английского математика Джорджа Буля (George Boole), который разработал математическое представление законов логики. В программировании *переменная Boolean* — это такая переменная, которая может принимать два значения: `true` (истина) или `false` (ложь). В прошлом в языке C++, так же как и в языке C, данные типа Boolean отсутствовали. Вместо этого, как вы увидите более подробно в главах 5 и 6, в языке C++ ненулевые значения интерпретировались как значение `true`, а нулевые значения — как значение `false`. Однако теперь для представления этих значений можно использовать тип данных `bool` и предопределенные литералы `true` и `false`, т.е. можно записывать операторы, подобные следующему:

3 Зак. 99

```
bool isready = true;
```

Литералы `true` и `false` могут быть преобразованы в данные типа `int` путем повышения типа, при этом `true` преобразуется в 1, а `false` — в 0:

```
int ans = true; //переменной ans
                 //присваивается 1
int promise = false; //переменной promise
                      //присваивается 0
```

Кроме того, любое числовое значение или значение указателя может быть преобразовано в значение типа `bool` неявно (т.е. без явного приведения типов). Любое ненулевое значение преобразуется в значение `true`, а нулевое значение — в значение `false`:

```
bool start = -100; //переменной start
                    //присваивается
                    //значение true
bool stop = 0;     //переменной stop
                    //присваивается
                    //значение false
```

В последующих главах иллюстрируется, как выполняется работа с этим типом данных.

Квалификатор `const`

А теперь вернемся к теме символьических имен для констант. Символическое имя может показывать, что собой представляет константа. Кроме того, если константа используется в программе в нескольких местах, то вы можете просто изменить одно определение символьической константы. В примечании об операторах `#define` ("Создание символьических констант с помощью директивы препроцессора"), приведенном ранее в настоящей главе, указывалось, что в языке C++ имеется лучший способ обработки символьических констант. Этот способ заключается в модификации объявления переменной и инициализации константы с помощью ключевого слова `const`. Предположим, например, что вам требуется символьическая константа, обозначающая число месяцев в году. Используйте в программе такую строку:

```
const int MONTHS = 12; //MONTHS — это
                       //символическая константа,
                       //обозначающая число 12
```

Теперь в программе вместо числа 12 можно использовать символьическое имя `MONTHS`. (Простое число 12 в программе может означать число дюймов в футе или количество пончиков в дюжине, но имя `MONTHS` прямо говорит о том, что оно представляет.) При инициализации константы ее значение устанавливается так же, как в случае с константой `MONTHS`. После этого компилятор не позволяет изменять значение константы `MONTHS`. Например, в реализации Borland C++ выдается сообщение об ошибке с информацией о том, что требуется `Lvalue`. Такое же сообщение появляется в том

случае, когда вы пытаетесь присвоить значение 4 значению 3. (*Lvalue* — это величина, подобная переменной, стоящей в левой части оператора присваивания.) Ключевое слово **const** называется *квалификатором*, так как оно квалифицирует смысл объявления.

Записывайте имя константы большими буквами, чтобы было видно, что это константа. Это никоим образом не универсальное правило, но оно помогает отличать константы от переменных при чтении программы. Используется и другое правило: только первая буква имени пишется заглавной. Третье правило состоит в том, что имя константы начинается с буквы "k", например, **kmonths**. Есть и другие правила. Во многих организациях имеются свои правила написания программ, которых придерживаются программисты.

Оператор создания константы имеет такой общий вид:

```
const type name = value;
```

Обратите внимание, что инициализация константы осуществляется вместе с ее объявлением. Следующая последовательность операторов некорректна:

```
const int toes;      //в этом операторе
                     //значение константы toes
                     //не определено
toes = 10;          //слишком поздно!
```

Если при объявлении константы вы не присваиваете ей значение, то она так и останется неопределенной, так как модифицировать ее после этого нельзя.

Если вы раньше программировали на языке C, то могли заметить, что рассмотренный ранее оператор **#define** выполняет ту же задачу. Но квалификатор **const** в некотором отношении лучше. Во-первых, он позволяет явно задавать тип константы. Во-вторых, используя такое понятие языка C++, как область действия, можно ограничивать действие констант отдельными функциями или файлами. (Область действия определяет, каким модулям известно о существовании константы; более подробно вы узнаете об этом в главе 8.) В-третьих, квалификатор **const** можно применять с более сложными типами данных, такими как массивы и структуры, которые будут рассматриваться в следующей главе.



СОВЕТ

Программисты, переходящие на язык C++ с языка C, для определения символьских констант склонны использовать оператор **#define**; лучше вместо него используйте квалификатор **const**.

В стандарте ANSI C также определен квалификатор **const**. Если вы знакомы с этим стандартом, то заметите, что в языке C++ квалификатор **const** определен немного иначе. Одно различие связано с областью действия; этот вопрос рассматривается в главе 8. Второе и главное

отличие заключается в том, что в языке C++ (но не в языке C) квалификатор **const** можно использовать для определения размера массива. Примеры этого вы увидите в следующей главе.

Числа с плавающей точкой

Теперь, когда вы познакомились со всеми типами целочисленных данных языка C++, давайте рассмотрим типы данных с плавающей точкой, которые составляют вторую большую группу основных типов данных языка C++. Эти типы данных позволяют представлять числа с дробными частями, с помощью которых, например, оценивается расстояние, которое проходит танк M1, расходуя один галлон бензина (0.56 мили на галлон). Они также обеспечивают намного больший диапазон значений. Если число слишком велико, чтобы быть представленным как тип данных **long**, например, число звезд в нашей галактике (по оценкам, приблизительно 400 000 000 000), можно использовать один из типов данных с плавающей точкой.

С помощью типов данных с плавающей точкой можно представлять такие числа, как 2.5, 3.14159 и 122442.32, т.е. числа с дробной частью. В компьютере такие числа хранятся в виде двух составных частей. Одна часть представляет некоторое значение, а вторая — степень этого значения. Обратите внимание на следующий пример. Рассмотрим два числа: 34.1245 и 34124.5. Они состоят из идентичных цифр, но имеют разный масштаб. Первое число можно представить как 0.341245 (основное значение) и 100 (степень) Второе число можно представить как 0.341245 (то же самое основное значение) и 100000 (большая степень). Степень служит для перемещения десятичной точки, отсюда и термин "плавающая точка". В языке C++ для представления чисел с плавающей точкой используется подобный метод, за исключением того, что внутри компьютера хранятся двоичные числа и степени кратны 2, а не 10. К счастью, вам не требуется подробно знать о внутреннем представлении. Главными моментами здесь является то, что типы данных с плавающей точкой позволяют представлять дробные, очень большие и очень маленькие числа, а также то, что их внутреннее представление совершенно иное, чем представление целых чисел.

Запись чисел с плавающей точкой

В языке C++ имеется два способа записи чисел с плавающей точкой. Первый способ — это стандартная запись чисел с десятичной точкой (десятичных дробей), которой вы обычно пользуетесь:

12.34	//число с плавающей точкой
939001.32	//число с плавающей точкой
0.00023	//число с плавающей точкой
8.0	//также число с плавающей точкой

Даже если дробная часть равна 0, как в числе 8.0, наличие десятичной точки гарантирует, что число будет храниться в формате чисел с плавающей точкой, а не целых чисел. (Стандарт языка C++ позволяет представлять числа с плавающей точкой в соответствии с правилами различных регионов; например, разрешается вместо десятичной точки использовать запятую, как это принято в Европе. Однако это определяет вид чисел только при вводе и выводе, а не в коде программы.)

Второй способ записи чисел с плавающей точкой называется экспоненциальной формой записи или просто экспоненциальной записью, например: **3.45E6**. Эта запись означает, что число 3.45 умножается на 1000000; E6 означает 10 в 6-й степени, т.е. 1 с шестью последующими нулями. Таким образом, **3.45E6** означает 3450000. Здесь 6 называется **экспонентой**, а 3.45 — **мантиссой**. Вот несколько примеров:

```
2.52e+8 //можно использовать Е или е,
//знак "+" необязателен
8.33E-4 //экспонента может быть
//отрицательной
7E5 //то же самое, что и 7.0E+05
-18.32e13 //перед мантиссой может
//стоять знак "+" или "-".
2.857e12 //государственный долг США, 1989 г.
5.98E24 //масса Земли в килограммах
9.11e-31 //масса электрона в килограммах
```

Как вы могли заметить, экспоненциальная запись наиболее полезна для очень больших и очень маленьких чисел. Экспоненциальная запись числа гарантирует, что число будет храниться в формате чисел с плавающей точкой, даже если десятичная точка отсутствует. Обратите внимание, что можно писать, как Е, так и е, а экспонента может иметь знак "+" или "-". (рис. 3.3). Однако внутри числа нельзя делать пробелы: запись **7.2 Е6** неправильна.

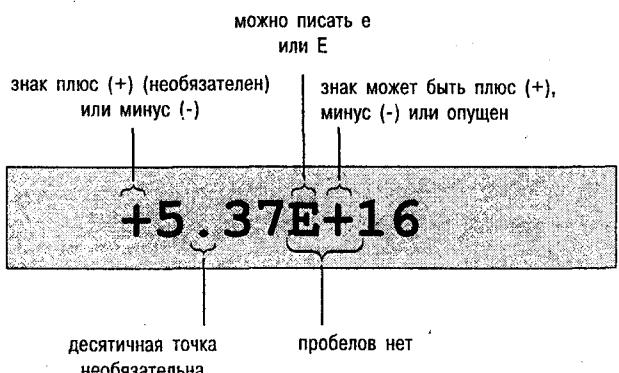


РИСУНОК 3.3 Экспоненциальная форма записи чисел.

Отрицательная экспонента означает не умножение, а деление на 10 в соответствующей степени. Таким образом, **8.33E-4** означает 8.33 разделить на 10^4 , или 0.000833. Аналогично масса электрона **9.11e-31** кг означает

0.0000000000000000000000000000911 кг

Выбирайте любую форму записи, какая вам нравится. (Междуд прочим, обратите внимание, что 911 — это, как правило, номер чрезвычайной службы спасения в США, а телефонные сообщения передаются с помощью электронов. Случайное совпадение или тайный говор учёных? Решайте сами.) Обратите также внимание, что **-8.33E4** соответствует **-83300**. Знак мантиссы является знаком числа, тогда как знак экспоненты относится к порядку числа.

ПОМНИТЕ

Запись **d.dddE+n** означает: переместить десятичную точку на **n** позиций вправо, а запись **d.dddE-p** означает: переместить десятичную точку на **p** позиций влево.

Типы данных с плавающей точкой

В языке C++, как и в стандарте ANSI C, имеется три типа данных с плавающей точкой: **float**, **double** и **long double**. Данные этих типов характеризуются числом значащих цифр, которые они могут иметь, и минимально допустимым диапазоном значений экспоненты. **Значащие цифры** — это цифры, определяющие значение числа. Например, когда высота горы Шаста (Shasta) в Калифорнии записывается в таком виде: 14162 фута, — то при этом используется пять значащих цифр, так как высота указывается с точностью до фута. Но когда пишут, что высота горы Шаста равна приблизительно 14000 футов, то при этом используются две значащие цифры, так как высота округлена до тысяч футов. В этом случае три оставшиеся цифры являются просто заполнителями разрядов числа. Количество значащих цифр не зависит от положения десятичной точки. Например, можно написать, что высота равна 14162 футов. Здесь опять используется пять значащих цифр, так как высота указывается с точностью до 5-й цифры.

В результате требований, предъявляемых в языках С и С++ к числу значащих цифр, данные типа **float** имеют минимальную величину 32 бита, данные типа **double** — минимальную величину 48 битов (но при этом не меньшую, чем данные типа **float**), и данные типа **long double** имеют величину, не меньшую, чем данные типа **double**. Все эти три типа данных могут иметь одинаковые размеры. Однако, как правило, данные типа **float** имеют величину 32 бита, данные типа **double** — 64 бита и данные типа **long double** — 80, 96 или 128 битов.

Кроме того, минимальный диапазон значений показателя для всех трех типов лежит в пределах от -37 до +37. Вы можете заглянуть в заголовочный файл `cfloat` или `float.h`, чтобы узнать предельные значения для всех типов данных с плавающей точкой в своей системе. (Файл `cfloat` — это адаптированная для языка C++ версия файла `float.h` языка C.) Вот, например, некоторые элементы из файла `float.h` для реализации Borland C++Builder с пояснениями:

```
// это минимальное число значащих цифр
#define DBL_DIG 15           // double
#define FLT_DIG 6            // float
#define LDBL_DIG 18          // long double

// это число битов, используемое для
// представления мантиссы
#define DBL_MANT_DIG 53
#define FLT_MANT_DIG 24
#define LDBL_MANT_DIG 64

// это максимальное и минимальное
// значения показателя
#define DBL_MAX_10_EXP +308
#define FLT_MAX_10_EXP +38
#define LDBL_MAX_10_EXP +4932

#define DBL_MIN_10_EXP -307
#define FLT_MIN_10_EXP -37
#define LDBL_MIN_10_EXP -4931
```

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторые реализации языка C++ заголовочный файл `cfloat` еще не добавлен, а в некоторых реализациях языка C++ с компиляторами, выпущенными до принятия стандарта ANSI C, заголовочный файл `float.h` отсутствует.

В программе из листинга 3.7 сравниваются типы данных `float` и `double` по критерию точности представления чисел (имеется в виду число значащих цифр). В программе используется метод `setf()` из класса `ostream`, рассмат-

риваемый в главе 16. Данный конкретный вызов метода приводит к тому, что выходные данные имеют формат с фиксированной точкой, что лучше иллюстрирует заданную точность данных. При этом предотвращается вывод больших чисел в экспоненциальной записи и обеспечивается отображение шести цифр после десятичной точки. Аргументы `ios_base::fixed` и `ios_base::floatfield` являются константами, которые становятся доступными для программы путем включения файла `iostream`.

В результате выполнения программы получаются следующие результаты:

```
tub = 3.333333, a million tubs = 3333333.250000,
and ten million tubs = 33333332.000000
mint = 3.333333 and a million mints = 3333333.333333
```

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В стандарте языка C++ аргумент `ios::fixed` был заменен на `ios_base::fixed`, а `ios::floatfield` — на `ios_base::floatfield`. Если ваш компилятор не воспринимает конструктор `ios_base`, попробуйте использовать `ios`. При выводе чисел с плавающей точкой в ранних версиях языка C++ по умолчанию отображалось шесть цифр после десятичной точки, например, 2345.831541. В стандарте C++ определено, что по умолчанию отображается всего шесть цифр (2345.83), а для чисел, равных миллиону или превышающих его, происходит переход к экспоненциальной форме записи (2.34583E+06). Однако в других режимах отображения, например в режиме `fixed`, отображается шесть цифр после десятичной точки, как в новой, так и в старой версии C++.

По умолчанию также подавляется отображение замыкающих нулей, и число 23.4500 отображается как 23.45. Реализации языка C++ отличаются тем, как они реагируют на изменение в операторе `setf()` настроек, используемых по умолчанию. В более старых версиях, таких как Borland C++ 3.1 для DOS, замыкающие нули в этом режиме также подавляются. В версиях, соответствующих стандарту, таких как Microsoft Visual C++ 5.0, Metrowerks CodeWarrior и Borland C++ Builder, эти нули отображаются, как показано в листинге 3.7.

Листинг 3.7 Программа `floatnum.cpp`.

```
// Программа floatnum.cpp — типы данных с плавающей точкой
#include <iostream>
using namespace std;
int main()
{
    cout.setf(ios_base::fixed, ios_base::floatfield);      // данные выводятся в режиме
                                                               // с фиксированной точкой
    float tub = 10.0 / 3.0;        // приблизительно шесть правильных значащих цифр
    double mint = 10.0 / 3.0;     // приблизительно 15 правильных значащих цифр
    const float million = 1.0e6;

    cout << "tub = " << tub;
    cout << ", a million tubs = " << million * tub;
    cout << ",\nand ten million tubs = ";
    cout << 10 * million * tub << "\n";

    cout << "mint = " << mint << " and a million mints = ";
    cout << million * mint << "\n";
    return 0;
}
```

Примечания к программе

Объект `cout` обычно игнорирует правые нули. Например, он отобразит число 3333333.250000 как 3333333.25. Вызов метода `cout.setf()` отменяет этот режим, по крайней мере, в новых реализациях. Главное, на что следует обратить здесь внимание, — это меньшая точность данных типа `float` по сравнению с данными типа `double`. Переменные `tub` и `mint` инициализируются одним и тем же значением 10.0/3.0. Это дает в результате число 3.33333333333333... (и т.д.). Поскольку объект `cout` выводит шесть цифр справа от десятичной точки, мы видим, что значения переменных `tub` и `mint` являются точными. Однако, после того как они умножаются в программе на миллион, вы видите, что значение переменной `tub` отличается от правильного значения после седьмой цифры "3". Переменная `tub` является правильной с точностью до семи значащих цифр. (Эта система гарантирует шесть правильных значащих цифр для данных типа `float`, но это в худшем случае.) Однако отображаемое значение переменной типа `double` содержит 13 цифр "3". Так что оно является правильным с точностью, по меньшей мере, до 13 значащих цифр. Это не должно вас удивлять, так как данный тип данных гарантирует 15 правильных значащих цифр. Кроме того, обратите внимание, что если умноженное на миллион значение переменной `tub` умножить еще на десять, то полученный результат будет неточным; это опять указывает на ограниченную точность данных типа `float`.

В классе `ostream`, к которому принадлежит объект `cout`, имеются функции-элементы класса, которые позволяют полностью форматировать выходные данные — устанавливать ширину полей, число знаков справа от десятичной точки, форму записи (десятичная или экспоненциальная) и т.д. Все эти возможности рассматриваются в главе 16. В примерах данной книги форматирование выходных данных не производится и обычно используется только операция `<<`. При этом иногда отображается больше цифр, чем необходимо, но это может вызывать только эстетическое неудовольствие. Если же такое положение дел вас не устраивает, можете заглянуть в главу 16, чтобы научиться использовать методы форматирования.

Константы с плавающей точкой

Когда вы используете в программе константу с плавающей точкой, в каком формате она будет храниться в компьютере? По умолчанию константы с плавающей точкой, например 8.24 и 2.4E8, будут иметь тип данных `double`. Если вам требуется константа типа `float`, добавьте к ней суффикс `f` или `F`. Чтобы тип константы был `long double`, добавьте суффикс `l` или `L`.

1.234f	// константа типа float
2.45E20F	// константа типа float
2.345324E28	// константа типа double
2.2L	// константа типа long double

Преимущества и недостатки типов данных с плавающей точкой

У типов данных с плавающей точкой имеется два преимущества по сравнению с целочисленными данными. Во-первых, дробные числа могут быть представлены только типами данных с плавающей точкой. Во-вторых, в связи с наличием экспоненты (степени), типы данных с плавающей точкой имеют намного больший диапазон значений. С другой стороны, операции над типами данных с плавающей точкой выполняются медленнее, чем операции над целочисленными данными, по крайней мере, в компьютерах без математических сопроцессоров; кроме того, для типов данных с плавающей точкой может произойти потеря точности. Последнее положение иллюстрируется в программе из листинга 3.8.

Листинг 3.8 Программа fitadd.cpp.

```
// программа fitadd.cpp — проблемы, связанные
// с потерей точности данных типа float
#include <iostream>
using namespace std;
int main()
{
    float a = 2.34E+22f;
    float b = a + 1.0f;
    cout << "a = " << a << "\n";
    cout << "b - a = " << b - a << "\n";
    return 0;
}
```

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторых первых реализациях языка C++ с компиляторами, выпущенными до появления стандарта ANSI C, нельзя использовать суффикс `f` для определения констант с плавающей точкой. Если у вас возникнет такая проблема, можете заменить `2.34E+22f` на `2.34E+22`, а `1.0f` — на `(float) 1.0`.

В программе к некоторому числу добавляется 1, а затем из суммы вычитается первоначальное число. В результате должна получиться единица. Не так ли? Вот результат, полученный в одной из вычислительных систем:

```
a = 2.34e+022
b - a = 0
```

Проблема заключается в том, что `2.34E+22` — это число с 23 значащими цифрами слева от десятичной точки. Прибавляя к нему единицу, вы пытаетесь добавить единицу к 23-й цифре этого числа. Но тип данных `float` имеет только шесть или семь значащих цифр, поэтому попытка изменить 23-ю цифру не оказывает на число никакого влияния.

КЛАССИФИКАЦИЯ ТИПОВ ДАННЫХ

Классификация основных типов данных в языке C++ производится путем разделения их на несколько семейств данных, что привносит в язык некоторый порядок. Типы данных ***signed char***, ***short***, ***int*** и ***long*** называются целочисленными типами данных со знаком. Разновидности этих типов данных без знака называются целочисленными типами данных без знака. Типы данных ***bool***, ***char***, ***wchar_t***, а также типы целочисленных данных со знаком и без знака все вместе называются целочисленными типами данных. Типы данных ***float***, ***double*** и ***long double*** называются типами данных с плавающей точкой. Целочисленные типы данных и типы данных с плавающей точкой вместе называются арифметическими типами данных.

Арифметические операции языка C++

Возможно, у вас остались теплые воспоминания от решения арифметических упражнений в школе. Такое же удовольствие вы можете доставить своему компьютеру. В языке C++ имеется пять основных арифметических операций: сложение, вычитание, умножение, деление и деление по модулю. В каждой из этих операций используются два числа (называемых *операндами*) для вычисления результата. Знак операции и два операнда составляют *выражение*. Например, рассмотрим следующий оператор:

```
int wheels = 4 + 2;
```

Числа **4** и **2** являются операндами, знак **"+"** является знаком сложения, а **4+2** является выражением, значение которого равно **6**.

Ниже описаны пять основных арифметических операций языка C++:

- Операция **+** задает сложение двух операндов. Например, **4 + 20** равно **24**.

- Операция **-** задает вычитание второго операнда из первого. Например, **12 - 3** равно **9**.
- Операция ***** задает умножение операндов. Например, **28 * 4** равно **112**.
- Операция **/** задает деление первого операнда на второй. Например, **1000 / 5** равно **200**. Если оба операнда — целые числа, результатом будет целая часть частного. Например, **17 / 3** равно **5**, дробная часть отбрасывается.
- Операция **%** задает нахождение модуля первого операнда по отношению ко второму, т.е. результатом операции будет остаток от деления первого операнда на второй. Например, **19 % 6** равно **1**, так как число **19** содержит три раза по **6** с остатком **1**. Оба операнда должны быть целочисленными данными. Если один из операндов отрицателен, знак результата зависит от реализации языка.

Конечно, в качестве operandов можно использовать и переменные, и константы. Именно так делается в программе из листинга 3.9. Поскольку операция **%** выполняется только над целыми числами, мы оставим ее для будущих примеров.

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если ваш компилятор не воспринимает слово **ios_base** в аргументах метода **setf()**, попробуйте ранее использовавшееся слово **ios**.

В результате выполнения программы получаются следующие результаты. Как видите, языку C++ в отношении выполнения простых арифметических действий можно доверять:

Листинг 3.9 Программа arith.cpp.

```
// Программа arith.cpp — некоторые арифметические операции языка C++
#include <iostream>
using namespace std;
int main()
{
    float hats, heads;

    cout.setf(ios_base::fixed, ios_base::floatfield); // Режим с фиксированной точкой
    cout << "Enter a number: ";
    cin >> hats;
    cout << "Enter another number: ";
    cin >> heads;

    cout << "hats = " << hats << "; heads = " << heads << "\n";
    cout << "hats + heads = " << hats + heads << "\n";
    cout << "hats - heads = " << hats - heads << "\n";
    cout << "hats * heads = " << hats * heads << "\n";
    cout << "hats / heads = " << hats / heads << "\n";
    return 0;
}
```

```
Enter a number: 50.25
Enter another number: 11.17
hats = 50.250000; heads = 11.170000
hats + heads = 61.419998
hats - heads = 39.080002
hats * heads = 561.292480
hats / heads = 4.498657
```

Ну, пожалуй, полностью доверять нельзя. Сложение чисел 11.17 и 50.25 должно дать в результате 61.42, но на самом деле получается 61.419998. Но это не проблема выполнения арифметической операции; эта проблема связана с ограниченной точностью (т.е. с ограниченным числом значащих цифр) данных типа `float`.

Помните, что в языке C++ у данных типа `float` гарантируется только шесть значащих цифр. Если вы округлите 61.419998 до шести значащих цифр, то получите 61.4200, что является правильным значением для этой гарантированной точности. Мораль здесь заключается в том, что если вам требуется большая степень точности, используйте данные типа `double` или `long double`.

Приоритет операций и ассоциативность

Можно ли в языке C++ выполнять сложные арифметические вычисления? Да, но нужно знать, какие правила языка C++ при этом используются. Например, многие выражения содержат более одной операции. Отсюда возникает вопрос, какая операция должна выполняться первой? Рассмотрим такой оператор:

```
int flyingpigs = 3 + 4 * 5; //35 или 23?
```

Оказывается, что цифра "4" является операндом как в операции сложения (+), так и в операции умножения (*). Когда один операнд может участвовать более чем в одной операции, чтобы решить какая операция должна выполняться первой, в языке C++ используются правила *приоритета операций*. Для арифметических операций используется обычная алгебраическая приоритетность, в соответствии с которой умножение, деление и деление по модулю выполняются перед сложением и вычитанием. Таким образом, $3 + 4 * 5$ означает $3 + (4 * 5)$, а не $(3 + 4) * 5$. Следовательно, ответ будет 23, а не 35. Конечно, с помощью круглых скобок можно изменить порядок выполнения операций. В приложении D описывается приоритет всех операций языка C++. Обратите внимание, что в нем операции *, / и % находятся в одном ряду. Это означает, что они имеют одинаковый приоритет. Аналогично сложение и вычитание обладают одинаковым, но более низким приоритетом.

Однако в некоторых случаях одного приоритета недостаточно. Рассмотрим следующий оператор:

```
float logs = 120 / 4 * 5; //150 или 6?
```

Цифра "4" опять является операндом двух операций. Однако операции / и * имеют одинаковый приоритет,

поэтому одного приоритета здесь недостаточно, чтобы определить, что делать в первую очередь: делить 120 на 4 или умножать 4 на 5. А этот порядок имеет важное значение, так как в первом случае результат равен 150, а во втором — 6. Когда две операции имеют одинаковый приоритет, порядок выполнения операций определяется тем, какую ассоциативность имеют эти операции: "слева направо" или "справа налево". Ассоциативность "слева направо" означает, что из двух операций, имеющих одинаковый приоритет, первой выполняется та, которая находится слева. При ассоциативности "справа налево" первой выполняется операция, которая находится справа. Информация об ассоциативности также приводится в приложении D. Там можно видеть, что умножение и деление обладают ассоциативностью "слева направо". Это означает, что цифра "4" сначала участвует в операции, расположенной слева, т.е. 120 делится на 4, а затем полученный результат 30 умножается на 5. В результате получается 150.

Обратите внимание, что приоритет и ассоциативность играют роль только тогда, когда один операнд участвует в двух операциях. Рассмотрим следующее выражение:

```
int dues = 20 * 5 + 24 * 6;
```

В соответствии с приоритетностью операций перед сложением необходимо выполнить две операции умножения: $20 * 5$ и $24 * 6$. Однако ни правила приоритета, ни правила ассоциативности ничего не говорят о том, какая операция умножения должна выполняться первой. Вы могли бы подумать, что в соответствии с правилами ассоциативности первой должна выполняться левая операция умножения; но в данном случае две операции умножения не имеют общего операнда, так что правила ассоциативности здесь неприменимы. Какой порядок выбрать — это остается на усмотрение разработчиков конкретной реализации языка. В данном примере любой порядок дает тот же самый результат, но в некоторых ситуациях результаты могут быть разными. Один такой случай вы увидите в главе 5 при рассмотрении оператора инкремента.

Разновидности операции деления

Мы еще не полностью рассмотрели операцию деления. То, как выполняется эта операция, зависит от типа операндов. Если оба операнда являются целыми числами, то выполняется операция деления целых чисел. Это означает, что любая дробная часть результата отбрасывается и результат становится целым числом. Если оба операнда являются числами с плавающей точкой, то дробная часть результата сохраняется и результат будет числом с плавающей точкой. В программе из листинга

3.10 иллюстрируется, как в языке C++ осуществляется деление данных различных типов. Для форматирования выходных данных в этой программе, так же как и в программе из листинга 3.8, вызывается функция-элемент `setf()`.

Листинг 3.10 Программа divide.cpp.

```
// Программа divide.cpp – деление целых
// чисел и чисел с плавающей точкой
#include <iostream>
using namespace std;
int main()
{
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << "Integer division: 9/5 = "
        << 9 / 5 << "\n";
    cout << "Floating-point division: 9.0/5.0 = ";
    cout << 9.0 / 5.0 << "\n";
    cout << "Mixed division: 9.0/5 = "
        << 9.0 / 5 << "\n";
    cout << "double constants: 1e7/9.0 = ";
    cout << 1.e7 / 9.0 << "\n";
    cout << "float constants: 1e7f/9.0f = ";
    cout << 1.e7f / 9.0f << "\n";
    return 0;
}
```

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если ваш компилятор не воспринимает слово `ios_base` в аргументах функции-элемента `setf()`, попробуйте вместо него использовавшееся ранее слово `ios`.

В некоторых реализациях языка C++ с компиляторами, выпущенными до появления стандарта ANSI C, нельзя использовать суффикс `f` в константах с плавающей точкой. Если у вас возникла эта проблема, можете заменить `1.e7f / 9.0f` на `[float] 1.e7 / [float] 9.0`.

В некоторых реализациях языка подавляется отображение замыкающих нулей.

Вот выходные данные, полученные в одной из реализаций языка:

```
Integer division: 9/5 = 1
Floating-point division: 9.0/5.0 = 1.800000
Mixed division: 9.0/5 = 1.800000
double constants: 1e7/9.0 = 1111111.111111
float constants: 1e7f/9.0f = 1111111.125000
```

Первая строка выходных данных показывает, что деление целого числа 9 на целое число 5 дает целое число 1. Дробная часть (4/5 или 0.8) отбрасывается. О практической пользе такого вида деления вы узнаете, когда изучите операцию деления по модулю. Следующие две строки показывают, что если, по крайней мере, один из операндов является числом с плавающей точкой, то результат также будет числом с плавающей точкой (1.8). Если в операции задействованы данные разных типов, то все они преобразуются в данные какого-то одного типа. Об этих осуществляемых автоматически преобразованиях вы узнаете далее в настоящей главе. Сравнительная точность данных в двух последних строках по-

казывает, что результат будет иметь тип `double`, если оба операнда имеют тип `double`, и он будет иметь тип `float`, если оба операнда являются данными типа `float`. Помните, что константы с плавающей точкой по умолчанию имеют тип `double`.

КРАТКОЕ ЗАМЕЧАНИЕ КАСАЮЩЕЕСЯ ПЕРЕГРУЗКИ ОПЕРАЦИЙ

В программе из листинга 3.10 представлены три разные операции деления: деление данных типа `int`, деление данных типа `float` и деление данных типа `double`. Какая именно операция будет выполняться – зависит от контекста программы (в данном случае от типа операндов). Использование одного знака для нескольких разных операций называется **перегрузкой операции** (**перегрузкой оператора**). В языке C++ имеется несколько случаев перегрузки операции. Язык C++ позволяет также расширять перегрузку операции на определяемые пользователем классы, поэтому то, что вы видите здесь, – это всего лишь предвестник важного свойства ООП (рис. 3.4).

Операция деления по модулю

Большинство людей гораздо лучше знакомы со сложением, вычитанием, умножением и делением, чем с делением по модулю, поэтому давайте рассмотрим эту операцию в действии. Результатом деления по модулю является остаток от деления целых чисел. Операция деления по модулю совместно с операцией деления целых чисел особенно полезна в тех задачах, где некоторую величину требуется разделить на различные целые единицы, например, преобразовать дюймы в футы и дюймы или преобразовать доллары в кварты, даймы, никели и пенни. В главе 2, в программе из листинга 2.6 вес в стоунах преобразовывался в фунты. В программе из листинга 3.11 осуществляется обратный процесс: преобразование веса в фунтах в стоуны. Стоун, как помните, равен 14 фунтам, и в Великобритании большинство напольных весов откалибровано в этих единицах измерения.

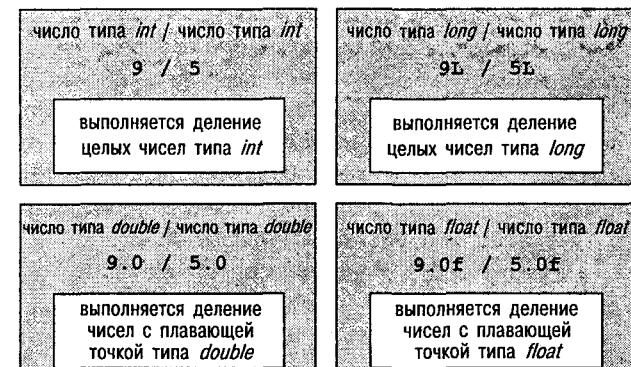


РИСУНОК 3.4 Разновидности операции деления.

Листинг 3.11 Программа modulus.cpp.

```
// Программа modulus.cpp – для преобразования фунтов в стоуны используется операция %
#include <iostream>
using namespace std;
int main()
{
    const int Lbs_per_stn = 14;
    int lbs;

    cout << "Enter your weight in pounds: ";
    cin >> lbs;
    int stone = lbs / Lbs_per_stn;           // число целых стоунов
    int pounds = lbs % Lbs_per_stn;          // остаток в фунтах
    cout << lbs << " pounds are " << stone;
    cout << " stone, " << pounds << " pound(s). \n";
    return 0;
}
```

Деление целых чисел используется в этой программе, чтобы найти число целых стоунов в весе, а деление по модулю — чтобы найти число оставшихся фунтов.

В результате выполнения программы получаются следующие результаты:

```
Enter your weight in pounds: 184
184 pounds are 13 stone, 2 pound(s).
```

В выражении `lbs / Lbs_per_stn` оба операнда имеют тип `int`, поэтому компьютер выполняет деление целых чисел. Если значение переменной `lbs` равно 184, то результат будет равен 13. Результат произведения 13 и 14 равен 182, поэтому остаток от деления 184 на 14 равен 2. Это и будет значение выражения `lbs % Lbs_per_stn`. Теперь вы готовы, если не морально, то технически, отвечать на вопросы о своем весе во время поездки в Великобританию.

Преобразования типов данных

Разнообразие типов данных в языке C++ дает программисту возможность выбирать тип данных, соответствующий конкретной потребности. Например, сложение двух чисел типа `short` может выполняться с помощью иных аппаратных инструкций, чем сложение двух чисел типа `long`. Когда имеется 11 типов целочисленных данных и три типа данных с плавающей точкой, компьютеру приходится обрабатывать множество различных случаев, особенно если в одной операции смешиваются данные различных типов. Чтобы не допустить возможной мешаницы, в языке C++ многие преобразования типов данных выполняются автоматически:

- Преобразование данных осуществляется, когда данные одного арифметического типа присваиваются переменной другого арифметического типа.
- Преобразование данных осуществляется, когда в выражении содержатся данные разных типов.
- Преобразование данных осуществляется при передаче аргументов в функции.

Если вам непонятно, что происходит при этих автоматических преобразованиях, то результаты выполнения некоторых программ могут вас озадачить. Поэтому рассмотрим эти правила более подробно.

Преобразование при присваивании

В языке C++ можно довольно свободно числовые данные одного типа присваивать переменной другого типа. При этом присваиваемые данные преобразуются к типу переменной. Например, предположим, что имеется переменная `so_long` типа `long` и переменная `thirty` типа `short`, а в программе присутствует следующий оператор:

```
so_long = thirty; //присваивание типа
//данных short
//переменной типа long
```

При выполнении этого присваивания программа берет значение переменной `thirty` (как правило, 16-битовое число) и расширяет его до типа данных `long` (как правило, 32-битовое число). Обратите внимание, что при расширении создается новое число, которое и присваивается переменной `so_long`; содержимое переменной `thirty` не изменяется.

Присваивание значения переменной, имеющей большую величину, обычно не вызывает проблем. Например, при присваивании значения типа `short` переменной типа `long` присваиваемое значение не изменяется; оно просто дополняется свободными битами. Однако при присваивании большого значения типа `long`, например 2111222333, переменной типа `float` приводит к некоторой потере точности. Поскольку переменная типа `float` может иметь только шесть значащих цифр, присваиваемое значение может быть округлено до 2.11122E9. В табл. 3.3 описываются некоторые проблемы, возникающие при преобразовании типов данных.

Нулевое значение, присваиваемое переменной типа `bool`, преобразуется в значение `false`, а ненулевое значение преобразуется в значение `true`.

Таблица 3.3 Возможные проблемы, возникающие при преобразовании типов данных.

Преобразование	Возможные проблемы
Данные с плавающей точкой большей величины в данные с плавающей точкой меньшей величины, например, типа данных double в тип данных float	Потеря точности (значащих цифр), преобразуемые данные могут быть большей величины, чем целевой тип данных, в этом случае результат не определен
Данные с плавающей точкой в целочисленные данные	Потеря дробной части, исходные данные могут быть большей величины, чем данные целевого типа, в этом случае результат не определен
Целочисленные данные большей величины в целочисленные данные меньшей величины, например, данные типа long в данные типа short	Исходное значение может быть большей величины, чем данные целевого типа, копируются, как правило, только младшие байты

При присваивании значений с плавающей точкой целочисленным переменным возникает две проблемы. Во-первых, преобразование данных с плавающей точкой в целочисленные данные приводит к урезанию числа (отбрасыванию дробной части). Во-вторых, значение типа **float** может быть слишком большим, чтобы поместиться в переменной типа **int**, имеющей меньшую величину. В языке C++ не определено, каким должен быть результат в данном случае; это означает, что в разных реализациях результаты могут быть разными. В программе из листинга 3.12 демонстрируется несколько преобразований при присваивании значений.

Листинг 3.12 Программа assign.cpp.

```
// Программа assign.cpp – изменение типа
// данных при присваивании
#include <iostream>
using namespace std;
int main()
{
    float tree = 3; //данные типа int
                     //преобразуются в
                     //данные типа float
    int guess = 3.9832; //данные типа float
                       //преобразуются в
                       //данные типа int
    int debt = 3.0E12; //в языке C++
                      //результат
                      //не определен
    cout << "tree = " << tree << "\n";
    cout << "guess = " << guess << "\n";
    cout << "debt = " << debt << "\n";
    return 0;
}
```

В результате выполнения программы в одной из систем получается следующий результат:

```
tree = 3
guess = 3
debt = 0
```

Здесь переменной **tree** присваивается значение с плавающей точкой 3.0. Однако поскольку объект **cout** при выводе данных опускает завершающие нули, то вместо

3.0 отображается 3. При присваивании числа 3.9832 переменной **guess** типа **int** оно урезается до числа 3; в языке C++ при преобразовании данных с плавающей точкой в целочисленные данные осуществляется урезание числа (отбрасывание дробной части), а не округление (замена ближайшим целым числом). И наконец, обратите внимание, что переменной **debt** типа **int** не может быть присвоено значение 3.0E12. Результат такой ситуации в языке C++ не определен. В нашей системе переменной **debt** присваивается 0. Ну что ж, это новый способ решения проблемы крупной задолженности!

Некоторые компиляторы выдают предупреждения о возможной потере данных в тех операторах, где целочисленные переменные инициализируются числами с плавающей точкой. Кроме того, отображаемое значение переменной **debts** меняется от компилятора к компилятору. Например, выполнение этой же самой программы в другой системе дает для переменной **debts** значение 2112827392.

Преобразования данных в выражениях

Теперь рассмотрим, что происходит, когда в одном выражении содержатся данные двух различных арифметических типов. В этом случае в языке C++ осуществляется два вида автоматических преобразований данных. Во-первых, некоторые типы данных в выражениях всегда автоматически преобразуются. Во-вторых, некоторые типы данных в выражениях преобразуются тогда, когда они находятся в комбинации с некоторыми другими типами данных.

Рассмотрим сначала первый вид автоматических преобразований. При вычислении выражений в языке C++ типы данных **bool**, **char**, **unsigned char**, **signed char** и **short** преобразуются в типы данных **int**. В частности, значение **true** устанавливается равным 1, а значение **false** — 0. Эти преобразования называются *интегральными повышениями*. Рассмотрим, например, следующие "птичьи" операторы:

```
short chickens = 20; //строка 1
short ducks = 35; //строка 2
short fowl = chickens + ducks; //строка 3
```

При выполнении оператора в строке 3 значения переменных `chickens` и `ducks` преобразуются в данные типа `int`. Затем результат преобразуется опять в данные типа `short`, так как он присваивается переменной типа `short`. Может показаться, что выражение вычисляется не самым прямым путем, но в этом есть смысл. Для конкретного вида компьютеров данные типа `int` выбираются так, чтобы они были наиболее естественными; таким образом, предполагается, что вычисления с данными этого типа компьютер выполняет быстрее всего.

Имеется еще несколько интегральных повышений: данные типа `unsigned short` преобразуются в данные типа `int`, если данные типа `short` имеют меньшую величину, чем данные типа `int`. Если оба эти типа данных имеют одинаковую величину, то данные типа `unsigned short` преобразуются в данные типа `unsigned int`. Это правило гарантирует, что при повышении типа данных `unsigned short` не произойдет потеря данных. Аналогично этому тип данных `wchar_t` повышается до наименьшего из следующих типов данных: `int`, `unsigned int`, `long` или `unsigned long`, способного хранить тип данных `wchar_t`.

Кроме того, некоторые преобразования выполняются тогда, когда данные разных типов объединяются арифметически, например, при сложении данных типа `int` и `float`. Когда некоторая операция выполняется над данными двух разных типов, то данные меньшего типа преобразуются в данные большего типа. Например, в программе из листинга 3.10 производится деление числа 9.0 на число 5. Поскольку 9.0 — это число типа `double`, перед делением тип числа 5 преобразуется в `double`. В более общем случае то, какие преобразования должны быть сделаны в арифметическом выражении, компилятор определяет с помощью определенного алгоритма. Вот этот алгоритм — именно в такой последовательности компилятор выполняет преобразования:

- Если один из operandов имеет тип `long double`, другой оператор преобразуется также в тип `long double`.
- В противном случае, если один из operandов имеет тип `double`, другой оператор преобразуется также в тип `double`.
- В противном случае, если один из operandов имеет тип `float`, другой operand преобразуется также в тип `float`.
- В противном случае operandы являются целочисленными данными и выполняются интегральные повышения.
- В этом случае, если один из operandов имеет тип `unsigned long`, другой operand преобразуется также в тип `unsigned long`.
- Если один из operandов имеет тип `long int`, а другой — тип `unsigned int`, то преобразование зависит от от-

носительных величин этих двух типов. Если величины данных типа `long` достаточны для представления данных типа `unsigned int`, оператор типа `unsigned int` преобразуется в тип `long`.

- В противном случае оба операнда преобразуются в тип `unsigned long`.
- В противном случае, если один из operandов имеет тип `long`, другой преобразуется также в тип `long`.
- В противном случае, если один из operandов имеет тип `unsigned int`, другой преобразуется также в тип `unsigned int`.
- Если компилятор доходит до этого места в списке, то оба операнда должны иметь тип `int`.

В стандарте ANSI C установлены такие же правила, как и в языке C++, но в стандарте K&R C правила немного иные. Например, в классическом языке С тип данных `float` повышается до типа данных `double` даже тогда, когда оба операнда имеют тип `float`.

Преобразования данных при передаче аргументов

Как вы узнаете из главы 7, в языке C++ контроль преобразования данных при передаче аргументов обычно осуществляется с помощью прототипов функций. Однако возможно, хотя, как правило, не очень целесообразно, при передаче аргументов отказываться от контроля с помощью прототипов функций. В этом случае в языке C++ производится интегральное повышение типов данных `char` и `short` (как `signed`, так и `unsigned`). Кроме того, чтобы сохранить совместимость с огромным количеством программ, написанных на классическом языке С, в языке C++ типы аргументов `floats` повышаются до типов `double` при передаче их в функцию, у которой отсутствует управление преобразованием данных с помощью прототипов.

Приведение типов

С помощью механизма приведения типов в языке C++ можно явно преобразовывать данные одного типа в данные другого типа. (Вот какой умелый язык C++!) Приведение типов осуществляется двумя способами. Например, чтобы преобразовать значение типа `int`, хранимое в переменной `thorn`, можно использовать одно из следующих выражений:

<code>(long) thorn</code>	//преобразует значение //переменной thorn в //данные типа long
<code>long (thorn)</code>	//преобразует значение //переменной thorn в //данные типа long

При приведении типов сама переменная `thorn` не изменяется; создается новое значение указанного типа,

которое затем можно использовать в каком-нибудь выражении.

В более общем виде это записывается так:

```
(имяТипа) значение //преобразует значение
//в данные типа имяТипа
имяТипа (значение) //преобразует значение
//в данные типа имяТипа
```

Первая форма записи определена в языке С, вторая форма — в языке С++. Новая форма введена для того, чтобы приведение типов было похожим на вызов функции. В результате этого приведения типов для готовых (встроенных) типов данных будут выглядеть как преобразования типов данных, которые можно выполнять (design) для определяемых пользователем классов.

В программе из листинга 3.13 кратко иллюстрируется применение обеих форм записи. Представьте себе, что первая часть этого листинга является частью мощной моделирующей экологической программы, в которой производятся вычисления над данными с плавающей точкой. Затем эти данные преобразуются в целочисленное количество птиц и зверей. Получаемые результаты будут зависеть от того, когда осуществляется преобразование. При вычислении значения переменной `auks` сначала складываются числа с плавающей точкой, а затем во время присваивания сумма преобразуется в данные типа `int`. Но при вычислении значений переменных `bats` и `coots` сначала числа с плавающей точкой преобразуются в данные типа `int` с помощью метода приведения типов и только потом складываются. В заключительной части программы показано, как, используя метод приведения типов, можно отображать код ASCII для значений типа `char`.

Листинг 3.13 Программа typecast.cpp.

```
// Программа typecast.cpp – изменение типов данных
#include <iostream>
using namespace std;
int main()
{
    int auks, bats, coots;

    // следующий оператор складывает числа как данные типа double,
    // а затем преобразует результат в данные типа int
    auks = 19.99 + 11.99;

    // эти операторы складывают числа как данные типа int
    bats = (int) 19.99 + (int) 11.99;      // старый синтаксис языка С
    coots = int (19.99) + int (11.99);     // новый синтаксис языка С++
    cout << "auks = " << auks << ", bats = " << bats;
    cout << ", coots = " << coots << '\n';

    char ch = 'Z';
    cout << "The code for " << ch << " is "; // выводит значение переменной как данные типа char
    cout << int(ch) << '\n';                  // выводит значение переменной как данные типа int
    return 0;
}
```

Вот результат выполнения этой программы:

```
auks = 31, bats = 30, coots = 30
The code for Z is 90
```

Сначала сложение чисел 19.99 и 11.99 дает результат 31.98. Когда это число присваивается переменной `auks` типа `int`, оно урезается до числа 31. Но затем с помощью приведения типов эти же два числа урезаются соответственно до чисел 19 и 11 еще до сложения. В результате значения переменных `bats` и `coots` получаются равными 30. В последнем операторе `cout` метод приведения типов используется для преобразования значения типа `char` в значение типа `int` перед отображением результата на экране. Это приводит к тому, что объект `cout` выводит данное значение как целое число, а не как символ.

В этой программе иллюстрируются две причины, по которым приходится применять метод приведение типов. Во-первых, в программе могут быть числа, которые хранятся как данные типа `double`, но используются для вычисления значений типа `int`. Например, можно моделировать целые величины, например численность населения, используя числа с плавающей точкой. Вам может потребоваться, чтобы при вычислениях эти числа трактовались как данные типа `int`. Приведение типов позволяет сделать это непосредственно. Обратите внимание, что вы получаете разные результаты (по крайней мере, для этих чисел) в том случае, когда сначала преобразуете их в данные типа `int`, а затем складываете, и когда сначала складываете, а затем преобразуете в данные типа `int`.

Во второй части этой программы демонстрируется наиболее распространенная причина, по которой применяется приведение типов, — возможность применять данные одного формата для различных целей. Например, в этой программе переменная `ch` типа `char` содержит код для буквы "Z". Объект `cout`, который выводит содержимое переменной `ch`, отображает на экране символ Z, так как объект `cout` обращает внимание главным образом на тот факт, что переменная `ch` имеет тип `char`. Однако, преобразуя переменную `ch` в тип `int` с помощью метода приведения типов, вы добьетесь того, что объект `cout` будет выводить ее содержимое как данные типа `int` и отобразит на экране хранимый в ней код ASCII.

Резюме

Основные типы данных языка C++ подразделяются на две группы. Одна группа состоит из типов данных, которые хранятся как целые числа. Вторая группа включает типы данных, которые хранятся в формате чисел с плавающей точкой. Различные типы целочисленных данных отличаются друг от друга количеством ячеек памяти, используемых для хранения чисел, а также наличием или отсутствием знака. Вот все типы целочисленных данных (начиная с самого маленького и заканчивая самым большим): `bool`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long` и `unsigned long`. Имеется также тип данных `wchar_t`, положение которого в этом ряду зависит от реализации языка. В языке C++ данные типа `char` должны иметь достаточно большую величину для представления основного набора символов системы, данные типа `wchar_t` — достаточно большую величину для представления расширенного набора символов системы, размер данных типа `short` — не менее 16 битов, размер данных типа `int` — не меньше размера данных типа `short`, размер данных типа `long` — не меньше 32 битов и не меньше размера данных типа `int`. Точная величина зависит от реализации языка.

Символы в компьютерах представлены с помощью своих числовых кодов. Система ввода/вывода определяет, каким образом трактуется код: как символ или как число.

Типы данных с плавающей точкой служат для представления дробных чисел и обеспечивают представление гораздо больших чисел, чем типы целочисленных данных. Существует три типа данных с плавающей точкой: `float`, `double` и `long double`. В языке C++ величина данных типа `float` должна быть не больше, чем размеры данных типа `double`, а величина данных типа `double` — не больше, чем величина данных типа `long double`. Как правило, данные типа `float` занимают 32 бита памяти, дан-

ные типа `double` — 64 бита и данные типа `long` — от 80 до 128 битов. Наличие в языке C++ большого количества типов данных различных величин, а также со знаком и без знака дает возможность программисту выбрать тот тип данных, который лучше всего соответствует конкретным требованиям, предъявляемым к данным.

В языке C++ имеются обычные арифметические операции, которые можно выполнять над числовыми данными различных типов: сложение, вычитание, умножение, деление и деление по модулю. Операции языка C++ подчиняются правилам приоритета и ассоциативности, которые определяют порядок выполнения операций в том случае, если один операнд участвует в двух операциях.

При присваивании значения одного типа переменной другого типа в языке C++ осуществляется преобразование этого значения в значение типа переменной. Кроме того, в языке C++ можно выполнять арифметические операции над данными разных типов, а также использовать метод приведения типов для выполнения преобразования данных из одного типа в другой. Многие преобразования данных из одного типа в другой являются безопасными в том смысле, что при этом не происходит потеря или изменение данных. Например, данные типа `int` можно преобразовывать в данные типа `long` без каких-либо проблем. Другие преобразования данных, например данных с плавающей точкой в целочисленные данные, требуют большей осторожности.

Возможно, большое количество основных типов данных в языке C++ поначалу покажется вам избыточным, особенно если принять во внимание различные правила преобразования данных. Однако вероятнее всего, рано или поздно вы окажетесь в ситуации, когда один из многочисленных типов данных окажется самым подходящим в данный момент, и тогда вы оцените это.

Вопросы для повторения

- Почему в языке C++ имеется несколько типов целочисленных данных?
- Определите следующие переменные:
 - Типа `short` со значением 80
 - Типа `unsigned int` со значением 42110
 - Целочисленного типа со значением 3000000000
- Каким образом в языке C++ предотвращается превышение предельных значений целочисленных данных какого-либо типа?
- Чем отличаются числа `33L` и `33`?

5. Эквивалентны ли следующие операторы языка C++:

```
char grade = 65;
char grade = 'A';
```

6. Как с помощью программы C++ найти, какому символу соответствует код 88? Приведите, по крайней мере, два способа решения этой задачи.

7. Присваивание значения типа `long` переменной типа `float` может привести к ошибке округления. А что будет в случае присваивания значения типа `long` переменной типа `double`?

8. Вычислите следующие выражения по правилам языка C++:

- a. $8 * 9 + 2$
- b. $6 * 3 / 4$
- c. $3 / 4 * 6$
- d. $6.0 * 3 / 4$
- e. $15 \% 4$

9. Предположим, что `x1` и `x2` — это две переменные типа `double`, которые вам требуется сложить как целочисленные данные и присвоить целочисленной переменной. Напишите оператор языка C++, выполняющий эту задачу.

Упражнения по программированию

1. Напишите короткую программу, которая запрашивает ваш рост в дюймах (с точностью до 1 дюйма), а затем выражает его в футах и дюймах. Пусть в программе используется символ подчеркивания, чтобы указать место, где вводить ответ. Кроме того, коэффициент преобразования (дюймов в футы) представьте в виде символьической константы `const`.
2. Напишите короткую программу, которая запрашивает ваш рост в футах и дюймах и ваш вес в фунтах. (Для хранения этой информации используйте три переменные.) Пусть программа вычислит и отобразит на экране ваш ИМТ (индекс массы тела). Чтобы вычислить ИМТ, рост в футах и дюймах выразите в дюймах. Затем рост в дюймах выразите в метрах, умножив его на 0.0254. Потом вес в фунтах выразите в килограммах, разделив его на 2.2. И наконец, вычислите ИМТ, разделив вес в килограммах на рост в метрах, возведенный в квадрат. Для представления различных коэффициентов преобразований используйте символьические константы.
3. Напишите программу, которая запрашивает, сколько миль вы проехали и сколько галлонов бензина израсходовали, а затем отображает на экране, сколько миль на один галлон проезжает ваш автомобиль. Или, если хотите, программа может запрашивать расстояние в километрах и количество бензина в литрах, а затем отображать результат в европейском стиле — расход бензина в литрах на 100 километров.

Производные типы данных

В этой главе рассматривается следующее:

- Создание и использование массивов
- Создание и использование строк
- Методы `getline()` и `get()`, используемые для чтения строк
- Смешанный строковый и числовой ввод
- Создание и использование структур
- Создание и использование объединений
- Создание и использование перечислений
- Создание и использование указателей
- Управление динамической памятью с помощью `new` и `delete`
- Создание динамических массивов
- Создание динамических структур
- Автоматическая, статическая и динамическая память

Представьте себе, что вы разработали компьютерную игру под названием *User-Hostile* (Враг пользователя), в котором игроки состязаются в остроумии с загадочным компьютерным интерфейсом. Теперь необходимо создать программу, которая будет контролировать ежемесячную продажу игр в течение пятилетнего периода, или, например, инвентаризировать накопленные данные по кредитным карточкам. Столкнувшись с поставленной задачей, вы скоро поймете, что нуждаетесь в чем-то большем чем простые основные типы данных C++. Язык C++ предлагает для этого производные типы. Это типы данных, сформированные на основе типа целого числа и типов с плавающей запятой. Наиболее перспективный производный тип — это класс, тот оплот ООП, к которому мы постепенно перемещаемся. Но C++ также поддерживает несколько более скромных производных типов, позаимствованных из языка C. Массив, например, может содержать несколько значений одного и того же типа. Специфический вид массива может содержать строку, которая содержит ряд символов. Структуры могут содержать несколько значений различных типов. Существуют еще указатели, являющиеся переменными, которые сообщают компьютеру, куда

помещены данные. В этой главе будут рассмотрены все эти производные формы данных (кроме классов), а также операторы `new` и `delete`. Читатель узнает, как можно их использовать, чтобы управлять данными.

Краткий обзор массивов

Массив — это тип данных, который может содержать несколько значений, все одного типа. Например, массив может содержать 60 значений типа `int`, которые представляют собой данные о продажах игр за пять лет, либо 12 значений типа `short` — число дней в каждом месяце, или 365 значений типа `float`, которые отражают ваши расходы на продукты питания в течение каждого дня года. Каждое значение сохраняется в отдельном элементе массива, и компьютер хранит все элементы массива последовательно в памяти.

Для создания массива используется оператор объявления. Объявление массива должно содержать три аргумента:

- Тип каждого элемента
- Название массива
- Число элементов в массиве

В C++ это реализуется путем изменения объявления простой переменной путем добавления скобок, которые содержат число элементов. Например, объявление

```
short months[12]; //создает массив из
//12 элементов типа short
```

создает массив с названием `months`, который содержит 12 элементов, каждый из которых может хранить значение типа `short`. Каждый элемент, в сущности, является переменной, и их можно обрабатывать как обычные переменные.

Общая форма для объявления массива выглядит так:

```
typeName arrayName[arraySize];
```

Выражение `arraySize`, которое задает число элементов, должно быть значением типа `const`, например 10, или другим значением типа `const`, либо постоянным выражением, например, `8 * sizeof (int)`, для которого все значения уже известны во время трансляции. В частности, `arraySize` не может быть переменной, значение которой устанавливается в то время, когда программа выполняется. Однако далее в этой главе вы увидите, как использовать оператор `new`, чтобы обойти это ограничение.

МАССИВ КАК ПРОИЗВОДНЫЙ ТИП ДАННЫХ

Массив именуется производным типом данных, потому что он основан на другом типе данных. Вы не можете просто объявить, что что-то является массивом; это всегда должен быть массив какого-либо специфического типа. Не существует обобщенного типа массива. Вместо этого имеются много специфических типов массивов, таких как массив типа `char` или `long`. Например, рассмотрим следующее объявление:

```
float loans[20];
```

Тип `loans` – не "массив"; скорее, это "массив элементов типа `float`". Это подчеркивает, что массив `loans` является производным от типа данных `float`.

Многие полезные свойства массива обусловлены тем фактом, что можно обращаться к элементам массива по отдельности. Для этого нужно использовать **субиндекс**, или **индекс**, для перечисления элементов. Массив в C++ нумеруется, начиная с 0. (Это правило нельзя изменить. Пользователям Pascal и BASIC придется приспособливаться). В C++ используется запись с индексом в квадратных скобках, чтобы определить элемент массива. Например, `months[0]` – первый элемент массива `months`, а `months[11]` – последний элемент. Обратите внимание, что индекс последнего элемента на единицу меньше, чем размер массива (рис. 4.1). Таким образом, объявление массива дает возможность создать множество переменных одним объявлением, а затем можно использовать индекс, чтобы идентифицировать отдельные элементы.

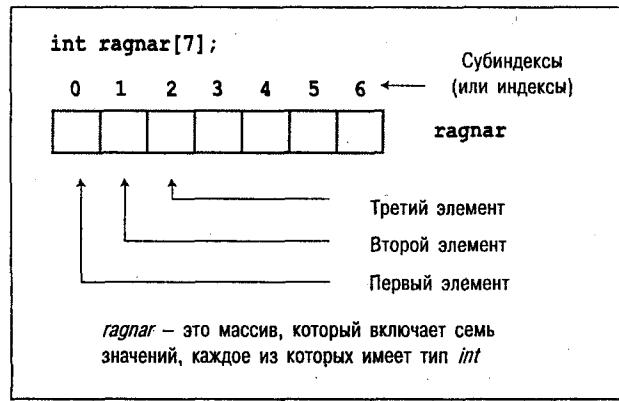


РИСУНОК 4.1 Создание массива.

В программе анализа батата, которая приведена в листинге 4.1, показано несколько свойств массивов, включая объявление массива, присвоение значений элементам массива и инициализацию массива.

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Текущие версии C++, также как С стандарта ANSI, позволяют инициализировать обычные массивы, определенные в функции. Однако в некоторых прежних реализациях, которые используют транслятор C++ вместо настоящего компилятора, транслятор C++ создает код С для компилятора С, который не полностью совпадает с компилятором С стандарта ANSI. В этом случае выводится сообщение об ошибках, как в следующем примере для системы Sun C++ 2.0:

```
"arrayone.cc", line 10: sorry, not
    implemented: initialization of yamcosts
        (automatic aggregate) Compilation failed
```

Для устранения проблемы применяется ключевое слово `static` в объявлении массива:

```
// инициализация, выполняемая до
// принятия стандарта ANSI
static int yamcosts[3] = {20, 30, 5};
```

Ключевое слово `static` заставляет компилятор использовать другую схему памяти для сохранения массива, а именно ту, которая допускает инициализацию даже для более ранних версий языка С, чем ANSI С. В главе 8 описывается ключевое слово `static` в разделе, касающемся классов памяти.

Результат выполнения программы из листинга 4.1:

```
Total yams = 21
The package with 8 yams costs 30 cents per
yam.
The total yam expense is 410 cents.
Size of yams array = 12 bytes.
Size of one element = 4 bytes.
```

Листинг 4.1 Программа arrayone.cpp.

```
// arrayone.cpp - маленький массив типа int
#include <iostream>
using namespace std;
int main()
{
    int yams[3]; // создается массив из трех элементов
    yams[0] = 7; // присваивается значение первому элементу
    yams[1] = 8;
    yams[2] = 6;
    int yamcosts[3] = {20, 30, 5}; // создается и инициализируется массив

// Примечание: если компилятор или транслятор
// C++ не может инициализировать этот массив, используйте
// static int yamcosts[3] вместо int yamcosts[3]

    cout << "Total yams = ";
    cout << yams[0] + yams[1] + yams[2] << "\n";
    cout << "The package with " << yams[1] << " yams costs ";
    cout << yamcosts[1] << " cents per yam.\n";

    int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
    total = total + yams[2] * yamcosts[2];
    cout << "The total yam expense is " << total << " cents.\n";

    cout << "\nSize of yams array = " << sizeof yams;
    cout << " bytes.\n";
    cout << "Size of one element = " << sizeof yams[0];
    cout << " bytes.\n";
    return 0;
}
```

Примечания к программе

Прежде всего программа создает трехэлементный массив, называемый `yams`. Поскольку `yams` имеет три элемента, они пронумерованы от 0 до 2, и программа `arrayone.cpp` использует значения индекса в диапазоне 0–2, чтобы присвоить значения трем отдельным элементам. Каждый отдельный элемент `yams` имеет тип `int` со всеми правами и привилегиями типа данных `int`, и, таким образом, `arrayone.cpp` может присваивать значения элементам, а также добавлять, умножать и отображать элементы.

Программа использует длинный способ присвоения значений элементам `yam`. С++ также позволяет инициализировать элементы внутри инструкции объявления. В листинге 4.1 используется этот короткий способ, чтобы присвоить значения массиву `yamcosts`:

```
int yamcosts[3] = {20, 30, 5};
```

Просто задайте разделенный запятыми список начальных значений, заключенный в скобки. Пробелы в списке необязательны. Если не инициализируется массив, определенный внутри функции, значения элемента остаются неопределенными. Это свидетельствует о том, что элемент принимает любое значение, которое предварительно находилось в том же месте в памяти.

Затем программа использует значения массива в нескольких вычислениях. Эта часть программы выглядит беспорядочной из-за индексов и скобок. Цикл `for`, который рассматривается в главе 5, предоставляет мощный способ для обращения с массивами и устраниет необходимость записывать каждый элемент явно. Тем временем мы будем использовать малые массивы.

Оператор `sizeof`, как вы помните, возвращает размер в байтах типа или объекта данных. Заметьте, что если используется оператор `sizeof` с названием массива, то выводится число байтов в целом массиве. Но если используется `sizeof` с элементом массива, получается размер в байтах элемента. Это поясняет то, что `yams` — массив, но `yams[1]` — всего лишь элемент типа `int`.

Еще об инициализации массива

Язык С++ имеет несколько правил относительно инициализации массива. Они носят ограничительный характер, когда массиву присваиваются значения, и определяют, что случится, если число элементов массива не соответствует числу значений в строке инициализации. Давайте исследуем эти правила.

Можно использовать форму инициализации только при определении массива. Вы не можете использовать ее позже, а также невозможно присвоить один массив целиком другому:

```
int cards[4] = {3, 6, 8, 10}; // правильно
int hand[4];                // правильно
hand[4] = {5, 6, 7, 9};      // неправильно
hand = cards;                // неправильно
```

Однако можно использовать индексы и присваивать значения элементам массива индивидуально.

При инициализации массива можно присваивать меньшее количество значений, чем количество элементов в массиве. Например, следующая инструкция инициализирует только первые два элемента массива `hoteltips`:

```
float hoteltips[5] = {5.0, 2.5};
```

Если массив инициализируется частично, то транслятор устанавливает оставшиеся элементы равными нулю. Таким образом, можно присвоить всем элементам массива нуль — нужно только явно присвоить нуль первому элементу и позволить компилятору инициализировать нулем оставшиеся элементы:

```
float totals[500] = {0};
```

Если при инициализации массива квадратные скобки оставить пустыми, компилятор C++ будет подсчитывать элементы массива. Предположим, например, выполняется такое объявление:

```
short things[] = {1, 5, 3, 8};
```

Компилятор создает массив `things` из четырех элементов.

ПОЗВОЛИМ КОМПИЛЯТОРУ СДЕЛАТЬ ЭТО

Обычно не рекомендуется разрешать компилятору подсчитывать число элементов, поскольку полученное таким образом количество может отличаться от имеющегося согласно вашим предположениям. Однако этот подход может быть более безопасным для инициализации символьного массива строкой, как скоро будет показано. И если главное для вас — то, что сама программа знает, насколько большим будет массив, вы можете сделать что-нибудь типа:

```
short things[] = {1, 5, 3, 8};
int num_elements = sizeof things /
    sizeof (short);
```

Является ли это полезным или нет — зависит от обстоятельств.

Строки

Строка — это ряд символов, хранящихся в последовательных байтах памяти. В C++ есть два способа обращения со строками. Первый, позаимствованный из C и часто называемый *строкой C-стиля*, является методом, который будет рассмотрен в этой главе. Глава 15 демонстрирует альтернативный метод, основанный на библиотеке класса `string`. Тем временем идея относительно ряда

символов, сохраненных в последовательных байтах, подразумевает, что можно хранить строку в массиве символов, с тем чтобы каждый символ хранился в своем собственном элементе массива. Строки обеспечивают удобный способ хранения текстовой информации, такой как сообщения пользователю ("Пожалуйста, сообщите мне ваш секретный номер счета в Швейцарском банке") или ответы от пользователя ("Вы, должно быть, шутите"). Строки C-стиля имеют одну особенность: последний символ каждой строки — нулевой символ. Этот символ пишется как `\0` и является символом, имеющим нулевой код ASCII; он служит для обозначения конца строки. Например, рассмотрите следующие два объявления:

```
char dog[5] = { 'b', 'e', 'a', 'u', 'x' };
// не строка!
char cat[5] = {'f', 'a', 't', 's', '\0'};
// строка!
```

Оба массива являются массивами символов, но только второй из них — строка. Нулевой символ играет фундаментальную роль в строках C-стиля. Например, C++ имеет много функций, которые обрабатывают строки, включая те, которые используются с конструкцией `cout`. Они все обрабатывают строки символ за символом, пока не достигнут нулевого символа. Если вы укажете, чтобы `cout` отобразил строку так же красиво, как это делается с помощью массива `cat` (см. выше), `cout` отобразит первые четыре символа, обнаружит нулевой символ и остановится. Но если указать конструкции `cout` отобразить массив `dog` (см. выше), который не является строкой, то `cout` выведет эти пять символов из массива и затем продолжит просматривать память байт за байтом, интерпретируя каждый байт как символ, который следует печатать, пока не будет достигнут нулевой символ. Поскольку нулевые символы, которые в действительности являются установленными в нуль байтами, имеют тенденцию встречаться в памяти часто, особого вреда не будет; тем не менее, не следует обрабатывать массивы символов, не являющиеся строками, как строки.

В примере с массивом `cat` выполняется утомительная инициализация массива строкой — применяются одинарные кавычки, кроме того, необходимо помнить про нулевой символ. Не волнуйтесь. Имеется лучший способ присвоения строки символьному массиву. Просто используйте строку в кавычках, называемую строковой константой или строковым литералом, как показано в следующем примере:

```
char bird[10] = "Mr. Cheep";
// подразумевается в конце \0
char fish[] = "Bubbles";
// позволим считать компилятору
```

Строки в кавычках всегда неявно включают конечный нулевой символ, так что не требуется его писать

(рис. 4.2). Кроме того, различные средства ввода C++ для чтения строки с клавиатуры в массив символов автоматически добавляют конечный нулевой символ. (Если при выполнении программы, приведенной в листинге 4.1, вы обнаружите, что необходимо использовать ключевое слово `static`, чтобы инициализировать массив, необходимо также использовать его с этими символьными массивами.)

Конечно, необходимо удостовериться, что массив достаточно большой, чтобы вместить все символы строки, включая нулевой символ. Инициализация символьного массива строковыми константами — это тот случай, когда может быть более безопасно позволить компилятору подсчитать количество элементов. В создании массива, имеющего больший размер, чем строка, нет никакого вреда, правда, при этом потребуется больше места. Дело в том, что функции, которые работают со строками, управляются позицией нулевого символа, а не размером массива. C++ не накладывает никаких ограничений на длину строки.

ПОМНИТЕ

При определении минимального размера массива, в который необходимо записать строку, не забывайте включать конечный нулевой символ в результат подсчета.

Обратите внимание на то, что строковая константа (двойные кавычки) и символьная константа (одинарные кавычки) не являются взаимозаменяемыми. Символьная константа, такая как 'S', является короткой записью кода для символа. В кодах ASCII запись 'S' — это просто другой способ записи числа 83. Таким образом, оператор

```
char shirt_size = 'S'; // отлично
```

присваивает 83 переменной `shirt_size`. Но "S" представляет собой строку, состоящую из двух символов, S и \0. Даже хуже, "S" фактически представляет собой адрес памяти, в котором сохранена строка. Так что оператор

```
char shirt_size = "S"; // запрещенное
// смешение типов
```

```
char boss[8] = "Bozo";
```

B	o	z	o	\0	\0	\0	\0
---	---	---	---	----	----	----	----

Нулевой символ
автоматически
добавляется в
конец

Оставшимся
элементам
присвоено
значение \0

пытаются назначать адрес памяти переменной `shirt_size`. Поскольку адрес — это отдельный тип в C++, компилятор C++ не допустит этого. (Мы возвратимся этому пункту позже, после того, как будут рассмотрены указатели). Но язык C, который является более снискожительным к проверке согласования типов, разрешает это объявление, отображая при этом предупреждение, а результатом будет являться "мусор".

Конкатенация строк

Иногда строка может быть слишком длинной, чтобы полностью вместиться в одной программной строке. C++ дает возможность связать строковые константы, т.е. объединять две строки в кавычках в одну. Действительно, любые две строковые константы, разделенные только незаполненным пространством (пробелы, знаки табуляции и символы новой строки) автоматически объединяются в одну. Таким образом, все показанные далее инструкции вывода эквивалентны друг другу:

```
cout << "I'd give my right arm to be"
      " a great violinist.\n";
cout << "I'd give my right arm to be a
      "great violinist.\n";
cout << "I'd give my right ar"
      "m to be a great violinist.\n";
```

Обратите внимание, что при объединении не добавляется никаких пробелов к соединенным строкам. Первый символ второй строки немедленно следует за последним символом, не считая символов \0 первой строки. Символ \0 первой строки заменен первым символом второй строки.

Использование строк в массиве

Два наиболее обычных пути записи строки в массив состоят в том, чтобы инициализировать массив строковой константой и считывать в массив результаты ввода с клавиатуры или из файла. Листинг 4.2 демонстрирует эти подходы, при этом одному массиву присваивается строка в кавычках и используется `cin`, что позволяет поместить вводимую строку во второй массив. Программа также использует стандартную библиотечную функцию `strlen()`, чтобы узнать длину строки. Стандартный заголовочный файл `cstring` (или `string.h` для более ранних реализаций) предоставляет объявления для этой и многих других функций, связанных со строками.

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если ваша система не поддерживает заголовочный файл `cstring`, попробуйте использовать более раннюю версию `string.h`.

РИСУНОК 4.2 Присваивание строки массиву

Листинг 4.2 Программа strings.cpp.

```
// strings.cpp - сохранение строк в массиве
#include <iostream>
#include <cstring> // для функции strlen()
using namespace std;
int main()
{
    const int Size = 15;
    char name1[Size]; // пустой массив
    char name2[Size] = "C++owboy"; // инициализированный массив
// Примечание: некоторые реализации могут требовать ключевое слово
// для инициализации массива name2
    cout << "Howdy! I'm " << name2;
    cout << "! What's your name?\n";
    cin >> name1;
    cout << "Well, " << name1 << ", your name has ";
    cout << strlen(name1) << " letters and is stored\n";
    cout << "in an array of " << sizeof name1 << " bytes.\n";
    cout << "Your initial is " << name1[0] << ".\n";
    name2[3] = '\0'; // нулевой символ
    cout << "Here are the first 3 characters of my name: ";
    cout << name2 << "\n";
    return 0;
}
```

Результат выполнения:

```
Howdy! I'm C++owboy! What's your name?
Basicman
Well, Basicman, your name has 8 letters and
is stored in an array of 15 bytes.
Your initial is B.
Here are the first 3 characters of my
name: C++
```

Примечания к программе

Чему можно научиться из этого примера? Сначала обратите внимание на то, что оператор `sizeof` дает размер полного массива, 15 байтов, но функция `strlen()` возвращает размер сохраненной в массиве строки, а не размер массива непосредственно. Функция `strlen()` также считает только видимые символы, а не нулевой символ. Таким образом, она возвращает значение 8, а не 9 для длины строки `Basicman`. Если `cosmic` — строка, минимальный размер массива для хранения этой строки будет `strlen(cosmic) + 1`.

Поскольку `name1` и `name2` — массивы, можно использовать индекс, чтобы обратиться к отдельным символам в массиве. Например, программа использует `name1[0]`, чтобы найти первый символ в массиве. Кроме того, программа устанавливает значение `name2[3]`, равное нулевому символу. Это заставляет строку завершиться после трех символов, несмотря на то, что в массиве остается большее количество символов (рис. 4.3).

Обратите внимание на то, что программа использует символьскую константу для хранения размера массива. Часто размер массива появляется в нескольких операторах программы. Использование символьской константы для представления размера массива упрощает пересмотр программы в целях использования массива другого размера; нужно заменить значение один раз в том месте, где определяется символьская константа.

Возможные нюансы при строковом вводе

Программа `strings.cpp` имеет недостаток, который был замаскирован при использовании методики тщательно отобранного ввода примеров. В листинге 4.3 удаляется "маскировка" и демонстрируется, что при вводе строк могут возникать проблемы.

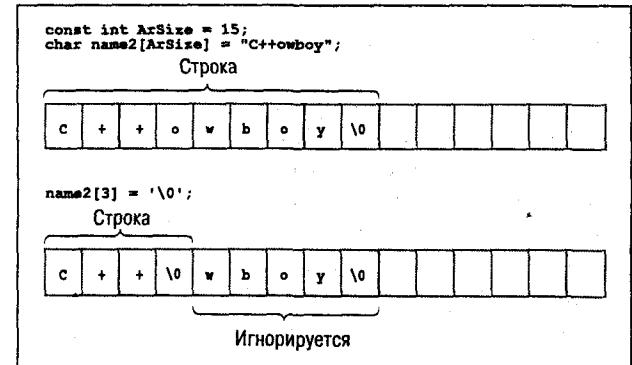


РИСУНОК 4.3 Укорачивание строки с помощью '\0'.

Листинг 4.3 Программа instr1.cpp.

```
// instr1.cpp - чтение более чем одной строки
#include <iostream>
using namespace std;
int main()
{
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];
    cout << "Enter your name:\n";
    cin >> name;
    cout << "Enter your favorite dessert:\n";
    cin >> dessert;
    cout << "I have some delicious "
        << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Цель использования листинга 4.3 простая — ввести с клавиатуры имя пользователя и его любимый десерт и затем отобразить информацию. Вот примерный диалог:

```
Enter your name:
Alistair Dreeb
Enter your favorite dessert:
I have some delicious Dreeb for you,
Alistair.
```

Мы не получили возможности ответить на приглашение ввести название десерта! Программа показала его и затем стала выполняться дальше, чтобы вывести заключительную строку.

Проблема состоит в том, как `cin` определяет, где завершен ввод строки. Вы не можете вводить нулевой символ с клавиатуры, так что `cin` нуждается в каких-то других средствах для определения конца строки. Методика `cin` должна использовать пробельные символы — символы пробела, табуляции и символы новой строки — для того, чтобы очертить строку. Это означает, что `cin` читает только одно слово, когда оно становится входным для символьного массива. После этого `cin` автоматически добавляет конечный нулевой символ, когда помещает строку в массив.

Практическим результатом в этом примере является то, что `cin` читает *Alistair* как полную первую строку и помещает ее в массив `name`. В результате *Dreeb* все еще остается во входной очереди. Когда `cin` ищет входную очередь для ответа на вопрос о любимом десерте, то все еще находит там *Dreeb*. Тогда `cin` "проглатывает" *Dreeb* и помещает его в массив `dessert` (рис. 4.4).

Другая проблема, которая не проявилась в приведенном диалоге, — это то, что входная строка могла бы оказаться длиннее, чем массив назначения. Использование `cin` в этом примере не дает возможности предотвратить размещение 30-символьной строки в 20-символьном массиве.

Многие программы зависят от ввода строки, так что стоит исследовать эту тему далее. Нам придется исследовать некоторые из наиболее перспективных особенностей `cin`, которые описаны в главе 16.

Строчно-ориентированный ввод: `getline()` и `get()`

Чтобы иметь возможность вводить целые фразы как строку вместо одиночных слов, нужен другой подход к вводу строк: требуется строчно-ориентированный метод вместо текстового метода. Вам повезло, для класса `istream`, примером которого является `cin`, имеются некоторые строчно-ориентированные функции — элементы класса. Функция `getline()`, например, читает целую строку, используя переданный с помощью клавиши ENTER символ новой строки, чтобы пометить конец ввода. Вы вызываете этот метод, используя `cin.getline()` как обращение к функции. Функция требует двух параметров. Первый параметр — название массива, предназначенному для того, что бы сохранить вводимую строку, а второй параметр ограничивает количество символов, которые нужно считать. Если этот предел, скажем, 20, функция читает не больше чем 19 символов, оставляя участок памяти, чтобы автоматически добавить нулевой символ в конец. Функция-элемент `getline()` останавливает чтение ввода, когда достигает числового предела или когда читает символ новой строки, независимо от того, какое из двух событий происходит первым.

Например, предположим, что нужно использовать `getline()`, чтобы считать название в 20-элементный массив `name`. Вам нужно было бы выполнить такой запрос:

```
cin.getline(name, 20);
```

Этот запрос считывает полную строку в массив `name` при условии, что строка состоит из 19 или меньшего количества символов. (Функция-элемент `getline()` также имеет необязательный третий параметр, который описывается в главе 16).

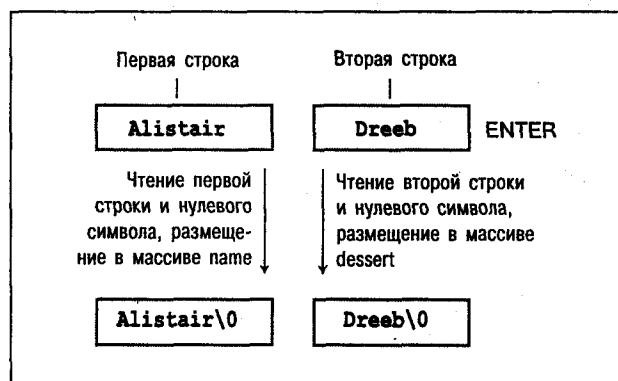


РИСУНОК 4.4 Взгляд `cin` на ввод строки.

Листинг 4.4 — это модификация листинга 4.3, выполненная таким образом, чтобы использовать `cin.getline()` вместо простого `cin`. В остальном программа осталась без изменений.

Листинг 4.4 Программа instr2.cpp.

```
// instr2.cpp - чтение более одного слова
// с помощью функции getline
#include <iostream>
using namespace std;
int main()
{
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";
    cin.getline(name, ArSize); //чтение с
    //помощью символа новой строки
    cout << "Enter your favorite dessert:\n";
    cin.getline(dessert, ArSize);
    cout << "I have some delicious "
        << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторых ранних версиях C++ не полностью осуществлялись все аспекты настоящего пакета ввода-вывода C++. В частности, функция-элемент `getline()` доступна не всегда. Если это вас волнует, просто прочтите этот пример и переходите к следующему, в котором используется функция-элемент, являющаяся предшественницей функции `getline()`. Прежние реализации Turbo C++ выполняют `getline()` немного по-другому, так, чтобы она сохраняла символ новой строки в строке. Microsoft Visual C++ 5.0 содержит ошибку для функции `getline()`, которая находится в заголовочном файле `iostream`, но не в версии `ostream.h`.

Результат выполнения программы:

```
Enter your name:
Melanie Poops
Enter your favorite dessert:
Raspberry Torte
I have some delicious Raspberry Torte for
you, Melanie Poops.
```

Теперь программа читает полные имена и отображает для пользователя заказанные десерты! Функция `getline()` воспринимает строку за один раз. Она осуществляет ввод с помощью символа новой строки, который обозначает конец строки, однако она не сохраняет символ новой строки. Вместо этого она заменяет его нулевым символом во время сохранения строки (рис. 4.5).

Давайте попробуем другой подход. В классе `istream` есть другая функция-элемент, которая называется `get()` и имеет несколько разновидностей. Одна из ее разновидностей очень похожа по своим действиям на `getline()`. Ей требуются те же самые параметры, интерпретируются

они тем же самым образом, и читает она до конца строки. Но вместо чтения и удаления символа новой строки `get()` оставляет этот символ во входной очереди. Предположим, что мы используем два запроса `get()` подряд:

```
cin.get(name, ArSize);
cin.get(dessert, Arsize); // проблема
```

Поскольку в результате первого вызова символ новой строки остается во входной очереди, то он будет первым символом, который доступен для второго запроса. Таким образом, `get()` заключает, что достигла конца строки, не найдя ничего для чтения. Без дополнительной помощи `get()` не может преодолеть этот символ новой строки.

К счастью, существует помощь в форме разновидности функции `get()`. Запрос `cin.get()` (без параметров) считывает один следующий символ, даже если это символ новой строки — так что вы можете использовать его для передачи символа новой строки и приготовиться к приему следующей строки ввода. Вот как выглядит эта последовательность:

```
cin.get(name, ArSize); //читает первую
//строчку
cin.get(); //читает символ новой строки
cin.get(dessert, Arsize); //читает вторую
//строку
```

Другой способ использования `get()` — *конкатенировать*, или объединить, эти две функции-элементы класса следующим образом:

```
cin.get(name, ArSize).get(); //конкатенация
//функций-элементов
```

Это возможно благодаря тому, что `cin.get(name, ArSize)` возвращает объект `cin`, который затем используется как объект, вызывающий функцию `get()`.

Код

```
char name[10];
cout << "Enter your name: ";
cin.getline(name, 10);
```

В ответ пользователь печатает `Jud`, затем нажимает клавишу `Enter`

```
Enter your name: Jud ENTER
```

Функция `cin.getline()` в ответ считывает `Jud`, читает новую строку, которая была сгенерирована после нажатия клавиши `Enter`, и заменяет ее нулевым символом.

J	u	d	\0					
---	---	---	----	--	--	--	--	--

РИСУНОК 4.5 Функция `getline()` читает и заменяет символ новой строки.

Проще говоря, выражение

```
cin.getline(name1, ArSize).getline(name2,
    ArSize);
```

считывает две последовательные входные строки в массивы `name1` и `name2`; это эквивалентно выполнению двух отдельных вызовов функции `cin.getline()`.

В листинге 4.5 используется конкатенация. В главе 10 вы узнаете, как включить эту особенность в определения класса.

Листинг 4.5 Программа instr3.cpp.

```
// instr3.cpp - чтение более чем одного
// слова с помощью функций get() & get()
#include <iostream>
using namespace std;
int main()
{
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];
    cout << "Enter your name:\n";
    cin.get(name, ArSize).get();           //чтение
                                         //строки, символа новой строки
    cout << "Enter your favorite dessert:\n";
    cin.get(dessert, ArSize).get();
    cout << "I have some delicious "
         << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Некоторые более старые версии C++ не обеспечивают выполнение функции `get()` без параметров. Однако они все же обеспечивают выполнение другого варианта функции `get()`, который требует использования единственного параметра типа `char`. Для того чтобы использовать его вместо функции `get()` без параметров, необходимо сначала объявить переменную типа `char`:

```
char ch;
cin.get(name, ArSize).get(ch);
```

Вы можете использовать этот код вместо того, который был показан в листинге 4.5. В главах 5, 6 и 16 более подробно описываются разновидности функции `get()`.

Результат выполнения:

```
Enter your name:
Mai Parfait
Enter your favorite dessert:
Chocolate Mousse
I have some delicious Chocolate Mousse for
you, Mai Parfait.
```

Один момент, на который нужно обратить внимание, состоит в том, как C++ допускает множественные версии функций при условии, что они имеют различные

списки параметров. Если вы используете, скажем, `cin.get(name, ArSize)`, компилятор отмечает, что вы используете форму, которая помещает строку в массив, и устанавливает соответствующую функцию-элемент. Если вместо этого используется `cin.get()`, компилятор понимает, что вам нужна форма, которая читает один символ. В главе 8 исследуется эта особенность, которая называется переопределением функции.

Зачем вообще использовать `get()` вместо `getline()`? Во-первых, в более ранних реализациях может не быть функции `getline()`. Во-вторых, функция `get()` позволяет вам быть немного более осторожным. Предположим, например, что `get()` использована для того, чтобы считывать строку в массив. Как вы сможете узнать, что эта функция прочла целую строку, прежде чем остановилась по причине того, что массив заполнен? Посмотрите на очередной вводимый символ. Если это символ новой строки, значит, была считана целая строка. Если это не символ новой строки, значит, в строке все еще имеется что-то для ввода. В главе 16 исследуется эта методика. Короче говоря, функция `getline()` немного легче использовать, но `get()` делает более простым исправление ошибок.

Пустые строки и другие проблемы

Что происходит после того, как `getline()` или `get()` считывает пустую строку? Первоначально практика состояла в том, что следующая инструкция ввода начинается с того места, где остановилась последняя функция `getline()` или `get()`. Однако сейчас, после того как функция `get()` (но не `getline()`) читает пустую строку, она устанавливает флаг называемый *failbit* (бит ошибки). Смысл этого действия состоит в том, что дальнейший ввод блокирован, но его можно восстановить с помощью следующей команды:

```
cin.clear();
```

Другая потенциальная проблема состоит в том, что вводимая строка может быть больше, чем выделенное для нее место. Если вводимая строка включает большие указанного числа символов, как `get()`, так и `getline()` оставляют оставшиеся символы во входной очереди. Однако `getline()` дополнительно устанавливает `failbit` и прекращает дальнейший ввод.

В главах 5, 6 и 16 исследуются эти свойства и рассказывается, как программа оперирует с ними.

Смешанный строчно-числовой ввод

Смешанный строчно-числовой ввод строки может вызывать проблемы. Рассмотрим простую программу, приведенную в листинге 4.6.

Листинг 4.6 Программа numstr.cpp.

```
//Numstr.cpp - чередование ввода чисел и строк
#include <iostream>
using namespace std;
int main()
{
    cout << "What year was your house built?\n";
    int year;
    cin >> year;
    cout << "What is its street address?\n";
    char address[80];
    cin.getline(address, 80);
    cout << "Year built: " << year << "\n";
    cout << "Address: " << address << "\n";
    return 0;
}
```

Выполнение этой программы будет выглядеть примерно так:

```
What year was your house built?
1966
What is its street address?
Year built: 1966
Address:
```

Вы никогда не получите возможность ввести адрес. Проблема состоит в том, что когда `cin` читает год, то оставляет символ новой строки, сгенерированный с помощью клавиши ENTER, во входной очереди. Затем `cin.getline()` воспринимает символ новой строки, как будто это пустая строка, и назначает ее массиву адреса. Исправить эту ошибку можно, читая и отбрасывая символ новой строки перед чтением адреса. Это можно сделать несколькими способами, включая использование `get()` без параметра или с параметром типа `char`, как описано в предшествующем примере. Вы можете выполнить этот запрос отдельно:

```
cin >> year;
cin.get(); // или cin.get(ch);
```

Или же можно выполнять конкатенацию, извлекая пользу из того факта, что выражение

```
expression cin >> year
```

возвращает объект `cin`:

```
(cin >> year).get();
// или (cin >> year).get(ch);
```

Если внести одно из этих изменений в листинг 4.6, программа начнет работать правильно:

```
What year was your house built?
1966
What is its street address?
43821 Unsigned Short Street
Year built: 1966
Address: 43821 Unsigned Short Street
```

Программы, написанные на C++, для обработки строк часто вместо массивов используют указатели. Мы займемся строками после того, как немного изучим указатели. Тем временем давайте посмотрим на другой производный тип — структуру.

Краткий обзор структур

Предположим, что нужно сохранить информацию об игроке в баскетбол. Нужно сохранять такие данные: его имя, заработка, рост, вес, средний рейтинг, процент попаданий и т.д. Вы хотели бы иметь некоторую форму данных, которая может содержать всю эту информацию в одном блоке. Массив с этим не справится. Хотя массив может содержать несколько элементов, каждый элемент должен быть одного типа. Иначе говоря, один массив может содержать двадцать элементов типа `int`, а другой — десять элементов типа `float`, но один и тот же массив не может содержать элементы типа `int` и `float`.

Ответом на ваше желание (относительно сохранения информации о баскетболисте) будет *структурата C++*. Структура — более универсальная форма данных, чем массив, поскольку одна структура может содержать элементы нескольких типов данных. Это предоставляет возможность объединить ваше представление данных, сохраняя всю связанную с баскетболистом информацию в единственной переменной структуры. Если необходимо включить данные относительно целой команды, можно использовать массив структур. Тип структуры также является мостиком к классу, который является основой ООП языка C++. Изучение структур позволит нам гораздо глубже понять, что представляет собой класс.

Структура — определяемый пользователем тип данных; объявление структуры служит для того, чтобы определить свойства типов данных. После того как вы определите тип, можно создавать переменные этого типа. Таким образом, создание структуры — процесс, состоящий из двух частей. Сначала вы описываете структуру, т.е. описываете и маркируете различные типы данных, которые могут содержаться в структуре. Затем можно создавать структурные переменные или, более широко, структурные объекты данных, которые повторяют схему описания.

Например, предположим, что компания Bloataire хочет создать некий тип для описания работников своей производственной линии по разработке надувных изделий. В частности, этот тип должен содержать название элемента, его объем в кубических футах и его продажную цену. Вот описание структуры, удовлетворяющей этим потребностям:

```

struct inflatable //описание структуры
{
    char name[20]; //элемент типа array
    float volume; //элемент типа float
    double price; //элемент типа double
};

```

Ключевое слово **struct** указывает на то, что код определяет формат структуры. Идентификатор **inflatable** — название, или тег, для этой формы; это делает **inflatable** названием нового типа данных. Таким образом, теперь можно создавать переменные типа **inflatable** так же, как переменные типа **char** или **int**. Между фигурными скобками находится список типов данных, которые будут содержаться в структуре. Каждый элемент списка — это оператор объявления. Здесь можно использовать любой из типов данных C++, включая массивы и другие структуры. В этом примере используется массив типа **char**, удобный для сохранения строки, а также **float** и **double**. Каждый отдельный элемент в списке называется **элементом структуры**, так что структура **inflatable** включает три элемента (рис. 4.6).

Если у вас есть шаблон, можно создавать переменные этого типа:

```

inflatable hat; //hat-структурная
                 //переменная типа inflatable
inflatable woopie_cushion; //структурная
                           //переменная типа inflatable
inflatable mainframe; //структурная
                      //переменная типа inflatable

```

Если вы знакомы со структурами в языке С, вы обратите внимание (вероятно, с удовлетворением) на то, что C++ позволяет опускать ключевое слово **struct**, когда объявляется переменные структуры:

```

struct inflatable goose; //в С ключевое
                          //слово struct требуется
inflatable vincent; //в C++ ключевое
                     //слово struct не требуется

```

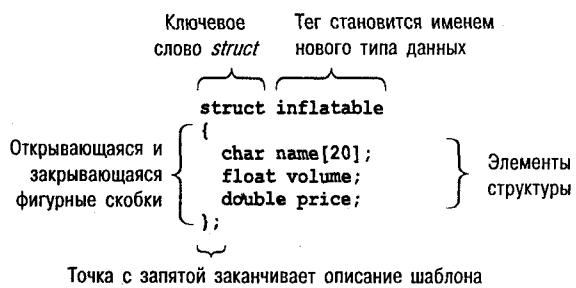


РИСУНОК 4.6 Части описания структуры.

В C++ тег структуры используется точно так же, как и название базового типа данных. Это изменение подчеркивает то, что объявление структуры определяет новый тип. Благодаря этому пропуск ключевого слова **struct** уже не является ошибкой.

Учитывая, что **hat** имеет тип **inflatable**, для обращения к отдельным элементам можно использовать оператор принадлежности **(.)**. Например, **hat.volume** обращается к элементу **volume** структуры, а **hat.price** обращается к элементу **price**. Аналогично **vincent.price** — элемент **price** переменной **vincent**. Короче говоря, названия элементов предоставляют возможность обратиться к элементам структуры во многом так же, как индексы дают возможность обратиться к элементам массива. Поскольку элемент **price** объявлен как тип **double**, **hat.price** и **vincent.price** эквивалентны переменным типа **double** и их можно использовать так же, как используется обыкновенная переменная типа **double**. Таким образом, **hat** — это структура, но **hat.price** — элемент типа **double**. Между прочим, этот метод используется для обращения к функциям-элементам класса, так же как **cin.getline()** инициализируется с помощью метода, который используется для обращения к переменным элементам структуры, таким как **vincent.price**.

Листинг 4.7 поясняет моменты, относящиеся к структуре. Здесь также показано, как ее можно инициализировать.

Листинг 4.7 Программа structur.cpp.

```

// structur.cpp - простая структура
#include <iostream>
using namespace std;
struct inflatable // шаблон структуры
{
    char name[20];
    float volume;
    double price;
};

int main()
{
    inflatable guest =
    {
        "Glorious Gloria", //имя
        1.88, //объем
        29.99 //цена
    }; //quest — переменная структура типа
      //inflatable. Инициализируется
      //указанными значениями

    inflatable pal =
    {
        "Audacious Arthur",
        3.12,
        32.99
    }; //вторая переменная типа inflatable

// Примечание: некоторые реализации требуют
// применения static inflatable guest =

```

```

cout << "Expand your guest list with "
    << guest.name;
cout << " and " << pal.name << "!\n";
// pal.name — это имя переменной pal
cout << "You can have both for $";
cout << guest.price + pal.price << "!\n";
return 0;
}

```

ЗАМЕЧАНИЯ ПО ПОВОДУ СОВМЕСТИМОСТИ

Поскольку некоторые реализации еще не поддерживают возможность инициализировать обычновенный массив, определенный в функции, они также не поддерживают возможность инициализировать обычновенную структуру, определенную в функции. Опять-таки, решением в этой ситуации будет использование ключевого слова `static` при объявлении.

Результат выполнения программы:

```

Expand your guest list with Glorious Gloria
and Audacious Arthur!
You can have both for $62.98!

```

Примечания к программе

Один важный момент состоит в том, чтобы определить, где надо размещать объявление структуры. Для `structur.cpp` есть два варианта. Вы могли бы поместить объявление внутри функции `main()`, сразу после открывающейся фигурной скобки. Второй вариант — тот, который реализован здесь, — поместить объявление вне функции `main()`, перед ней. Когда объявление размещается вне любой функции, оно называется *внешним объявлением*. Для этой программы нет никакой практической разницы между этими двумя вариантами. Но для программ, состоящих из двух или более функций, разница может быть критической. Внешнее объявление может использоваться всеми функциями, следующими за ним, тогда как внутреннее объявление может быть использовано только функцией, в которой оно содержится. Чаще всего используется внешнее объявление структуры, так, чтобы все функции могли бы использовать структуру этого типа (рис. 4.7).

Переменные также могут быть определены как внутренние или внешние, при этом необходимо, чтобы внешние переменные были общедоступными для функций. (В главе 8 этот вопрос будет рассмотрен более детально.) В C++ нет возможности использовать внешние переменные, однако поощряется использование внешних объявлений структуры. Кроме того, часто имеет смысл объявлять символические внешние константы.

Далее обратите внимание на процедуру инициализации:

```

inflatable guest =
{
    "Glorious Gloria",      // имя
    1.88,                   // объем
    29.99                  // цена
};

```

Как и с массивами, используется список значений, разделенных запятыми и заключенных в пару фигурных скобок. Программа размещает по одному значению в строке, но их можно размещать все в одной строке. Только не забудьте разделять элементы запятой:

```
inflatable duck = {"Daphne", 0.12, 9.98};
```

Вы можете инициализировать каждый элемент структуры соответствующим начальным значением. Например, элемент `name` — символьный массив, так что можно инициализировать его строкой.

Каждый элемент структуры обрабатывается как переменная этого типа. Таким образом, `pal.price` — переменная типа `double`, а `pal.name` — массив элементов типа `char`. И когда программа использует `cout`, чтобы вывести `pal.name`, то отображает этот элемент как строку. Между прочим, так как `pal.name` — символьный массив, можно использовать индексы, чтобы обратиться к отдельным символам в массиве. Например, `pal.name[0]` — это символ A. Но `pal[0]` — это бессмыслица, потому что `pal` — это структура, а не массив.

Другие свойства структуры

C++ делает определяемые пользователем типы настолько близкими к встроенным типам, насколько это возможно.

```

#include <iostream>
using namespace std;
struct parts
{
    unsigned long part_number;
    float part_cost;
};

void mail ();
int main()
{
    struct perks
    {
        int key_number;
        char car[12];
    };
    perks chicken;
    perks mr_bug;
    ...
}

void mail()
{
    parts studebaker;
    ...
}

Переменная типа parts — parts studebaker;
Переменная типа perks — ...
Невозможно объявить ...
здесь переменную ...
типа perks

```

РИСУНОК 4.7 Объявление внутренней и внешней структур.

Например, можно передавать структуры как аргументы функции, и вы можете построить функцию, которая использует структуру как возвращаемое значение. Можно также использовать оператор присвоения (`=`), чтобы присвоить одну структуру другой того же самого типа. Такое действие приводит к тому, что каждому элементу одной структуры будет присвоено значение соответствующего элемента другой структуры, даже если этот элемент — массив. Этот вид назначения называется *поэлементным присваиванием*. Мы отложим тему передачи и возвращения структур, пока не обсудим функции в главе 7. Однако теперь мы можем бегло рассмотреть назначение структуры. Листинг 4.8 дает нам один из примеров.

Листинг 4.8 Программа assgn_st.cpp.

```
// assgn_st.cpp - назначение структур
#include <iostream>
using namespace std;
struct inflatable
{
    char name[20];
    float volume;
    double price;
};
int main()
{
    inflatable bouquet =
    {
        "sunflowers",
        0.20,
        12.49
    };
    inflatable choice;
    cout << "bouquet: " << bouquet.name
        << " for $" ;
    cout << bouquet.price << "\n";
    choice = bouquet; //присвоение одной
                      //структуре другой
    cout << "choice: " << choice.name
        << " for $" ;
    cout << choice.price << "\n";
    return 0;
}
```

В результате выполнения программы получаются следующие результаты:

```
bouquet: sunflowers for $12.49
choice: sunflowers for $12.49
```

Как вы видите, здесь выполнено поэлементное присваивание — элементам структуры `choice` присвоены те же самые значения, что хранились в структуре `bouquet`.

Вы можете комбинировать определение формы структуры с созданием переменных структуры. Для этого после закрывающей фигурной скобки следует написать имя (или имена) переменной:

```
struct perks
{
    int key_number;
    char car[12];
} mr_smith, ms_jones; //две переменные perks
```

Вы даже можете инициализировать переменную, которая была создана, таким способом:

```
struct perks
{
    int key_number;
    char car[12];
} mr_glitz =
{
    7, //значение для элемента mr_glitz.key
    "Packard" //значение для элемента
               //mr_glitz.car
};
```

Однако при сохранении объявления структуры отдельно от объявлений переменных программа обычно становится более легкой для чтения и последующего выполнения.

Можно также создавать структуру без названия типа. Вы делаете это, опуская название тега, одновременно определяя форму структуры и переменной:

```
struct // без названия
{
    int x; //два элемента
    int y;
} position; //переменная структуры
```

Эта программа создает одну переменную структуры с названием `position`. Вы можете обращаться к ее элементам с помощью оператора членства, например, `position.x`, но для типа нет никакого общего названия. Впоследствии вы не сможете создавать другие переменные того же типа. В этой книге мы не будем использовать эту ограниченную форму структуры.

Кроме того факта, что программа, написанная на C++, может использовать тег структуры как название типа, структуры языка С имеют все особенности, которые мы рассмотрели пока для структур C++. Но структуры C++ являются более совершенными. В отличие от структур C, например, структуры C++ могут иметь функций-элементы в дополнение к переменным-элементам. Но эти более продвинутые особенности чаще используются с классами, чем со структурами, так что мы обсудим их тогда, когда дойдем до классов.

Массивы структур

Структура `inflatable` содержит массив (массив `name`). Можно также создавать массивы, элементы которых будут структурами. Методика — точно такая же, как и при создании массивов базовых типов данных. Например, чтобы создать массив из 100 структур `inflatable`, выполните следующее:

```
inflatable gifts[100]; //массив из 100
//структурой inflatable
```

В результате будет создан массив `gifts` типа `inflatable`. Следовательно, каждый элемент массива, такой как `gifts[0]` или `gifts[99]`, — это объект типа `inflatable`, и его можно использовать с оператором принадлежности:

```
cin >> gifts[0].volume; //используется
//элемент volume первой структуры
cout << gifts[99].price << endl; //показан
//элемент price последней структуры
```

Имейте в виду, что сам `gifts` — это массив, а не структура, так что конструкции типа `gifts.price` не являются правильными.

Чтобы инициализировать массив структур, объедините правило для инициализации массивов (заключенный в фигурные скобки, разделенный запятыми список значений для каждого элемента) с правилом для структур (заключенный в фигурные скобки, разделенный запятыми список значений для каждого элемента). Поскольку каждый элемент массива — это структура, ее значение представлено инициализацией структуры. Таким образом, вы, в конце концов, получите заключенный в фигурные скобки, разделенный запятыми список значений, каждое из которых непосредственно заключено в фигурные скобки, разделенным запятыми списком значений:

```
inflatable guests[2] = //инициализация
//массива структур
{
    {"Bambi", 0.5, 21.99}, //первая
    //структура в массиве
    {"Godzilla", 2000, 565.99} //следующая
    //структура в массиве
};
```

Как обычно, можно отформатировать вывод таким способом, который вам нравится. Обе инициализации могут располагаться на одной строке, или, например, каждая отдельная инициализация элемента структуры может получить собственную строку.

Разрядные поля

C++, подобно C, дает возможность определить элементы структуры, которые занимают указанное количество разрядов. Это может быть удобно для создания структуры данных, которая соответствует, скажем, регистру на некотором устройстве. Тип поля должен быть целым или типом перечисления (перечисления описываются далее в этой главе), а двоеточие, за которым следует число, указывает фактическое число разрядов, которые нужно использовать. Вы можете использовать неименованные поля, чтобы обеспечить интервал. Каждый элемент называется разрядным полем. Вот пример:

```
struct toggle_register
{
    int SN : 4; //четыре разряда для
    //значения SN
    int : 4; //четыре разряда
    //не используются
    bool goodIn : 1; //корректный ввод
    //один разряд
    bool goodToggle : 1; //успешное
    //присвоение
};
```

Для обращения к разрядным полям используется стандартная система обозначений структуры:

```
toggle_register tr;
...
if (tr.goodIn)
...
```

Разрядные поля обычно используются в программировании на нижнем уровне. Часто использование целочисленного типа и поразрядных операторов из приложения E представляет собой альтернативный подход.

Объединения

Объединение — это формат данных, который может содержать различные типы данных, но только один тип одновременно. Иначе говоря, в то время как структура может содержать, скажем, элементы типа *int*, *long*, *double*, объединение может содержать *или int, или long, или double*. Синтаксис выглядит так же, как и для структуры, но значение отличается. Например, рассмотрим следующее объявление:

```
union one4all
{
    int int_val;
    long long_val;
    double double_val;
};
```

Вы можете использовать переменную `one4all`, чтобы сохранить элемент типа `int`, `long` или `double`, как вы делали это в других случаях:

```
one4all pail;
pail.int_val = 15; //содержит int
cout << pail.int_val;
pail.double_val = 1.38; //содержит double,
//значение int потеряно
cout << pail.double_val;
```

Таким образом, `pail` может служить как переменная типа `int` в одном случае и как переменная типа `double` в другом. Имя элемента идентифицирует роль переменной. Поскольку объединение содержит только одно значение одновременно, оно должно иметь объем, достаточный для того, чтобы сохранить самый большой элемент. Следовательно, размер объединения соответствует размеру его самого большого элемента.

Одна из целей использования объединения — для экономии памяти, когда элемент данных может использовать два или больше формата, но не одновременно. Например, предположим, что поддерживается смешанная опись приборов, некоторые пункты в которой имеют целочисленный идентификатор, а некоторые — строковый. Тогда можно реализовать следующее объявление:

```
struct widget
{
    char brand[20];
    union id //формат зависит от типа widget
    {
        long id_num;      // тип 1 widget
        char id_char[20]; // другие типы
                           // widget
    };
    int type;
};

...
widget prize;
...

if (prize.type == 1)
    cin >> prize.id.id_num; // используется
                           // имя элемента для индикации режима
else
    cin >> prize.id.id_char;
```

Анонимное объединение не имеет никакого имени; в сущности, его элементы становятся переменными, которые разделяют один адрес. Естественно, в одно и то же время текущим может быть только один элемент:

```
struct widget
{
    char brand[20];
    union // формат зависит от типа widget
    {
        long id_num;      // тип 1 widget
        char id_char[20]; // другие типы
                           // widget
    };
    int type;
};

...
widget prize;
...

if (prize.type == 1)
    cin >> prize.id_num;
else
    cin >> prize.id_char;
```

Поскольку объединение анонимно, `id_num` и `id_char` трактуются как два элемента `prize`, которые совместно используют один и тот же адрес. Потребность в промежуточном идентификаторе устранена. Именно программа должна следить за тем, какой из вариантов активен.

Перечисления

Средство C++ `enum` предоставляет для `const` альтернативные способы создания символьических констант. Это

также позволяет определять новые типы данных, но довольно ограниченным способом. Синтаксис для использования перечисления похож на синтаксис структуры. Например, рассмотрим следующий оператор:

```
enum spectrum {red, orange, yellow, green,
               blue, violet, indigo, ultraviolet};
```

Этот оператор выполняет две функции:

- делает `spectrum` названием нового типа данных; `spectrum` называется *перечислением*, подобно тому как переменная `struct` называется структурой.
- определяет `red`, `orange`, `yellow` и т.п. как символьические константы для целых чисел от 0 до 7. Эти константы названы *перечислителями*.

По умолчанию перечислители определяются как целые значения, начиная с 0 для первого перечислителя, 1 — для второго перечислителя и т.д. Вы можете обойти значение, принятое по умолчанию, явно назначая целочисленные значения. Мы покажем, как это сделать, далее.

Можно использовать название перечисления, чтобы объявить переменную этого типа:

```
spectrum band;
```

Переменная перечисления имеет некоторые специальные свойства, которые мы рассмотрим далее.

Единственными действительными значениями, которые можно назначить переменной типа перечисления без преобразования типа, являются значения перечислителя, использованные в определении типа. Таким образом, мы имеем следующее:

```
band = blue; // правильно, blue -
              // это перечислитель
band = 2000; // неправильно, 2000 -
              // это не перечислитель
```

Таким образом, переменная `spectrum` ограничена только восьмью возможными значениями. Одни компиляторы выдают ошибку, если попытаться назначить недействительное значение, тогда как другие выдают предупреждение. Для максимальной степени совместимости нужно расценивать назначение неперечисляемого значения переменной типа перечисления как ошибку.

Для перечислений определен только оператор присваивания. В частности арифметические действия не определены:

```
band = orange;           // правильно
++band;                 // неправильно
band = orange + red;    // неправильно
...
```

Однако некоторые реализации не выполняют это ограничение. Это может привести к нарушению пределов определения типа. Например, если `band` имеет зна-

чение `ultraviolet`, или 7, то `++band`, если это законно, увеличивает `band` до 8, что уже не является правильным значением для типа `spectrum`. Опять-таки, для максимальной степени совместимости следует принять более строгие ограничения.

Перечислители имеют целочисленный тип и могут быть преобразованы в тип `int`, но элементы типа `int` не преобразовываются автоматически в тип перечисления:

```
int color = blue; //правильно, тип spectrum
                  //преобразуется в int
band = 3; //неправильно, int не
          //преобразуется в spectrum
color = 3 + red; //правильно, red
                  //преобразуется в int
...
```

Заметьте, что, хотя значение 3 соответствует перечислителю `green`, присвоение этого значения переменной `band` вызовет ошибку типа. Но присвоение `green` к `band` проходит нормально, поскольку они оба принадлежат к типу `spectrum`. Опять-таки, некоторые реализации не обеспечивают этого ограничения. В выражении `3 + red` сложение не определено для перечислителей. Однако `red` преобразовывается в тип `int`, и результатом будет тип `int`. Хотя сложение для перечислений не определено, можно использовать перечисления в арифметических выражениях.

Вы можете назначать значение типа `int` перечислению при условии, что это значение правильное и что используется явное преобразование типа:

```
band = spectrum(3); //преобразование типа 3
                     //в тип spectrum
```

Что будет, если попробовать преобразовать тип несоответствующего значения? Результат не определен, это значит, что попытка не будет помечена как ошибка, но нельзя полагаться на результат:

```
band = spectrum(40003); //не определено
```

Как можно увидеть дальше, правила, управляющие перечислениями, являются довольно ограничивающими. Практически перечисления используются чаще как способ определения связанных символьических констант, чем как способ определения нового типа. Например, можно было бы использовать перечисление, чтобы определить символьические константы для инструкции `switch` (см. главу 6). Если планируется использовать только константы и не создавать переменных типа перечисления, можно опустить название типа перечисления:

```
enum {red, orange, yellow, green, blue,
      violet, indigo, ultraviolet};
```

Установка значений перечислителя

Вы можете устанавливать значения перечислителя явно, используя оператор присвоения:

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
```

Установленные значения должны быть целыми числами. Вы также можете определять явно только некоторые из перечислителей:

```
enum bigstep{first, second = 100, third};
```

В этом случае `first` по умолчанию равен 0. Последующие неинициализированные перечислители больше своих предшественников на единицу. Так, `third` имел бы значение 101.

Наконец, можно создавать более одного перечислителя с тем же самым значением:

```
enum {zero, null = 0, one, numeroUno = 1};
```

Тут оба значения, и `zero`, и `null`, равны 0, а `one` и `numeroUno` равны 1. В более ранних версиях C++ можно было присваивать перечислителям только значения типа `int` (или значения, которые могут быть преобразованы в тип `int`), но это ограничение сменено, с тем чтобы можно было использовать значения типа `long`.

Диапазоны значений для перечислений

Первоначально единственными правильными значениями для перечисления были те, которые назначались при объявлении. Однако теперь C++ поддерживает усовершенствованную концепцию, в соответствии с которой можно вполне законно присваивать значения путем приведения типа к типу переменной перечисления. Каждое перечисление имеет диапазон, и можно присваивать любое целочисленное значение из диапазона, даже если оно не является значением перечислителя, выполняя приведение типа к переменной перечисления. Например, предположим, что `myflag` — переменная типа `bits` (как было предварительно определено). Тогда следующее выражение будет правильным:

```
myflag = bits(6); //правильно, поскольку 6
                  //находится в диапазоне bits
```

Здесь 6 — не одно из перечислений, однако оно находится в диапазоне, который определяют перечисления.

Диапазон определяется следующим образом. Сначала, чтобы найти верхний предел, возьмите самое большое значение перечислителя. Найдите самую маленькую степень двойки, превышающую это самое большое значение, вычтите единицу, и это будет верхний предел диапазона. Например, самым большим значением `bigstep`, как предварительно определено, является 101. Самая маленькая степень двойки, превышающая это

значение, — 128, так что верхний предел диапазона — 127. Затем, чтобы найти нижний предел, найдите самое маленькое значение перечислителя. Если оно нулевое или больше, нижним пределом для диапазона будет нуль. Если самый маленький перечислитель меньше нуля, используйте тот же самый подход, что и при вычислении верхнего предела, но отбросьте знак "минус". Например, если самый маленький перечислитель равен -6, следующая степень двойки (отбросим знак "минус") равна -8, а нижний предел равен -7.

Идея состоит в том, что компилятор может выбирать, сколько свободного пространства займет перечисление. Он мог бы использовать один байт или меньше для перечисления с малым диапазоном, и четыре байта для перечисления со значениями типа `long`.

Указатели и свободная память

В начале главы 3 упоминаются три фундаментальных вопроса, которые должна учитывать компьютерная программа при сохранении данных. Перечислим их здесь еще раз:

- Где хранится информация
- Какое значение там хранится
- Какая хранится информация

Вы использовали одну стратегию для достижения этих целей: определение простой переменной. Инструкция объявления дает тип и символическое название для значения. Она также заставляет программу распределять память для значения и внутренне следить за расположением.

Давайте посмотрим теперь на вторую стратегию, ту, которая становится особенно важной при разработке классов C++. Эта стратегия основана на указателях, которые являются переменными, сохраняющими адреса значений вместо непосредственно самих значений. Но перед обсуждением указателей давайте посмотрим, как явно найти адреса для обычных переменных. Нужно только применить к переменной операцию определения адреса, представленную знаком `&`, чтобы получить ее расположение; например, если `home` — переменная, `&home` является ее адресом. Листинг 4.9 демонстрирует применение этого оператора.

Листинг 4.9 Программа address.cpp.

```
// Address.cpp — использование оператора &
// для поиска адреса
#include <iostream>
using namespace std;
int main()
{
```

```
    int donuts = 6;
    double cups = 4.5;

    cout << "donuts value = " << donuts;
    cout << " and donuts address = "
        << &donuts << "\n";

// ПРИМЕЧАНИЕ: вам может потребоваться
// применить unsigned (& donuts)
// и unsigned (& cups)

    cout << "cups value = " << cups;
    cout << " and cups address = "
        << & cups << "\n";
    return 0;
}
```

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

`cout` является "умным" объектом, но одни версии "умнее" других. Таким образом, некоторые реализации могли бы быть не в состоянии распознать тип указателя. В этом случае нужно привести адрес к распознаваемому типу, например, к такому, как `unsigned int`. Соответствующее приведение типа зависит от модели памяти. Заданная по умолчанию модель памяти DOS использует 2-байтовый адрес, следовательно, `unsigned int` — правильный тип приведения. Однако некоторые модели памяти DOS используют 4-байтовый адрес, который требует приведения к типу `unsigned long`.

Вот результат выполнения программы:

```
donuts value = 6 and donuts address =
0x8566ffff
cups value = 4.5 and cups address =
0x8566ffec
```

Когда отображаются адреса, `cout` использует шестнадцатиричную систему представления, потому что она является обычной системой представления, используемой при описании памяти. Наша реализация сохраняет `cups` в ячейке памяти, номер которой меньше, чем номер `donuts`. Разность между двумя адресами будет `0x8566ffff` — `0x8566ffec`, или 8. Это имеет смысл, поскольку `cups` имеет тип `double`, который использует восемь байтов. (Если это вам интересно, эти специфические представления адресов отражают метод описания адреса памяти в ПК путем задания значения сегмента и смещения.) Значение сегмента (в этом случае `8566`) идентифицирует блок памяти, используемый для сохранения данных; это реальный адрес, разделенный на 16. Смещения (в этом случае `ffec` и `ffff`) представляют позицию памяти относительно начала сегмента. Программы ПК могут использовать 2-байтовые указатели, которые указывают только смещение, если все данные находятся в одном сегменте. Или они могут использовать 4-байтовые указатели, с первыми двумя байтами, содержащими значение сегмента, и вторыми двумя байтами, содержащими значение смещения.

Используя в таком случае обычные переменные, можно обработать значение как именованную величину, а его адрес как производную величину. Теперь посмотрите на стратегию указателя, который является важным для философии программного управления памятью C++ (см. замечание "Указатели и философия C++").

УКАЗАТЕЛИ И ФИЛОСОФИЯ С++

Объектно-ориентированное программирование отличается от традиционного процедурного программирования принятием решений во время выполнения, а не во время компиляции. "Во время выполнения" — значит в то время, как программа работает, а "во время компиляции" — это когда компилятор собирает программу. "Принятие решения во время выполнения" — это как, например, находясь на каникулах, выбирать, какую достопримечательность посмотреть в зависимости от погоды и вашего настроения в настоящее время, тогда как принятие решения во время компиляции — скорее, как соблюдение предварительно установленного графика независимо от условий.

Принятие решения во время выполнения обеспечивает гибкость, что позволяет принародливаться к текущим обстоятельствам. Например, рассмотрим распределение памяти для массива. Традиционный путь состоит в том, чтобы объявить массив. Чтобы объявить массив в C++, необходимо определиться насчет специфического размера массива. Таким образом, размер массива установлен, когда программа откомпилирована; это — принятие решения во время компиляции. Возможно, вы думаете, что массива из 20 элементов будет достаточно 80% времени, но иногда программа должна будет обработать 200 элементов. На всякий случай лучше использовать массив из 200 элементов. Это приводит к тому, что ваша программа тратит память впустую большую часть времени выполнения. В ООП предпринимается попытка сделать программу более гибкой, откладывая такие решения до времени выполнения. Таким образом, после того как программа запущена, можно сообщить ей, что вам требуется только 20 элементов в одном случае или 205 — в другом случае.

Одним словом, решение о размере массива принимается во время выполнения. Чтобы сделать такой подход возможным, язык должен позволить создавать массив или его эквивалент во время выполнения программы. Метод C++, как вы скоро увидите, определяет использование ключевого слова **new**, чтобы запросить корректный объем памяти, и использование указателей, для того чтобы следить за тем, где расположена зарезервированная память.

Новая стратегия, используемая для обработки хранимых данных изменяет положение вещей, выполняя обработку адресов как именованную величину, а значение — как производную величину. Специальный тип переменной — **указатель** — содержит адрес значения. Таким образом, название указателя представляет его местоположение. Применяя операцию *****, называемую *косвенным значением* или операцией *разыменования*, получаем значение по данному адресу. (Да, это тот самый символ **"***, который используется для обозначения умножения; в

C++ из контекста понятен смысл символа — умножение это или разыменование). Предположим, например, что **manly** — это указатель. Тогда **manly** представляет собой адрес, а ***manly** — значение по данному адресу. Комбинация ***manly** становится эквивалентом обычной переменной типа **int**. В листинге 4.10 эти моменты отражены. Здесь также показано, как объявлять указатель.

Листинг 4.10 Программа pointer.cpp.

```
// pointer.cpp — наша первая
// переменная-указатель

#include <iostream>
using namespace std;
int main()
{
    int updates = 6; // объявление переменной
    int * p_updates; // объявление указателя
                      // как int

    p_updates = &updates; // присвоение адреса
                         // int указателю
    // выражение значений двумя путями
    cout << "Values: updates = " << updates;
    cout << ", *p_updates = " << *p_updates
        << "\n";

    // выражение адресов двумя путями
    cout << "Addresses: &updates = "
        << &updates;
    cout << ", p_updates = " << p_updates
        << "\n";

    // использование указателя для
    // изменения значения
    *p_updates = *p_updates + 1;
    cout << "Now updates = " << updates << "\n";
    return 0;
}
```

Результат выполнения программы:

```
Values: updates = 6, *p_updates = 6
Addresses: &updates = 0x85b0fff4, p_updates
            = 0x85b0fff4
Now updates = 7
```

Очевидно, что переменная типа **int updates** и переменная-указатель **p_updates** являются всего лишь двумя сторонами одной и той же медали. Переменная **updates** представляет значение как первичное и использует операцию **&** для получения адреса, тогда как переменная **p_updates** представляет адрес как первичный и использует операцию ***** для получения значения (рис. 4.8). Поскольку **p_updates** указывает на **updates**, переменные ***p_updates** и **updates** полностью эквивалентны. Вы можете использовать ***p_updates** точно так же, как переменную типа **int**. Как видно из программы, можно даже присваивать значения переменной ***p_updates**. При этом значение указателя изменяется на значение **updates**.

Объявление и инициализация указателей

Давайте рассмотрим процесс объявления указателей. Компьютеру необходимо отслеживать тип значения, на которое ссылается указатель. Например, адрес переменной типа **char** выглядит так же, как и адрес переменной типа **double**, но типы **char** и **double** используют различное количество байтов и различные внутренние форматы для хранения значений. Поэтому при объявлении указателя необходимо точно определять, на какой тип данных он указывает.

Например, последний пример содержит такое объявление:

```
int * p_updates;
```

Это указывает, что комбинация *** p_updates** принадлежит к типу **int**. Поскольку операция ***** выполняется по отношению к указателю, переменная **p_updates** сама должна быть указателем. Мы говорим, что **p_updates** указывает на тип **int**. Мы также говорим, что типом для **p_updates** должен быть указатель на **int** или, более четко, **int ***. Повторим: **p_updates** является указателем (адресом), а ***p_updates** — переменной типа **int**, но не указателем (рис. 4.9).

Между прочим, использование пробелов вокруг знака операции **"*"** является необязательным. По традиции, программисты, работающие на С, используют такую форму:

```
int *ptr;
```

Это подчеркивает мысль о том, что комбинация **"*ptr"** имеет значение типа **int**. Многие программисты, которые пишут на C++, с другой стороны, используют такую форму:

```
int* ptr;
```

Приведенная форма акцентирует внимание на том, что **int*** — это тип "указатель на **int**". Компилятору же совершенно безразлично, где будут размещены пробелы. Однако помните, что объявление

```
int* p1, p2;
```

создает один указатель (**p1**) и одну обычную переменную типа **int** (**p2**). Необходимо использовать **"*"** для каждого имени объявляемой переменной типа указатель.

ПОМНИТЕ

В C++ комбинация **int *** является производным типом "указатель на **int**".

Для объявления указателей на другие типы переменных используется тот же самый синтаксис:

```
double * tax_ptr; // tax_ptr указывает
                  // на тип double
char * str; // str указывает на тип char
```

Поскольку **tax_ptr** объявляется как указатель на тип **double**, компилятор знает, что значение ***tax_ptr** принадлежит к типу **double**. Другими словами, он знает, что ***tax_ptr** представляет собой величину, которая хранится в формате с плавающей точкой и занимает (в большинстве систем) восемь байтов. Переменная-указатель никогда не бывает просто указателем. Она всегда указывает на определенный тип. Так, переменная **tax_ptr** имеет тип "указатель на тип **double**" (или тип "**double ***"), а **str** — "указатель на тип **char**" (или "**char ***"). Несмотря на то что обе переменные являются указателями, они указывают на два различных типа. Подобно массивам, указатели являются производными от других типов данных.

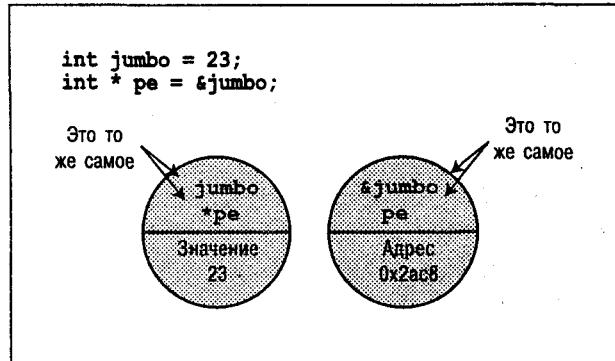


РИСУНОК 4.8 Две стороны одной медали.

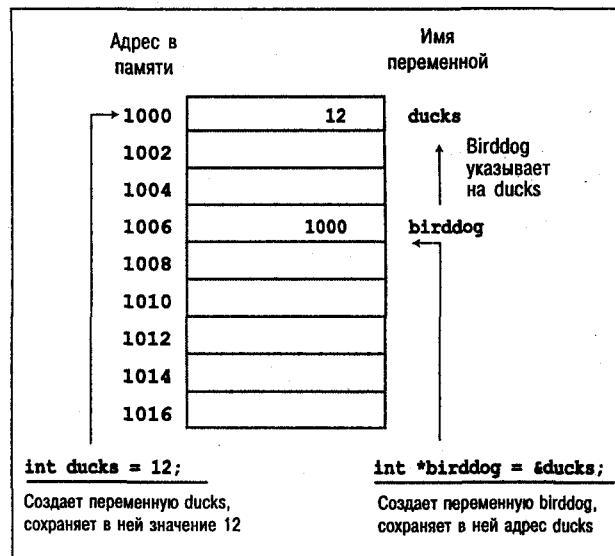


РИСУНОК 4.9 Указатели хранят адреса.

Заметьте, что, в то время как `tax_ptr` и `str` указывают на типы данных, имеющие различные размеры, две переменные, `tax_ptr` и `str`, сами по себе имеют одинаковый размер. Иными словами, адрес переменной типа `char` имеет тот же размер, что и адрес переменной типа `double`, точно так же, как 1016 может быть адресом большого склада, а 1024 — адресом маленького дома. Размер или значение адреса ничего реально не сообщает о размере или типе переменной, так же как номер дома — о здании, которое находится по этому адресу. Обычно для хранения адреса требуются два или четыре байта, в зависимости от компьютерной системы. (Некоторые системы могут иметь более длинные адреса или использовать указатели различной длины для различных типов данных.)

Вы можете использовать операторы объявления для инициализации указателя. В этом случае обычный указатель, не являющийся указателем на значение, получает начальное значение, т.е. операторы

```
int higgens = 5;
int * pi = &higgens;
```

присваивают указателю `pi` (но не `*pi`) значение `&higgens`.

Листинг 4.11 демонстрирует присваивание указателю конкретного адреса.

Листинг 4.11 Программа init_ptr.cpp.

```
// init_ptr.cpp — инициализация указателя
#include <iostream>
using namespace std;
int main()
{
    int higgens = 5;
    int * pi = &higgens;

    cout << "Value of higgens = " << higgens
        << "; Address of higgens = "
        << &higgens << "\n";
    cout << "Value of *pi = " << *pi
        << "; Value of pi = " << pi << "\n";
    return 0;
}
```

Результаты выполнения программы:

```
Value of higgens = 5; Address of higgens =
0068FDFO
Value of *pi = 5; Value of pi = 0068FDFO
```

Вы можете видеть, что программа присваивает адрес переменной `higgens` указателю `pi`, но не `*pi`.

Опасность подстерегает тех, кто неосторожно использует указатели. Одним крайне важным моментом является то, что, когда создается указатель в C++, компьютер распределяет память для хранения адреса, но не распределяет память для хранения данных, на которые указывает этот адрес. Выделение места для данных тре-

бует выполнения дополнительного действия. Игнорирование этого действия, как в нижеследующем примере, является прямой дорогой к краху:

```
long * fellow; // создание указателя-to-long
*fellow = 223323; // присвоение указателю
// неопределенного значения
```

Конечно, переменная `fellow` является указателем. Но на что она указывает? Программа не смогла присвоить адрес переменной `fellow`. Так куда будет помещено значение 223323? Мы не в состоянии ответить на этот вопрос. Поскольку переменная `fellow` не получила никакого начального значения, она может принять любое. Но какое бы значение она ни приняла, программа интерпретирует его как адрес, по которому нужно сохранить 223323. Если `fellow` вдруг получит значение 1200, то компьютер попытается поместить данные по адресу 1200, даже если окажется, что этот адрес находится в середине кода программы. Наиболее вероятно, что куда бы ни указывал указатель `fellow`, это не будет местом, куда вы хотите поместить число 223323. Ошибки подобного рода приводят к самым коварным последствиям.

ПРЕДОСТЕРЕЖЕНИЕ

Золотое правило для указателей: ВСЕГДА присваивайте указателю определенное и подходящее значение адреса до применения к нему операции разыменования (*).

Указатели и числа

Указатели не принадлежат к целочисленным типам, даже несмотря на то, что компьютер, как правило, хранит адреса как целочисленные значения. В принципе, указатели отличаются от типа целочисленных значений. Целочисленные значения являются числами, с которыми можно производить сложение, вычитание, деление и т.д. Но указатель описывает месторасположение, и поэтому, например, лишено смысла умножение адресов двух месторасположений друг на друга. В понятиях операций, которые можно производить, указатели и целочисленные значения отличаются друг от друга. Следовательно, нельзя просто присвоить целочисленное значение переменной-указателю:

```
int * pi;
pi = 0xB8000000; // несоответствие типов
```

Здесь левая часть является указателем на тип `int`, поэтому можно присвоить ей значение адреса, но правая часть является просто целочисленным значением. Вы, возможно, знаете, что `0xB8000000` представляет собой комбинированный адрес области видеопамяти вашей системы, состоящий из сегмента и смещения, но ничто в операторе не говорит программе, что данное значение является адресом. Язык C позволяет выполнять подобные присвоения. Но C++ более строго относится к со-

блюдению типов данных, и компилятор выдаст сообщение об ошибке несовпадения типов. Если вы желаете использовать числовое значение как адрес, то необходимо выполнить приведение типов для превращения числа в приемлемый для значения адреса тип:

```
int * pi;
pi = (int *) 0xB8000000; // сейчас типы совпадают
```

Теперь обе стороны оператора присваивания представляют собой целочисленные значения, поэтому присваивание возможно. Заметим, что переменная *pi* сама по себе не принадлежит к типу *int* всего лишь потому, что она является адресом значения типа *int*. Например, в большой модели памяти для IBM PC, использующей DOS, тип *int* имеет размер два байта, в то время как адреса занимают четыре байта.

Указатели имеют некоторые другие интересные свойства, которые мы обсудим тогда, когда они будут относиться к рассматриваемой теме. Между тем давайте взглянем на то, как указатели могут использоваться для управления распределением памяти во время выполнения программы.

Распределение памяти с помощью оператора new

Теперь, когда вы имеете некоторое представление о том, как работают указатели, давайте посмотрим, как они могут реализовать такую важную технологию ООП, как распределение памяти во время выполнения программы. До сих пор мы присваивали указателям адреса переменных; переменные являются *именованной* памятью, которая резервируется во время компиляции, а указатели всего лишь обеспечивают псевдонимы для областей памяти, к которым можно в любом случае обратиться напрямую по имени. Истинная ценность указателей проявляется тогда, когда для хранения данных вы распределяете *неименованную* область памяти во время выполнения программы. В этом случае указатели становятся единственным способом доступа к такой памяти. В языке C можно распределять память, используя библиотечную функцию *malloc()*. Вы можете поступить так и в C++, но в C++ существует лучший способ — оператор *new*.

Давайте попробуем использовать эту новую технологию путем создания неименованного хранилища значений типа *int* во время исполнения программы и обращаясь к этим значениям с помощью указателя. Сделаем мы это с помощью оператора языка C++ *new*. Вы указываете оператору *new*, для какого типа данных выделяется память; *new* ищет блок памяти нужного размера и возвращает адрес этого блока. Присваивая этот адрес указателю, вы получаете искомое. Ниже приводится пример реализации описанной технологии:

```
int * pn = new int;
```

Часть *new int* сообщает программе о том, что необходимо некоторое новое хранилище, подходящее для размещения переменной типа *int*. Оператор *new* использует тип для вычисления количества необходимых байтов. Затем он отыскивает область памяти и возвращает адрес. Далее присваиваем адрес переменной *pn*, которая объявляется как указатель на тип *int*. Теперь *pn* является адресом, а **pn* — значением, хранимым по этому адресу. Сравните это с присваиванием указателю адреса переменной:

```
int higgens;
int * pi = &higgens;
```

В обоих случаях (*pn* и *pi*) вы присваиваете указателю адрес переменной типа *int*. Во втором случае можно также обращаться к переменной типа *int* по имени: *higgens*. В первом же случае можно обращаться исключительно через указатель. Возникает вопрос: так как область памяти, на которую указывает *pi*, не имеет имени, как же к ней обращаться? Мы говорим, что *pi* указывает на объект данных. Но это не "объект" в смысле "объектно-ориентированного программирования"; это всего лишь "объект" в смысле "предмет". "Объект данных", под которым понимается любой блок памяти, выделенный для хранения элементарной группы данных, является более общим понятием, чем "переменная". Поэтому переменная также является объектом данных, но память, на которую указывает *pn*, не является переменной. Способ управления объектами данных с помощью указателей может показаться на первый взгляд довольно неудобным, однако он предоставляет большие возможности по управлению организацией памяти в вашей программе.

Общей формой для получения и назначения памяти для одного объекта данных, которым может быть как структура, так и основной тип, является следующая:

```
имяТипа имяУказателя = new имяТипа
```

Вы используете тип данных дважды: один раз — для указания вида запрашиваемой памяти и второй раз — для объявления подходящего указателя. Конечно, если вы уже объявили указатель нужного типа, то можете использовать его вместо объявления нового. Листинг 4.12 иллюстрирует использование оператора *new* с двумя различными типами данных.

Листинг 4.12 Программа use_new.cpp.

```
// use_new.cpp — использование оператора new
#include <iostream>
using namespace std;
int main()
{
    int * pi = new int; // выделение
                        // пространства для int
    *pi = 1001; // сохранение здесь значения
```

```

cout << "int ";
cout << "value = " << *pi
<< ": location = " << pi << "\n";
double * pd = new double; // выделение
// пространства для double
*pd = 10000001.0; // присваивание
// значения типа double
cout << "double ";
cout << "value = " << *pd
<< ": location = " << pd << "\n";
cout << "size of pi = " << sizeof pi;
cout << ": size of *pi = "
<< sizeof *pi << "\n";
cout << "size of pd = " << sizeof pd;
cout << ": size of *pd = "
<< sizeof *pd << "\n";
return 0;
}

```

укаляет на действительные данные, так что он часто используется для указания на сбой в работе оператора или функции, в случае же успеха возвращается "правильное" значение указателя. После того как вы познакомитесь с работой оператора `if` (главе 6), вы сможете проверить, когда оператор `new` возвращает нулевой указатель, и таким образом защитить свою программу от выхода за границы. В дополнение к возвращению нулевого указателя в случае неудачи при распределении памяти оператор `new` может вызвать исключительную ситуацию `bad_alloc`. Механизм исключительных ситуаций обсуждается в главе 14.

Освобождение памяти с помощью оператора `delete`

Использование оператора `new` для запроса памяти тогда, когда вам это необходимо, является наиболее эффективной стороной механизма управления памятью в C++. Другой стороной является оператор `delete`, который позволяет вам возвратить память в пул памяти после того, как вы закончите ее использовать. Это является важным шагом к наиболее эффективному использованию памяти. Память, которую вы возвращаете или освобождаете, может быть повторно использована другими частями программы. При использовании оператора `delete` за ним должен следовать указатель на блок памяти, изначально зарезервированный с помощью оператора `new`:

```

int * ps = new int; // выделение памяти
// с помощью new
...
...           // использование памяти
delete ps;   // очистка памяти с
// помощью delete по
// окончании выполнения
// программы

```

В этом примере очищается область памяти, на которую указывает `ps`, но сам указатель `ps` не удаляется. Вы можете снова использовать его, например, для указания на другую распределенную память. Необходимо всегда уравновешивать использование оператора `new` и оператора `delete`, иначе вы можете столкнуться с *утечкой памяти*, т.е. наступит момент, когда распределенную память нельзя будет больше использовать. Если утечка памяти чересчур велика, она может привести к поиску программой большего объема памяти для останова.

Не следует пытаться повторно очистить блок памяти, который уже освободили. Результат такого действия не определен. Невозможно также использовать оператор `delete` для освобождения памяти, созданной объявленными переменными:

```

int * ps = new int;      // хорошо
delete ps;               // хорошо
delete ps;               // не очень хорошо
int jugs = 5;            // хорошо
int * pi = & jugs;        // хорошо
delete pi;               // не разрешено, память
// не распределена оператором new

```

НЕХВАТКА ПАМЯТИ?

Возможно, что компьютер не имеет достаточно памяти для выполнения запроса оператора `new`. Когда это случается, оператор `new` возвращает значение 0. В языке C++ указатель со значением 0 называется нулевым указателем. C++ гарантирует, что такой нулевой указатель никогда не

ПРЕДОСТЕРЖЕНИЕ

Используйте оператор **delete** только для очистки памяти, распределенной с помощью оператора **new**. Однако вполне безопасно применять оператор **delete** к нулевому указателю.

Заметьте, что решающим моментом при применении оператора **delete** является использование его по отношению к памяти, распределенной с помощью оператора **new**. Это не значит, что нужно использовать тот же самый указатель, что использовался с оператором **new**; вместо этого нужно использовать тот же самый адрес:

```
int * ps = new int; //распределение памяти
int * pq = ps;    //второму указателю
                  //присваивается адрес
                  //того же самого блока
delete pq; //оператор delete со вторым указателем
```

Обычно не требуется создавать два указателя на один и тот же блок памяти, поскольку возрастает вероятность того, что вы ошибочно попытаетесь очистить один и тот же блок памяти дважды. Но, как вы скоро увидите, использование второго указателя имеет смысл, когда вы работаете с функцией, возвращающей указатель.

Использование оператора **new** для создания динамических массивов

Если все, что нужно программе, — это одно значение, то можно также объявить простую переменную, что проще, чем использование оператора **new** и указателя для управления одним маленьким объектом данных. Более распространенным является использование оператора **new** с большими порциями данных, например, с массивами, строками и структурами. Здесь от оператора **new** больше пользы. Предположим, например, что создается программа, для которой массив может либо потребоваться, либо нет, в зависимости от информации, получаемой в ходе выполнения программы. Если создается массив путем объявления, часть памяти будет отведена под него уже при компиляции программы. Независимо от того, использует ли программа созданный массив или нет, память он занимать будет в любом случае. Распределение массива в процессе компиляции называется *статическим связыванием*, что означает, что массив встроен в программу во время компиляции. При необходимости можно создавать массив с помощью оператора **new** во время выполнения программы, а если нет такой необходимости, то создавать его не нужно. Можно также выбрать размер массива в процессе выполнения программы. Это называется *динамическим связыванием*, что означает, что массив будет создан в ходе выполнения программы. Такой массив называется *динамическим*. При статическом связывании необходимо определить размер массива во время создания программы. При динамичес-

ком связывании программа сама принимает решение о размере создаваемого массива в ходе своего выполнения.

Теперь рассмотрим два базовых понятия, касающихся динамических массивов: как использовать оператор языка C++ **new** для создания массива и как использовать указатель для доступа к элементам массива.

Создание динамического массива с помощью оператора **new**

Создать динамический массив в C++ просто. Необходимо указать оператору **new** тип элементов массива и их желаемое количество. Синтаксис требует, чтобы было указано имя типа, а за ним — количество элементов в квадратных скобках. Например, если нужен массив из десяти переменных типа **int**, выполните следующее:

```
int * psome = new int [10]; // получение
                           // блока из 10 значений типа int
```

Оператор **new** возвращает адрес первого элемента выделенного блока. В приведенном примере это значение присваивается указателю **psome**. Необходимо согласовать количество вызовов оператора **new** с количеством вызовов оператора **delete**, чтобы тогда, когда программа закончит использование этого блока памяти, она его освободила.

Когда для создания массива используется оператор **new**, следует употребить другую форму оператора **delete**, которая указывает, что освобождается именно массив:

```
delete [] psome; // освобождается
                  // динамический массив
```

Квадратные скобки свидетельствуют о том, что необходимо освободить целый массив, а не только элемент, на который указывает указатель. Заметьте, что квадратные скобки располагаются между оператором **delete** и именем указателя. Если оператор **new** используется без квадратных скобок, то и оператор **delete** следует использовать без квадратных скобок. Такая же зависимость устанавливается и при использовании оператора **new** с квадратными скобками. Ранние версии языка C++ могут не распознавать записи с квадратными скобками. Однако для стандарта ANSI/ISO несовпадение форм операторов **new** и **delete** не определяется, это значит, что вы не можете полагаться на определенную реакцию компиляторов указанного стандарта.

```
int * pi = new int;
short * ps = new short [500];
delete [] pi; // результат не определен,
              // не делайте так
delete ps;   // результат не определен,
              // не делайте так
```

Коротко перечислим правила, которые необходимо соблюдать при использовании операторов **new** и **delete**:

- Не используйте оператор **delete** для освобождения памяти, которую не распределяли с помощью оператора **new**.
- Не используйте оператор **delete** для освобождения одного и того же блока памяти дважды подряд.
- Используйте оператор **delete []**, если был применен оператор **new []** для распределения массива.
- Используйте оператор **delete** (без квадратных скобок), если был применен оператор **new** для распределения одного элемента.
- Безопасно применять оператор **delete** к нулевому указателю (ничего не происходит).

Теперь давайте вернемся к динамическому массиву. Заметьте, что **psome** является указателем на один элемент типа **int**, первый элемент блока. А ваша забота — следить за количеством элементов блока. Поэтому, если компилятор не отслеживает факт того, что **psome** указывает на первые десять целых элементов, необходимо так писать свои программы, чтобы самим вести учет количества элементов.

Обычно программа не следит за объемом распределаемой памяти, поэтому важно правильно освобождать ее в дальнейшем — путем использования оператора **delete []**. Но эта информация не является общедоступной — вы не можете применить оператор **sizeof**, например, для определения количества байтов в динамически распределенном массиве.

Общая форма распределения и выделения памяти для массива такова:

```
type_name pointer_name = new type_name  
[num_elements];
```

В результате вызова оператора **new** закрепляется блок памяти, достаточный для хранения элементов **num_elements** типа **type_name**, а указатель **pointer_name** устанавливается на первый элемент блока. Как видно, во многих случаях можно использовать **pointer_name** как имя массива.

Использование динамического массива

Как используется динамический массив после его создания? Сначала рассмотрим эту проблему на понятийном уровне. Оператор

```
int * psome = new int [10]; // получаем  
// группу из 10 целых чисел
```

создает указатель **psome**, который указывает на первый элемент группы из десяти переменных типа **int**. Представьте его в виде пальца, который направлен на этот элемент. Предположим, **int** занимает четыре байта.

Следовательно, передвинув палец на четыре байта в правильном направлении, вы сможете указать на второй элемент. В целом существует десять элементов, которые составляют диапазон, в пределах которого можно передвигать палец. Таким образом, оператор **new** предоставляет всю информацию, которая требуется для идентификации каждого элемента группы.

Теперь подумаем об этой проблеме с практической точки зрения. Как вы обращаетесь к какому-либо из этих элементов? С первым элементом нет никаких проблем. Поскольку **psome** указывает на первый элемент массива, ***psome** является значением первого элемента. Остается еще девять элементов, к которым надо получить доступ. Самый простой способ может удивить вас, если вы никогда не работали с языком С: просто используйте указатель, как будто он является именем массива. Иными словами, для первого элемента можно использовать **psome[0]** вместо ***psome**, для второго — **psome[1]** и т.д. Оказывается, можно очень просто использовать указатель для обращения к динамическому массиву, даже если вам не сразу понятно, как этот метод работает. Можно этим пользоваться, потому что языки С и C++ обрабатывают массивы внутренним образом, в любом случае используя указатели. Такое близкое подобие массивов и указателей является одним из преимуществ языков С и C++. Сейчас мы в деталях рассмотрим это подобие. Для начала взгляните на листинг 4.13, который демонстрирует, как можно использовать оператор **new** для создания динамического массива, а затем использовать запись массива для доступа к его элементам. Этот листинг также показывает фундаментальное различие между указателем и настоящим именем массива.

Листинг 4.13 Программа arraynew.cpp.

```
// arraynew.cpp — использование оператора new  
// для массивов  
#include <iostream>  
using namespace std;  
int main()  
{  
    double * p3 = new double [3]; //пространство  
    //для трех элементов типа double  
    p3[0] = 0.2; //p3 трактуется как имя массива  
    p3[1] = 0.5;  
    p3[2] = 0.8;  
    cout << "p3[1] is " << p3[1] << ".\n";  
    p3 = p3 + 1; // приращение указателя  
    cout << "Now p3[0] is " << p3[0] << " and "  
    cout << "p3[1] is " << p3[1] << ".\n";  
    p3 = p3 - 1; //указывает на начало  
    delete [] p3; //освобождение памяти  
    return 0;  
}
```

Результаты выполнения программы из листинга 4.13:

```
p3[1] is 0.5.  
Now p3[1] is 0.5 and p3[1] is 0.8.
```

Как видно, `arraynew.cpp` использует указатель `p3`, как будто он является именем массива, первый элемент которого — `p3[0]` и т.д. Принципиальная разница между именем массива и указателем проявляется в следующей строке:

```
p3 = p3 + 1; // правильно для указателей,  
// неправильно для имен массивов
```

Вы не можете изменить значение имени массива. Однако указатель является переменной, следовательно, его значение можно изменять. Обратите внимание на эффект прибавления единицы к `p3`. Теперь выражение `p3[0]` ссылается на бывший второй элемент массива. Таким образом, после прибавления единицы к `p3` он указывает на второй элемент вместо первого. При вычитании единицы указатель возвращается к его первоначальному значению, в результате чего программа может выполнить `delete []` с правильным адресом.

Настоящие адреса последовательных чисел типа `int` обычно отличаются на 2-4 байта, поэтому тот факт, что при прибавлении единицы к `p3` получается адрес следующего элемента, свидетельствует о том, что существует какая-то особенность в арифметике указателей. Рассмотрим эту особенность.

Листинг 4.14 Программа addpntrs.cpp.

```
#include <iostream>  
using namespace std;  
int main()  
{  
    double wages[3] = {10000.0, 20000.0, 30000.0};  
    short stacks[3] = {3, 2, 1};  
  
    // Два пути для получения адреса массива  
    double * pw = wages;           // имя массива = адрес  
    short * ps = &stacks[0];        // или используется оператор адреса  
                                // вместе с элементом массива  
    cout << "pw = " << pw << ", *pw = " << *pw << "\n";  
    pw = pw + 1;  
    cout << "add 1 to the pw pointer:\n";  
    cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";  
  
    cout << "ps = " << ps << ", *ps = " << *ps << "\n";  
    ps = ps + 1;  
    cout << "add 1 to the ps pointer:\n";  
    cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";  
  
    cout << "access two elements with array notation\n";  
    cout << stacks[0] << " " << stacks[1] << "\n";  
    cout << "access two elements with pointer notation\n";  
    cout << *stacks << " " << *(stacks + 1) << "\n";  
  
    cout << sizeof wages << " = size of wages array\n";  
    cout << sizeof pw << " = size of pw pointer\n";  
    return 0;  
}
```

Указатели, массивы и арифметика указателей

Близкое подобие указателей и имен массивов проистекает из *арифметики указателей*, а также из того, что C++ обрабатывает массивы внутренне. Для начала давайте проверим арифметику. При прибавлении единицы к целой переменной ее значение увеличивается на единицу. Однако, прибавив единицу к переменной-указателю, мы увеличим ее значение на число байтов того типа, на который она указывает. Если прибавить единицу к указателю на `double`, то числовое значение в системах с 8-байтовыми типами `double` увеличится на 8. А если прибавить единицу к указателю на `short`, то значение указателя увеличится на 2 при условии, что `short` составляет 2 байта. Листинг 4.14 демонстрирует этот удивительный факт. Этот листинг также иллюстрирует и второй важный момент: в языке C++ имя массива интерпретируется как адрес.

Результаты выполнения программы из листинга 4.14:

```
pw = 0068FDE0, *pw = 10000  
прибавляем 1 к указателю pw:  
pw = 0068FDE8, *pw = 20000  
ps = 0068FDD0, *ps = 3  
add 1 to the ps pointer:  
ps = 0068FDD2, *ps = 2
```

```

access two elements with array notation
3 2
access two elements with pointer notation
3 2
24 = size of wages array
4 = size of pw pointer

```

Примечания к программе

В большинстве контекстов C++ интерпретирует имя массива как адрес его первого элемента. Так, оператор

```
double * pw = wages;
```

делает `pw` указателем на тип `double` и затем инициализирует `pw` значением `wages`, что является адресом первого элемента массива `wages`. Для `wages`, равно как и для любого другого массива, можно записать следующее равенство:

```
wages = &wages[0] = адрес первого элемента
        массива.
```

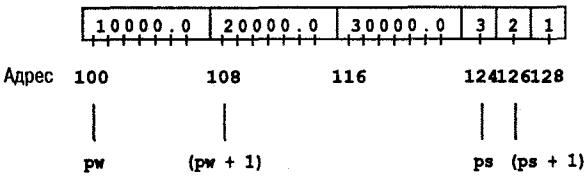
Чтобы показать, что это не пустые слова, в программе явно используется адресный оператор в выражении `&stacks[0]`, чтобы присвоить указателю `ps` начальное значение, равное первому элементу целого массива.

Далее программа проверяет значения указателей `pw` и `*pw`. Первый является адресом, а второй — значением, находящимся по этому адресу. Поскольку `pw` указывает на первый элемент, значение `*pw` отображается как значение первого элемента, т.е. 10000. Затем программа прибавляет 1 к `pw`. Как и было обещано, при этом к числовому значению адреса прибавляется 8 ($E0 + 8 = E8$ в шестнадцатиричной системе счисления), так как `double` в нашей системе занимает 8 байтов. Благодаря этому `pw` становится равным адресу второго элемента. Так, `*pw` теперь равно 20000 — значению второго элемента (рис. 4.10). (Значения адресов на рисунке подобраны для того, чтобы сделать его нагляднее.)

```

double wages[3] = {10000.0, 20000.0, 30000.0};
short stacks[3] = {3, 2, 1};
double * pw = wages;
short * ps = &stacks[0];

```



`pw` указывает на тип `double`, поэтому при прибавлении 1 к `pw` его значение изменяется на 8 байтов.

`ps` указывает на тип `short`, поэтому при прибавлении 1 к `ps` его значение изменяется на 2 байта.

После этого программа выполняет подобные действия для указателя `ps`. В этот раз из-за того, что `ps` указывает на тип `short`, а `short` занимает 2 байта, при прибавлении 1 к указателю его значение увеличивается на 2. И снова указатель в результате указывает на следующий элемент массива.

ПОМНИТЕ

Запомните, если прибавить 1 к переменной-указателю, его значение увеличивается на число байтов того типа, на который он указывает.

Теперь рассмотрим выражение `stacks[1]`. Компилятор C++ обрабатывает это выражение так, как будто вы написали его как `*(stacks + 1)`. Второе выражение означает вычисление адреса второго элемента массива, а затем поиск значения, которое хранится по этому адресу. В результате получается точно значение элемента `stacks[1]`. (Приоритет операторов требует, чтобы использовались круглые скобки. Без них единица будет прибавлена к `*stacks` вместо `stacks`.)

Результат выполнения программы показывает, что `*(stacks + 1)` и `stacks[1]` — это одно и то же. Аналогично `*(stacks + 2)` — то же самое, что и `stacks[2]`. В общем, когда бы вы ни осуществляли написание имени массива, C++ делает следующее преобразование:

```
arrayname[i] becomes *(arrayname + i)
```

И если вы используете указатель вместо имени массива, C++ выполняет такое же преобразование:

```
pointername[i] becomes *(pointername + i)
```

Таким образом, во многих случаях можно использовать имена указателей и имена массивов одинаково. Вы можете использовать скобки в выражениях для массивов с любым из них. Можно применять операцию разыменования (*) к любому из них. В большинстве выражений каждый из них представляет собой адрес. Единственное отличие состоит в том, что можно изменять значение указателя, в то время как имя массива является константой:

```
pointername = pointername + 1; // допускается
arrayname = arrayname + 1; // не допускается
```

Второе отличие состоит в том, что, применяя оператор `sizeof` к имени массива, мы получаем размер массива, а применяя `sizeof` к указателю, получаем размер указателя, даже если он указывает на массив. Например, в листинге 4.15 оба указателя, `pw` и `wages`, указывают на один и тот же массив. Но применение к ним оператора `sizeof` дает следующие результаты:

```

24 = size of wages array // отображение
      // sizeof wages
4 = size of pw pointer // отображение
      // sizeof pw

```

РИСУНОК 4.10 Сложение указателей.

Это единственный случай, когда C++ не интерпретирует имя массива как адрес.

Одним словом, очень просто использовать оператор `new` для создания массива, а указатель для обращения к различным элементам. Просто рассматривайте указатель как имя массива. Однако понять, почему это именно так происходит, очень сложно. Если вы действительно хотите разобраться с массивами и указателями, необходимо тщательно рассмотреть их взаимосвязь. Вы уже получили довольно много информации об указателях, поэтому давайте подытожим, что вы знаете об указателях и массивах.

Основные сведения об указателях

Объявление указателей. Чтобы объявить указатель на какой-либо тип, используйте следующую форму:

```
typeName * pointerName;
```

Примеры:

```
double * pn; //pn указывает на значение double
char * pc; //pc указывает на значение char
```

Здесь `pn` и `pc` — указатели, а `double *` и `char *` — записи языка C++ для типов "указатель на `double`" и "указатель на `char`".

Присвоение значений указателям. Следует присвоить указателю адрес в памяти. Вы можете применять операцию `&` к имени переменной, чтобы получить адрес именованной записи, при этом оператор `new` возвращает адрес безымянной записи.

Примеры:

```
double bubble = 3.2;
pn = &bubble; //присваивает указателю pn
              //адрес bubble
pc = new char; //присваивает указателю pc
              //адрес только что зарезерви-
              //рованной записи char
```

Разыменование указателей. Разыменование указателей означает обращение к значению, на которое указывает указатель. Чтобы разыменовать указатель, примените к нему операцию разыменования (`*`). Так, если `pn` — это указатель на `bubble`, как в предыдущем примере, то `*pn` — это значение, на которое указывает `pn`, т.е. в этом случае 3.2.

Примеры:

```
cout << *pn; //печатает значение bubble
*pc = 'S'; //помещает 'S' в ячейку памяти,
            //адресом которой является pc
```

Никогда не разыменовывайте указатель, которому не был присвоен соответствующий адрес.

Различие между указателем и значением, на которое он указывает. Запомните, что если `pi` — это указатель на `int`, то `*pi` — это не указатель на `int`, а полный эквива-

лент переменной типа `int`. Именно `pi` является указателем.

Примеры:

```
int * pi = new int; // присваивает адрес
                    // указателю pi
*pi = 5; //сохраняет значение 5 по этому адресу
```

Имена массивов. В большинстве контекстов C++ интерпретирует имя массива как эквивалент адреса первого элемента массива.

Пример:

```
int tacos[10]; // теперь tacos — то же
                // самое, что и &tacos[0]
```

Единственное исключение — это когда вы используете имя массива вместе с оператором `sizeof`. В таком случае `sizeof` возвращает размер целого массива в байтах.

Арифметика указателей. C++ позволяет прибавлять целые числа к указателям. В результате прибавление единицы равносильно прибавлению значения, равного количеству байтов в объекте, на который указывает указатель, к исходному значению адреса. Еще можно вычитать целые числа из указателей, а также один указатель из другого. Последняя операция, результатом которой является целое число, является значимой, только если оба указателя указывают на один и тот же массив (указание на какую-либо позицию за пределами массива также возможно). В результате выполнения этой операции возвращается разница между двумя элементами.

Примеры:

```
int tacos[10] = {5,2,8,4,1,2,2,4,6,8};
int * pt = tacos; // предположим, что pt и
                  // fog — это адрес 3000
pt = pt + 1; // теперь pt равно 3004, если
              // int занимает четыре байта
int * pe = &tacos[9]; // pe равно 3096,
                     // если int занимает четыре байта
pe = pe - 1; // теперь pe равно 3032,
              // адрес facos[8]
int diff = pe - pt; // diff равен 7,
                     // разница между facos[8] и facos[1]
```

Динамическое и статическое связывание для массивов. Используйте объявление массива для создания массива со статическим связыванием, т.е. массива, размер которого устанавливается во время компиляции:

```
int tacos[10]; // статическое связывание,
                // размер устанавливается во время компиляции
```

Используйте оператор `new []` для создания массива с динамическим связыванием (динамический массив), т.е. массива, для которого зарезервировано место, а его размер может быть установлен во время выполнения программы. Освободите память с помощью оператора `delete []` после того, как размер будет установлен:

```

int size;
cin >> size;
int * pz = new int [size]; // динамическое
                           // связывание
...
delete [] pz; // освобождение памяти
               // по завершению

```

Запись массива и запись указателя. Использование скобок в записи массива эквивалентно разыменованию указателя:

```

tacos[0] равносильно *tacos и равносильно
значению по адресу tacos
tacos[3] равносильно *(tacos + 3) и
равносильно значению по адресу tacos + 3

```

Это справедливо и для имен массивов, и для переменных-указателей, поэтому можно использовать как запись в виде указателя, так и запись в виде массива с указателями и именами массива.

Примеры:

```

int * pi = new int [10]; // pi указывает
                        // на группу из 10 целых чисел
*pi = 5; // присваивает нулевому элементу
          // значение 5
pi[0] = 6; // переустанавливает элемент
            // zero в 6
pi[9] = 44; // устанавливает элемент tenth
             // в 44
int tacos[10];
*(tacos + 4) = 12; // устанавливает
                    // tacos[4] в 12

```

Указатели и строки

Особая связь между массивами и указателями распространяется также и на строки. Рассмотрим следующий код:

```

char flower[10] = "rose";
cout << flower << "s are red\n";

```

Имя массива — это адрес его первого элемента, поэтому `flower` в операторе `cout` — это адрес элемента `char`, который содержит символ `r`. Объект `cout` предполагает, что адрес элемента `char` — это адрес строки, поэтому он печатает символ по этому адресу и затем продолжает печать символов, пока не достигнет нулевого символа (`\0`). Одним словом, если вы присвойте оператору `cout` адрес символа, он напечатает все, начиная с этого символа и заканчивая первым нулевым символом, который встретится за ним.

Решающим здесь является не то, что `flower` — это имя массива, а то, что `flower` выступает адресом элемента `char`. Это подразумевает, что можно также использовать переменную-указатель на `char` в качестве аргумента оператора `cout`, так как он тоже является адресом элемента `char`. Безусловно, такой указатель должен указывать на начало строки, и скоро мы покажем это.

Но сначала давайте разберемся с конечной частью оператора, который идет перед `count`. Если `flower` на самом деле является адресом первого символа строки, что же значит выражение "`s are red\n`"? Для согласования с тем, как оператор `cout` обрабатывает результат строки, эта строка в кавычках тоже должна быть адресом. И это так и есть, поскольку в C++ строка в кавычках так же, как и имя массива, является адресом своего первого элемента. Предыдущий код на самом деле не передает в `cout` целую строку, он просто передает адрес строки. Это значит, что строки в массиве, строковые константы в кавычках и строки, описанные указателями, все обрабатываются эквивалентно. На самом деле каждый из них передается как адрес. При этом, безусловно, выполняется меньше работы, чем если бы обрабатывался каждый символ в строке.

ПОМНИТЕ

В операторе `cout`, как и в большинстве выражений в C++, имя массива элементов `char`, указатель на `char` и строковая константа в кавычках — все интерпретируются как адрес первого символа строки.

Листинг 4.15 иллюстрирует использование различных видов строк. В нем используются две функции из библиотеки строк. Функция `strlen()`, которую мы уже использовали ранее, возвращает длину строки. Функция `strcpy()` копирует строку с одного места в другое. У обеих функций есть прототипы в заголовочном файле `cstring` (или `string.h` в более старых трактовках). В программе также приведены некоторые случаи неправильного употребления указателей.

ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если в вашей системе нет заголовочного файла `cstring`, используйте более старую версию — `string.h`.

Ниже приведен пример выполнения программы из листинга 4.15:

```

bear and wren
" output from cout << ps << "\n";
Enter a kind of animal: fox
foxs!
Before using strcpy():
fox at 0068FDE4
fox at 0068FDE4
After using strcpy():
fox at 0068FDE4
fox at 007B0D80

```

Примечания к программе

Программа, представленная в листинге 4.15, создает один массив типа `char` (`animal`) и две переменные-указатели на `char` (`bird` и `ps`). Программа начинается с присваивания массиву `animal` строки `"bear"`, точно так же, как мы ранее присваивали значения массивам.

Листинг 4.15 Программа ptrstr.cpp.

```
// ptrstr.cpp - использование указателей на строки
#include <iostream>
using namespace std;
#include <cstring> // объявление strlen(), strcpy()
int main()
{
    char animal[20] = "bear"; // animal хранит bear
    const char * bird = "wren"; // bird хранит адрес строки
    char * ps; // не инициализировано

    cout << animal << " and "; // отображение bear
    cout << bird << "\n"; // отображение wren
    cout << ps << "\n"; // blunder - отображение "мусора"

    cout << "Enter a kind of animal: ";
    cin >> animal; // хорошо, если ввод содержит < 20 символов
    // cin >> ps; слишком ужасно пытаться использовать blunder; ps не может указывать на
    // распределенное пространство

    ps = animal; // установка ps в качестве указателя на строку данных
    cout << ps << "!\n"; // хорошо, то же, что и использование animal
    cout << "Before using strcpy():\n";
    cout << animal << " at " << (int *) animal << endl;
    cout << ps << " at " << (int *) ps << endl;

    ps = new char[strlen(animal) + 1]; // получение новой памяти
    strcpy(ps, animal); // копирование строки в новую память
    cout << "After using strcpy():\n";
    cout << animal << " at " << (int *) animal << endl;
    cout << ps << " at " << (int *) ps << endl;
    delete [] ps;
    return 0;
}
```

Затем программа делает кое-что новое. Она присваивает строку указателю на **char**:

```
const char * bird = "wren"; // bird
                           // содержит адрес строки
```

Помните, на самом деле "wren" представляет собой адрес строки, поэтому этот оператор назначает адрес "wren" указателю **bird**. (Как правило, компилятор оставляет без внимания область в памяти, предназначенную для хранения всех строк, используемых в исходном коде программы; вместо этого он связывает каждую сохраняемую строку с ее адресом.) Это значит, что можно использовать указатель **bird** так, как вы использовали бы строку "wren", как, например, в операторе **cout**:

```
cout << "A concerned " << bird
     << " speaks\n"
```

Строковые литералы являются константами, вот почему в коде при объявлении используется ключевое слово **const**. Употребление **const** таким способом означает, что можно использовать **bird** для обращения к строке, однако не можете ее таким образом изменять. В главе 7 тема указателей на тип **const** рассматривается более де-

тально. В результате указателю **ps** не было присвоено никакого адреса, поэтому он не указывает ни на одну строку. (Как вы помните, обычно рекомендуется присваивать указателю адрес, а данный пример демонстрирует несоблюдение этого правила).

Далее, из программы видно, что в операторе **cout** можно использовать имя массива **animal** и указатель **bird** равнозначно. В конце концов, они оба являются адресами строк, и **cout** отображает две строки ("bear" и "wren"), которые хранятся по этим адресам. Когда в коде допускается ошибка, которая заключена в попытке отобразить **ps**, мы получаем пустую строчку. Создание указателя, которому не присвоено никакого адреса, похоже на выдачу пустого чека с подписью; вы не можете про-контролировать, как он будет использован. В данном примере нам немного повезло, так как указатель **ps**, не имея никакого значения, мог бы случайно указать на какой-нибудь неудобный адрес. Но в нашем при-мере он указывает на адрес, по которому хранится нуль, поэтому ничего и не отобразилось. В противном случае при выводе мог бы появиться какой-нибудь "мусор".

При вводе ситуация немного иная. Вы можете использовать массив **animal** для ввода до тех пор, пока выводимые строки достаточно короткие, чтобы уместиться в массиве. Однако было бы неправильным использовать для ввода указатель **bird**:

- Некоторые компиляторы обрабатывают строковые литералы как константы "только чтение", в результате чего возникает ошибка времени выполнения, если вы пытаетесь записать вместо них новые данные. Сейчас в языке C++ строчные литералы являются константами, однако некоторые компиляторы все еще работают по-старому.
- Некоторые компиляторы просто используют одну копию каждого литерала, содержащегося в строке, для представления каждый раз, как эта буква встречается в программе.

Позвольте нам подробнее остановиться на втором из положений. В C++ нет гарантий, что литералы, содержащиеся в строке, хранятся однозначно. Другими словами, если вы используете какой-либо литерал строки "**wren**" несколько раз, компилятор мог бы сохранять несколько копий строки или только одну копию. В последнем случае присваивание строки **bird** указателю на одну из строк "**wren**" приводит к тому, что он указывает на единственную копию этой строки. Считывание значения в одну строку могло бы воздействовать на объект так, что в любом другом месте вы бы приняли его за независимую строку. В любом случае, поскольку указатель **bird** объявлен как **const**, компилятор предотвращает любые попытки изменить содержимое того адреса, на который указывает **bird**.

Еще хуже — пытаться считывать информацию, хранимую по адресу, на который указывает **ps**. Поскольку указателю **ps** не присвоено никакого адреса, вы не знаете, откуда будет взята информация. Она может быть даже перезаписана в памяти. К счастью, очень легко избежать этих проблем — просто используйте достаточно большой массив типа **char** для вводимых данных. Не пользуйтесь для получения данных строковыми константами или указателями, которым не присвоен никакой адрес.

ПРЕДОСТЕРЖЕНИЕ

Когда вы считываете строку в программу, нужно всегда использовать адрес, предварительно зарезервированный в памяти. Такой адрес может быть в виде имени массива или указателя, которому было присвоено значение с помощью оператора **new**.

Теперь посмотрите, что выполняет следующий код:

```
ps = animal; // указатель ps указывает на строку
....
```

```
cout << animal << " at " << (int *) animal
<< endl;
cout << ps << " at " << (int *) ps
<< endl;
```

Он дает следующий результат:

```
fox at 0068FDE4
fox at 0068FDE4
```

Обычно, если вы используете в операторе **cout** указатель, он выводит адрес. Но если тип указателя — **char ***, оператор **cout** отображает строку, на которую он указывает. Если вы хотите увидеть адрес строки, следует использовать приведение типа указателя к указателю другого типа, такого как **int ***, что и выполняет данный код. Так что **ps** выводит строку "fox", но **(int *) ps** выводит адрес, по которому найдена эта строка. Примите к сведению, что при присвоении **animal** указателю **ps** копируется не строка, а ее адрес. В результате получаем два указателя (**animal** и **ps**) на один и тот же адрес в памяти и строку.

Чтобы получить копию строки, нужно потрудиться немного больше. Во-первых, необходимо зарезервировать память для сохранения строки. Это можно сделать, объявив второй массив или используя оператор **new**. Второй подход позволяет зарезервировать место для строки:

```
ps = new char[strlen(animal) + 1];
// получает новое место
```

Строка "fox" не заполняет массив **animal** полностью, поэтому место в памяти не используется. В этой части кода используется функция **strlen()** для определения длины строки. Затем программа использует оператор **new** для резервирования ровно такого объема памяти, которого достаточно для хранения этой строки.

Далее потребуется способ, позволяющий скопировать строку из массива **animal** в новое зарезервированное место. Вы не можете назначить массив **animal** указателю **ps**, так как это только изменит адрес, который хранится в **ps**, и, таким образом, теряется единственный способ, с помощью которого программа должна обращаться к новому зарезервированному месту памяти. Вместо этого следует использовать библиотечную функцию **strcpy()**:

```
strcpy(ps, animal);
// копирует строку в новое место
```

Функция **strcpy()** содержит два аргумента. Первый из них — это адрес назначения, второй — адрес строки, которую требуется скопировать. Это ваше дело, как определить, что по адресу назначения память действительно зарезервирована и места для хранения копии достаточно. В данном примере это было выполнено с помощью функции **strlen()** для определения правильного

размера и оператора `new` для получения свободной памяти.

Часто бывает необходимо поместить строку в массив. Когда нужно присвоить значение массиву, используйте оператор присвоения либо функцию `strcpy()` или `strncpy()`. Вы уже видели функцию `strcpy()`; она работает следующим образом:

```
char food[20] = "carrots"; // присваивание
                           // значения
strcpy(food, "flan"); // другим способом
```

Заметьте, что конструкция типа

```
strcpy(food, "a picnic basket filled with
many goodies");
```

может привести к появлению проблем, так как массив `food` меньше, чем строка. В этом случае функция копирует остаток строки в байты памяти, которые размещены сразу после массива, в результате чего могут измениться данные, которые использует ваша программа. Для избежания этой проблемы используйте функцию `strncpy()`. Для нее требуется третий аргумент: максимальное количество символов, которые требуется скопировать. Однако помните, что если выделенная для этой функции память завершится прежде, чем функция достигнет конца строки, она не прибавит нулевой символ. Так, следует использовать функцию, подобную следующей:

```
strncpy(food,
        "a picnic basket filled with many goodies",
        19);
food[19] = '\0';
```

Здесь происходит копирование до 19 символов в массив и затем последний элемент назначается нулевым символом. Если строка короче 19 символов, функция `strncpy()` добавляет нулевой символ раньше, для того чтобы отметить настоящий конец строки.

ПОМНИТЕ

Для назначения строки массиву используйте функцию `strcpy()` или `strncpy()`, а не оператор присваивания.

Использование оператора `new` для создания динамических структур

Вы увидели преимущества того, что массивы создаются во время выполнения программы, а не во время компиляции. То же самое можно сказать и о структурах. При этом нужно зарезервировать место только для такого количества структур, которое требуется программе во время конкретного исполнения. Снова используется такое средство, как оператор `new`. С его помощью можно создавать динамические структуры. Опять же, слово "динамические" означает то, что память выделяется во време-

мя выполнения программы, а не во время компиляции. Кроме того, вы можете применять технику, которая была рассмотрена в отношении структур, к классам, так как классы во многом похожи на структуры.

Использование оператора `new` при работе со структурами является "двухэтажным": создание структуры и доступ к ее элементам. Чтобы создать структуру, используйте тип структуры с оператором `new`. Например, чтобы создать безымянную структуру типа `inflatable` и присвоить ее адрес подходящему указателю, можно сделать следующее:

```
inflatable * ps = new inflatable;
```

Здесь указателю `ps` присваивается адрес части свободной памяти, достаточной для хранения структуры типа `inflatable`. Заметьте, что синтаксис здесь точно такой же, какой используется в C++ для встроенных типов данных.

Получить доступ к элементам структуры сложнее. Когда вы создаете динамическую структуру, вы не можете применить точечный оператор принадлежности к имени структуры, так как у структуры нет имени. Все, что у вас есть, — это ее адрес. В C++ есть оператор, специально предназначенный для этого случая: стрелочный оператор принадлежности (`->`). Этот оператор, который формируется из дефиса и знака "больше", действует на указатели на структуры так же, как точечный указатель на имена структур. Например, если `ps` указывает на структуру типа `inflatable`, то `ps->price` — это элемент `price` указанной структуры (рис. 4.11).

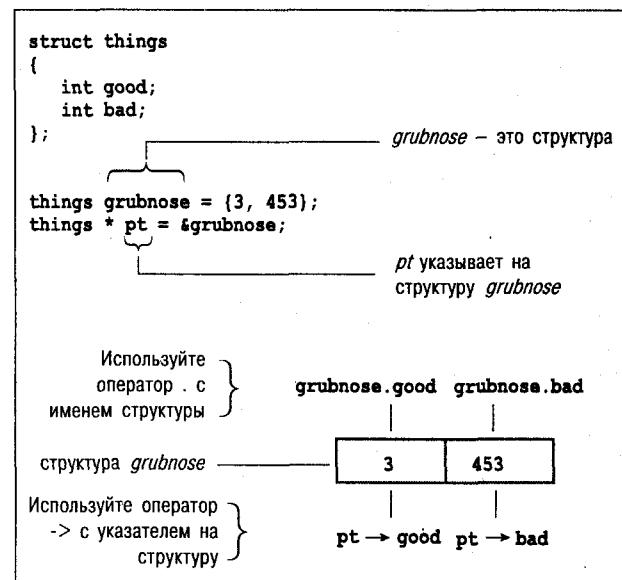


РИСУНОК 4.11 Идентифицирующий элемент структуры

ПОМНИТЕ

Иногда начинающие пользователи путают, когда следует использовать точечный оператор, а когда стрелочный для определения элемента структуры. Правило простое. Если идентификатор структуры представлен в виде имени структуры, используйте точечный оператор. А если идентификатор — это указатель на структуру, используйте стрелочный оператор.

Второй, более неприятный принцип — понимание того факта, что если `ps` — это указатель на структуру, то `*ps` представляет значение, на которое он указывает, т.е. саму структуру. Далее, так как `ps` — это структура, то `(*ps).price` — это элемент `price` структуры. Правила приоритета операторов в языке C++ требуют использования в этой конструкции круглых скобок.

В листинге 4.16 используется оператор `new` для создания безымянной структуры, а также демонстрируются оба способа записи указателей для доступа к элементам структуры.

Результаты выполнения программы из листинга 4.16:

```
Enter name of inflatable item: Fabulous Frodo
Enter volume in cubic feet: 1.4
Enter price: $17.99
Name: Fabulous Frodo
Volume: 1.4 cubic feet
Price: $17.99
```

Пример использования операторов `new` и `delete`

Давайте рассмотрим пример, в котором операторы `new` и `delete` используются для осуществления ввода с клави-

атуры строки, которая будет сохранена в памяти. В листинге 4.17 определяется функция, которая возвращает указатель на вводимую строку. Эта функция считывает вводимые данные во временный массив большого размера и затем использует оператор `new []` для создания блока памяти, размер которого точно подходит для сохранения вводимой строки. После этого функция возвращает указатель на этот блок. При таком подходе может заблокироваться большой участок памяти, если в программе происходит чтение большого количества строк.

Предположим, в вашей программе должно выполняться чтение 1000 строк и самая длинная строка содержит 79 символов, но большинство строк намного короче. Если бы для хранения строк использовались бы массивы типа `char`, вам потребовалось бы 1000 массивов, каждый по 80 символов. Это оставляет 80 тыс. байтов, и большая часть этого блока памяти в итоге не используется. В качестве альтернативного варианта можно создать массив из 1000 указателей на тип `char` и затем использовать оператор `new` для резервирования только того объема памяти, который требуется для каждой строки. Так можно было бы сэкономить десятки из тысяч байтов. Вместо того чтобы использовать большой массив для каждой строки, вы подстраиваете память к вводу. И даже еще лучше, с помощью оператора `new` можно определить объем памяти для хранения ровно такого количества указателей, которое требуется. Для нашего случая это довольно громоздко. Даже использование массива из

Листинг 4.16 Программа newstrct.cpp.

```
// newstrct.cpp - использование new со структурой
#include <iostream>
using namespace std;
struct inflatable // шаблон структуры
{
    char name[20];
    float volume;
    double price;
};
int main()
{
    inflatable * ps = new inflatable; // распределение пространства для структуры

    cout << "Enter name of inflatable item: ";
    cin.get(ps->name, 20); // метод 1 для доступа к элементу структуры
    cout << "Enter volume in cubic feet: ";
    cin >> (*ps).volume; // метод 2 для доступа к элементу структуры
    cout << "Enter price: $";
    cin >> ps->price;
    cout << "Name: " << (*ps).name << "\n"; // метод 2
    cout << "Volume: " << ps->volume << " cubic feet\n";
    cout << "Price: $" << ps->price << "\n"; // метод 1
    return 0;
}
```

Листинг 4.17 Программа delete.cpp.

```

// delete.cpp - использование оператора delete
#include <iostream>
#include <cstring>           // либо string.h
using namespace std;
char * getname(void);        // прототип функции
int main()
{
    char * name;             // создается указатель, но не память
    name = getname();         // имени присваивается адрес строки
    cout << name << " at " << (int *) name << "\n";
    delete [] name;          // очистка памяти

    name = getname();         // повторное использование очищенной памяти
    cout << name << " at " << (int *) name << "\n";
    delete [] name;          // повторная очистка памяти
    return 0;
}

char * getname()            // возврат указателя на строку new
{
    char temp[80];           // временная память

    cout << "Enter last name: ";
    cin >> temp;
    char * pn = new char[strlen(temp) + 1];
    strcpy(pn, temp);        // копировать строку с учетом меньшего объема памяти

    return pn;                // временная потеря при завершении функции
}

```

1000 указателей — это, можно сказать, грандиозное решение для данного случая. Однако в листинге 4.17 показано несколько приемов, а также пример, как работает оператор **delete**, который использован в программе для очистки памяти в целях повторного использования.

Пример выполнения программы:

```

Enter last name: Fredeldumpkin
Fredeldumpkin at 007B0B40
Enter last name: Pook
Pook at 007B0DC0

```

Примечания к программе

Во-первых, рассмотрим функцию **getname()**. Она использует **cin** для размещения вводимого слова во временный массив. Далее она использует оператор **new** в целях резервирования нового места в памяти для сохранения слова. С учетом нулевого символа программе требуется **strlen(temp) + 1** символ для сохранения строки. Это и есть то значение, которое передается оператору **new**. После того как место в памяти становится доступным, **getname()** использует стандартную библиотечную функцию **strcpy()** для копирования строки из временного массива **temp** в новый блок. Эта функция не проверяет, достаточно ли места для запоминания данной строки; это выполняет функция **getname()**, запрашивая требуемое

количество байтов с помощью оператора **new**. В результате функция возвращает указатель **pn** — адрес копии строки.

В **main()** возвращенное значение (адрес) присваивается указателю **name**. Этот указатель определен в **main()**, но указывает на блок памяти, зарезервированный в функции **getname()**. Затем программа выводит строку и ее адрес.

После того как программа очищает блок памяти, на который указывает **name**, **main()** вызывает **getname()** второй раз. В C++ нет гарантии, что только что очищенный блок памяти сразу же будет задействован при следующем использовании оператора **new**, это подтверждается в данном примере выполнения программы.

Чтобы оценить более тонкие моменты этой программы, следует узнать немного более о том, как C++ работает с памятью. Поэтому давайте немного опередим события и рассмотрим материал, который более подробно изложен в главе 8.

Автоматическая, статическая и свободная память

В C++ можно назвать три типа памяти — *автоматическая*, *статическая* и *свободная*. Данные, к которым применяется тот или иной тип памяти, отличаются друг от

друга своим временем существования. Мы кратко рассмотрим каждый тип.

Автоматические переменные

Обычные переменные, определенные внутри функции, называются *автоматическими*. Они начинают свое существование автоматически, когда вызвана функция, которая их содержит, а прекращают существование, когда заканчивает свое действие функция. Например, массив **temp** в листинге 4.17 существует только до тех пор, пока функция **getname()** активна. Когда управление в программе возвращается к **main()**, память, используемая для **temp**, очищается автоматически. Если бы **getname()** возвратила адрес **temp**, указатель **name** в **main()** указывал бы на область памяти, которая в скором времени была использована повторно. Это одна из причин, почему мы были вынуждены использовать оператор **new** в **getname()**.

На самом деле автоматические переменные локальны по отношению к блоку, в котором они содержатся. Блок — это часть кода, заключенная в фигурные скобки. До сих пор все наши блоки были полноценными функциями. Но, как вы увидите в следующей главе, могут быть блоки внутри функции. Если в одном из таких блоков будет описана переменная, она будет существовать, только пока программа выполняет операторы внутри этого блока.

Статическая память

К статической памяти относятся переменные, которые существуют во время выполнения программы в целом. Существует два способа, позволяющие сделать переменную статической. Первый — определить ее как внешнюю, вне функции. Второй — использовать ключевое слово **static** при объявлении переменной:

```
static double fee = 56.50;
```

В языке С Кернигана и Ритчи мы можем объявлять только статические массивы и структуры, в то время как в C++ версии 2.0 (и более поздних) и в ANSI C можно объявлять также автоматические массивы и структуры. Однако, как некоторые из вас могли обнаружить, в некоторых реализациях C++ все еще невозможно объявлять автоматические массивы и структуры.

В главе 8 статическая память будет рассмотрена более детально. Главный вывод, который можно сделать сейчас об автоматической и статической памяти, состоит в том, что эти типы памяти жестко определяют время существования переменной. Переменная существует либо во время выполнения всей программы (статическая переменная), либо только во время действия какой-либо из функций (автоматическая переменная).

Свободная память

В то же время операторы **new** и **delete** обеспечивают более гибкий подход. Они резервируют некоторый пул памяти, к которому C++ обращается как к свободной памяти. Этот путь отделен от памяти, используемой для статических и автоматических переменных. Как видно из листинга 4.17, операторы **new** и **delete** позволяют вам в одной функции выделять память, а в другой — очищать ее. Таким образом, время существования данных не привязано жестко ко времени работы программы или функции. Используя эти операторы вместе, можно лучше контролировать то, как программа использует память, чем с помощью обычных переменных.

ПРИМЕЧАНИЕ

Указатели являются одним из самых мощных средств языка C++. В то же время они являются самым опасным средством, так как позволяют выполнять действия, "неудобные" для компьютера. Имеется в виду, например, использование указателя, которому не присвоено значение, для доступа в память или попытка очистить один и тот же блок памяти дважды. Пока вы на практике не привыкнете к записям указателей и к самой концепции указателей, они могут вносить путаницу в вашу работу. В этой книге мы еще несколько раз вернемся к указателям. Надеемся, что с каждым разом вам будет все легче работать с ними.

Резюме

Три производных типа данных в языке C++ — это массивы, структуры и указатели. В массиве могут храниться значения одного и того же типа. С использованием индекса или субиндекса вы можете обращаться к отдельным элементам массива.

Структура может содержать несколько значений разных типов; вы можете использовать оператор членства (.) для обращения к отдельным элементам структуры. Первым шагом в использовании структур является создание шаблона структуры, который определяет, какие элементы содержит структура. Имя, или тег, этого шаблона становится впоследствии идентификатором нового типа. После этого можно объявлять структурные переменные такого типа.

Объединение может хранить только одно значение, которое, однако, может принадлежать к нескольким типам. Имя элемента показывает, какой тип используется в данный момент.

Указатели — это переменные, в которых хранятся адреса. Мы говорим, что указатель указывает на адрес, который он содержит. При объявлении указателя необходимо указывать, на объект какого типа он указывает. Результатом применения операции разыменования (*) к

указателю является значение по адресу, на который он указывает.

Строка — это серия символов, которая заканчивается нулевым символом. Страна может быть представлена строковой константой в кавычках; в этом случае нулевой символ присутствует неявно. Вы можете сохранять строку в массив типа `char`, а также представлять ее указателем на `char`, который указывает на эту строку. Функция `strlen()` возвращает длину строки без учета нулевого символа. Функция `strcpy()` копирует строку с одного положения в другое. При использовании этих функций включайте в программу заголовочный файл `cstring` или `string.h`.

Оператор `new` позволяет резервировать память для какого-либо из объектов данных во время выполнения программы. Этот оператор возвращает адрес памяти, который он получил; вы можете присвоить этот адрес указателю. Единственный способ получения доступа к этому блоку памяти — использование указателя. Если объект данных — это простая переменная, можно использовать операцию разыменования (*) для получения значения, хранимого по этому адресу. Если объект данных — это массив, для доступа к его элементам можно использовать указатель так, будто он является именем массива. Если объект данных — это структура, для доступа к элементам структуры можно использовать операцию разыменования указателей (->).

Указатели и массивы тесно связаны. Если, например, `ar` — это имя массива, то выражение `ar[i]` интерпретируется как `(ar + i)`, а имя массива интерпретируется как адрес первого элемента массива. Таким образом, имя массива играет ту же роль, что и указатель. В свою очередь, можно использовать имя указателя с записью массива для обращения к элементам массива, назначенным с помощью оператора `new`.

Операторы `new` и `delete` дают возможность контролировать назначение объектов данных и их возвращение в пул памяти. Автоматические переменные, т.е. переменные, которые объявлены внутри функции, и статические переменные, определенные за пределами функции или с помощью ключевого слова `static`, являются менее гибкими. Автоматические переменные появляются, когда управление осуществляется в содержащем их блоке (обычно это объявление функции), и исчезают, когда управление передается за пределы блока. Статические переменные существуют в течение всего времени действия программы.

Вопросы для повторения

- Как бы вы объявили следующие объекты данных?
 - `actors` — массив длиной 30, имеющий тип `char`.
 - `betsie` — массив длиной 100, имеющий тип `short`.
 - `chuck` — массив длиной 13, имеющий тип `float`.
 - `dipsea` — массив длиной 64, имеющий тип `long double`.
 - Объявите массив из пяти переменных, имеющих тип `int`, и заполните его первыми пятью целыми нечетными числами.
 - Сконструируйте оператор, который присваивает сумму пяти первых и последних элементов массива из вопроса 2 переменной `even`.
 - Сконструируйте оператор, который показывает значение второго элемента из массива `ideas` типа `float`.
 - Объявите массив типа `char` и присвойте ему значение строки "cheeseburger".
 - Придумайте объявление структуры, описывающей рыбу. Структура должна включать вид, вес в целых граммах и длину в долях метра.
 - Объявите переменную типа, объявленного в вопросе 6, и присвойте ей начальное значение.
 - Используйте `enum` для определения типа `Response` со значениями "Да", "Нет" и "Возможно". "Да" должно иметь порядковый номер 1, "Нет" — 0 и "Возможно" — 2.
 - Предположим, что переменная `ted` имеет тип `double`. Объявите указатель на `ted` и используйте его для вывода значения переменной `ted`.
 - Предположим, что `treacle` является массивом из 10 переменных типа `float`. Определите указатель, который указывает на первый элемент массива `treacle`, и используйте указатель для вывода первого и последнего значений массива.
 - Создайте фрагмент кода, который просит пользователя вводить положительные целые числа, а затем создает динамический массив из этих значений типа `int`.
 - Является ли следующий код правильным? Если да, то что он выведет?
- ```
cout << (int *) "Home of the jolly bytes";
```
- Напишите фрагмент кода, который динамически определяет структуру типа, описанного в вопросе 6, а затем считывает значение для элемента этой структуры.

14. Листинг 4.6 иллюстрирует задачу с текущим цифровым вводом со строчно-ориентированным вводом строк. Как бы замена

```
cin.getline(address, 80);
на
cin >> address;
```

повлияла на работу программы?

## Упражнения по программированию

1. Напишите программу на C++, которая запрашивает и выводит информацию как показано ниже. Обратите внимание, что в этой программе должна быть возможность принимать имена, состоящие более чем из одного слова. Заметьте также, что программа упорядочивает значения по убыванию. Предположим, что пользователь запрашивает A, B или C, поэтому вам не нужно беспокоиться о промежутке между D и F.

```
What is your first name? Betty Sue
What is your last name? Yew
What letter grade do you deserve? B
What is your age? 22
Name: Yew, Betty Sue
Grade: C
Age: 22
```

2. Структура **CandyBar** содержит три элемента. Первый элемент хранит имя производителя конфеты-батончика. Второй — вес (определенный с помощью вещественного числа) конфеты-батончика, а третий — количество калорий (целое значение) в конфете-батончике. Напишите программу, которая объявляет

такую структуру и создает переменную **CandyBar** с именем **snack**, присваивая элементам этой структуры значения "Mocha Munch", 2.3 и 350 соответственно. Присваивание должно быть частью объявления **snack**. В результате программа должна вывести содержимое переменной **snack**.

3. Структура **CandyBar** содержит три элемента, как описано в упражнении по программированию 2. Напишите программу, которая создает массив из трех структур **CandyBar**, присваивает им значения на ваш выбор и затем выводит содержимое каждой структуры.

4. Уильям Вингейт предлагает услуги по учету пиццы. Для каждой пиццы он записывает такую информацию:

- Имя компании-производителя пиццы, которое может содержать более одного слова
- Диаметр пиццы
- Вес пиццы

Придумайте структуру, которая может хранить такую информацию, и напишите программу, использующую переменную специального типа для такой структуры. Программа должна предлагать пользователю ввести информацию для каждого из названных пунктов и затем выводить эту информацию. Используйте **cin** (или его методы) и **cout**.

5. Напишите программу к упражнению 2, но используйте оператор **new** для распределения структуры вместо объявления структурной переменной. Программа должна также запрашивать диаметр пиццы до ввода названия компании — производителя пиццы.

# Циклы и выражения сравнения

**В этой главе рассматривается следующее:**

- Цикл `for`
- Выражения и операторы
- Операторы инкремента и декремента: `++` и `--`
- Комбинированные операторы присваивания
- Составные операторы (блоки)
- Оператор "запятая"
- Операторы сравнения: `>`, `>=`, `==`, `<=`, `<` и `!=`
- Цикл `while`
- Инstrumentальное средство `typedef`
- Цикл `do while`
- Метод ввода символов `get()`
- Условие достижения конца файла
- Вложенные циклы и двумерные массивы

Компьютеры предназначены отнюдь не только для хранения данных. Они анализируют, объединяют, реорганизуют, извлекают, видоизменяют, экстраполируют, синтезируют или иным образом манипулируют данными. Они даже искажают и портят данные, однако мы пытаемся избегать такого рода поведения. Для осуществления своих удивительных манипуляций программам необходимы инструментальные средства, позволяющие выполнять повторяющиеся действия или принимать решения. Безусловно, такие инструментальные средства имеются в C++. В самом деле, здесь используются те же самые циклы `for`, `while`, `do while`, операторы `if` и `switch`, которые используются в обычном языке C, поэтому те, кто знает C, могут пропустить эту главу и перейти к следующей. (Но прежде чем это сделать, все-таки просмотрите эту главу и прочитайте, как в методе `cin` обрабатывается ввод символов!) Поведение различных управляющих операторов программы может изменяться с помощью логических выражений и выражений сравнения. В этой главе рассматриваются циклы и выражения сравнения, а в следующей главе эту тему продолжит описание операторов перехода и логических выражений.

## Обзор возможностей цикла `for`

Обстоятельства нередко вынуждают программу выполнять неоднократно повторяющиеся задачи, например, такие, как поочередное сложение элементов массива или двадцатикратная распечатка какого-нибудь хвалебного

отзыва. Имеющийся в C++ цикл `for` упрощает эту задачу. Обратимся к листингу 5.1, посмотрим, что он позволяет делать, а затем обсудим, каким образом он действует.

### Листинг 5.1 Программа `forloop.cpp`.

---

```
// forloop.cpp -- Краткий обзор возможностей
// цикла for
#include <iostream>
using namespace std;
int main()
{
 int i; // Создать переменную цикла
 // Инициализация, проверка
 // и обновление цикла
 for (i = 0; i < 5; i++)
 cout << "C++ knows loops.\n";
 cout << "C++ knows when to stop.\n";
 return 0;
}
```

---

Ниже приведен результат выполнения данной программы:

```
C++ knows loops.
C++ knows when to stop.
```

Рассматриваемый цикл начинается с присвоения значения 0 целочисленной переменной `i` (здесь `i` играет роль переменной цикла):

```
i = 0
```

Эта часть цикла называется *инициализацией цикла*. Затем в той части цикла, которая называется *проверкой цикла*, программа проверяет, является ли значение переменной *i* меньше 5:

```
i < 5
```

Если это так, тогда программа выполняет следующий оператор, который образует *тело цикла*:

```
cout << "C++ knows loops.\n";
```

Затем программа использует ту часть цикла, где происходит обновление *переменной цикла* (в нашем случае *i* увеличивается на единицу):

```
i++
```

При этом используется оператор *++*, который называется *оператором инкремента*. Он увеличивает на единицу значение своего операнда. Применение оператора инкремента отнюдь не ограничивается циклами *for*. Например, оператор

```
i++;
```

может быть использован вместо оператора

```
i = i + 1;
```

в качестве типичного оператора программы. Увеличением на единицу значения переменной *i* завершается первая итерация цикла.

Далее новая итерация цикла начинается со сравнения нового значения переменной *i* с числом 5. А поскольку новое значение этой переменной (1) также меньше 5, то в цикле выводится еще одна строка, после чего он опять завершается увеличением на единицу значения переменной *i*. Это служит основанием для новой проверки цикла, выполнения операторов и обновления значения переменной *i*. Этот процесс продолжается до тех пор, пока в результате обновления значение переменной *i* не станет равным 5. Тогда последующая проверка завершится неудачно, и программа перейдет к оператору, следующему за циклом.

## Составные элементы цикла *for*

Цикл *for* предоставляет способ поэтапного выполнения повторяющихся действий. Элементы цикла *for* позволяют выполнять такие этапы, как:

- Установка первоначального значения
- Проверка на предмет продолжения цикла
- Выполнение действий внутри цикла
- Обновление значений, используемых для проверки

Конструкция цикла C++ позволяет располагать эти элементы таким образом, чтобы их можно было сразу же обнаружить. Действия, связанные с инициализацией, проверкой и обновлением переменной цикла, образуют

область управления циклом, состоящую из трех частей и заключенную в скобки. Каждая часть является выражением, причем точки с запятой отделяют одно выражение от другого. Оператор, который следует после области управления циклом, образует *тело цикла* и выполняется до тех пор, пока не становится ложным *условие продолжения цикла*:

```
for (инициализация цикла; условие продолжения цикла; обновление переменной цикла)
 тело цикла
```

Цикл *for* рассматривается в синтаксисе C++ в качестве единого оператора, хотя в его теле может входить один или более операторов.

*Инициализация цикла* выполняется только один раз. Как правило, это выражение применяется для задания начального значения переменной цикла, после чего данная переменная может использоваться для подсчета количества итераций цикла (в этом случае ее называют *счетчиком цикла*).

*Условие продолжения цикла* определяет, следует ли завершить выполнение цикла. Как правило, это выражение является выражением сравнения. В рассматриваемом примере значение переменной *i* сравнивается с числом 5, чтобы определить, является ли оно меньше 5. Если результат сравнения оказывается истинным, тогда программа выполняет тело цикла. На самом деле в качестве *условия продолжения цикла* в C++ может применяться не только выражение сравнения, образующее значения типа "истина-ложь". Для этого может быть использовано любое выражение. Так, если в результате вычисления подобного выражения получается нуль, тогда цикл завершается. А если результат вычисления выражения оказывается ненулевым, тогда выполнение цикла продолжается. В листинге 5.2 это показано на примере использования выражения с переменной *i*, используемой для проверки условия цикла. (В выражении обновления переменной цикла оператор *i--* может использоваться наравне с оператором *i++*.)

### Листинг 5.2. Программа num\_test.cpp

```
// num_test.cpp -- Использование числового
// теста в цикле
#include <iostream>
using namespace std;
int main()
{
 cout << "Enter the starting countdown value: ";
 int limit;
 cin >> limit;
 int i;
 for (i = limit; i; i--) // Выйти из цикла,
 // когда i = 0
 cout << "i = " << i << "\n";
 cout << "Done now that i = " << i << "\n";
 return 0;
}
```

Ниже приведен результат выполнения указанной выше программы:

```
Enter the starting countdown value: 4
i = 4
i = 3
i = 2
i = 1
Done now that i = 0
```

Следует заметить, что данный цикл завершается, когда переменная *i* достигает значения 0.

Каким же образом такое выражение, как *i < 5*, вписывается в рассмотренную выше схему завершения цикла при нулевом значении? Первоначально результатом вычисления выражения сравнения была 1, если оно оказывалось истинным, и 0, если это выражение оказывалось ложным. Таким образом, значение выражения *3 < 5* было равно 1, а значение выражения *5 < 5* было равно 0. Однако теперь, когда в C++ появился тип данных *bool*, в результате вычисления выражений сравнения вместо значений 1 и 0 получаются литералы *true* и *false* типа *bool*. Тем не менее, указанное изменение не приводит к несовместимости, так как программа, написанная на языке C++, преобразует значения *true* и *false* в значения 1 и 0 там, где предполагаются целые значения. А там, где предполагаются значения типа *bool*, она преобразует значение 0 в значение *false*, а ненулевое значение в значение *true*.

Цикл *for* является циклом с входным условием (или с предусловием). Это означает, что *условие продолжения цикла* проверяется перед выполнением каждой итерации цикла. При этом тело цикла вообще не выполняется, если не выполняется *условие продолжения цикла*. Допустим, например, что приведенная в листинге 5.2 программа выполняется повторно, однако на сей раз в качестве начального значения задан 0. Вследствие того что *условие продолжения цикла* ложно изначально, тело цикла вообще не будет выполнено:

```
Enter the starting countdown value: 0
Done now that i = 0
```

Благодаря этому можно избежать возможных затруднений в программе.

*Обновление переменной цикла* происходит в конце цикла после выполнения его тела. Обычно это приводит к увеличению или уменьшению значения переменной цикла на величину, которая называется *шагом цикла*.

В качестве выражений, управляющих циклом, может использоваться любое действительное выражение C++. Это делает цикл *for* способным на нечто большее чем простой подсчет чисел от 0 до 5, как это имело место в первом примере цикла. Примеры более изощренного применения оператора *for* приведены далее в этой главе.

Тело цикла *for* состоит из одного оператора, однако вскоре станет ясно, каким образом это правило может быть расширено. На рис. 5.1 приведена общая структура цикла *for*.

Оператор *for* подобен вызову функции, поскольку в нем используется имя с последующей парой скобок. Однако положение, которое оператор *for* занимает в качестве ключевого слова C++, избавляет компилятор от необходимости рассматривать его в качестве функции. Кроме того, это препятствует присвоению функции имени *for*.

### СОВЕТ

Распространенным стилем написания программ на C++ является размещение пробела между оператором *for* и последующими скобками, а также пропуск пробела между именем функции и последующими скобками:

```
for (int i = 6; i < 10; i++)
 smart_function(i);
```

Аналогично оператору *for* трактуются и другие управляющие операторы, в частности, операторы *if* и *while*. Это позволяет подчеркнуть наглядное отличие между управляющим оператором и вызовом функции. Кроме того, распространенной практикой является создание отступа в теле цикла *for* для его наглядного выделения.

### Выражения и операторы

В области управления циклом *for* используются три выражения. Невзирая на ограничения синтаксиса C++, язык все же не теряет свою выразительность. Любое

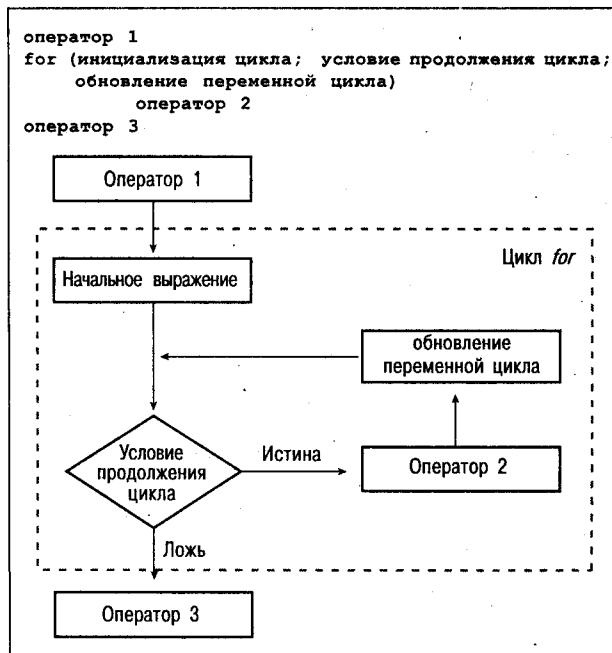


РИСУНОК 5.1 Цикл *for*.

значение либо любая допустимая комбинация значений и операций рассматривается в качестве выражения. Например, 10 является выражением со значением 10 (что неудивительно), 28 \* 20 является выражением со значением 560. В C++ у каждого выражения имеется определенное значение. Нередко это значение вполне очевидно. Например, выражение

**22 + 27**

образовано из двух значений и оператора сложения, а его значение равно 49. Иногда значения выражения оказывается менее очевидным. Например,

**x = 20**

является выражением, поскольку оно образовано из двух значений и оператора присваивания. В C++ значение оператора присваивания определяется как значение расположенного слева компонента, поэтому значение приведенного выше выражения равно 20. Тот факт, что у выражений присваивания имеются значения, допускает наличие таких операторов, как следующий:

**maids = (cooks = 4) + 3;**

Значение выражения **cooks = 4** равно 4, поэтому переменной **maids** присваивается значение 7. Однако, один лишь факт, что в C++ допускается подобное поведение, отнюдь не означает, что его следует поощрять. Тем не менее, правило, которое допускает данный конкретный оператор, позволяет использовать и следующий оператор:

**x = y = z = 0;**

Это быстрый способ задания одного и того же значения для нескольких переменных. Как следует из таблицы приоритетов операций, приведенной в приложении D, присваивание осуществляется справа налево, поэтому сначала нуль присваивается переменной **z**, а затем значение выражения **z = 0** присваивается переменной **y** и т.д.

И наконец, как упоминалось выше, в результате вычисления таких выражений отношения, как **x < y**, получаются значения **true** или **false** типа **bool**. В короткой программе, приведенной в листинге 5.3, демонстрируются некоторые особенности вычисления значений выражений. При этом оператор **<<** обладает более высоким приоритетом, чем операторы, используемые в выражениях, поэтому в данной программе для подчеркивания правильного порядка вычислений используются скобки.

### Листинг 5.3 Программа express.cpp.

```
// express.cpp - значения выражений
#include <iostream>
using namespace std;
int main()
{
```

```
 int x;
 cout << "The expression x = 100 has the value ";
 cout << (x = 100) << "\n";
 cout << "Now x = " << x << "\n";
 cout << "The expression x < 3 has the value ";
 cout << (x < 3) << "\n";
 cout << "The expression x > 3 has the value ";
 cout << (x > 3) << "\n";
 cout.setf(ios_base::boolalpha);
 cout << "The expression x < 3 has the value ";
 cout << (x < 3) << "\n";
 cout << "The expression x > 3 has the value ";
 cout << (x > 3) << "\n";
 return 0;
}
```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В прежних реализациях C++ в качестве аргумента функции **cout.setf()**, возможно, потребуется использовать **ios::boolalpha** вместо **ios\_base::boolalpha**. А в еще более старых реализациях подобная форма может быть вообще не распознана.

Ниже приведен результат выполнения указанной выше программы:

```
The expression x = 100 has the value 100
Now x = 100
The expression x < 3 has the value 0
The expression x > 3 has the value 1
The expression x < 3 has the value false
The expression x > 3 has the value true
```

Как правило, метод **cout** преобразует значения типа **bool** в значения типа **int** перед их отображением, однако при вызове функции **cout.setf(ios::boolalpha)** устанавливается признак, который дает методу **cout** команду отображать слова **true** и **false** вместо значений **1** и **0**.

### ПОМНИТЕ

Выражение в C++ является значением или сочетанием значений и операторов, причем значение имеется у каждого выражения C++.

Для вычисления выражения **x = 100** в C++ переменной **x** должно быть присвоено значение 100. Когда в результате действия, связанного собственно с вычислением выражения, значение в памяти изменяется, говорят, что вычисление обладает  **побочным эффектом**. Таким образом, побочный эффект вычисления выражения присваивания состоит в изменении значения объекта, которому это значение присваивается. Присваивание можно рассматривать в качестве подразумеваемого действия, однако с точки зрения принципов построения C++ вычисление выражения является основным действием. Побочные эффекты имеются отнюдь не у всех выражений. Например, в результате вычисления выражения **x + 15** получается новое значение, однако оно не изменяет значение переменной **x**. А вот при вычис-

лении выражения `++x + 15` проявляется побочный эффект, поскольку при этом происходит приращение значения переменной `x`.

Выражение от оператора отделяет лишь один короткий шаг. Для этого достаточно указать точку с запятой. Таким образом,

```
age = 100
```

является выражением, тогда как

```
age = 100;
```

является оператором. Любое выражение может стать оператором, если в него добавить точку с запятой, однако такое действие может в итоге не иметь никакого смысла с точки зрения программирования. Например, если `rodents` является переменной, тогда

```
rodents + 6; // действительный и, тем
// не менее, бесполезный оператор
```

является действительным оператором C++. Компилятор допускает такой оператор, однако он ничего полезного не выполняет. При этом программа вычисляет лишь сумму, ничего с ней не делает и переходит к следующему оператору. (Компилятор с развитой логикой может даже пропустить подобный оператор.)

## Операторы и выражения

Знание некоторых понятий, в частности структуры цикла `for`, имеет решающее значение для понимания C++. Однако существуют и менее значительные аспекты синтаксиса, которые неожиданно сбивают с толку, когда кажется, что рассматриваемый язык понятен. Здесь будут рассмотрены некоторые из подобных аспектов.

Несмотря на то что добавление точки с запятой в любое выражение действительно превращает это выражение в оператор, удаление точки запятой из оператора отнюдь не обязательно превращает его в выражение. Из всех операторов, которые были до сих пор рассмотрены, операторы возврата, операторы объявления и операторы `for` не подпадают под следующее правило: *оператор = выражение + точка с запятой*. Например, несмотря на то что

```
int toad;
```

является оператором, фрагмент кода `int toad` не является выражением и не имеет значения. В итоге следующий код оказывается неправильным:

```
eggs = int toad * 100; // неверно,
// поскольку не является выражением
cin >> int toad; // объявление нельзя
// объединять с оператором cin
```

Аналогично цикл `for` не является выражением и его нельзя присвоить переменной:

```
int fx = for (int i = 0; i < 4; i++)
cout >> i; // неверно
```

Здесь цикл `for` не является выражением, поэтому у него нет значения, которое может быть присвоено.

## Незначительное нарушение правил

По сравнению с C в циклы C++ введено свойство, которое требует выполнения некоторой искусственной настройки синтаксиса цикла `for`. Первоначально синтаксис этого цикла имел следующий вид:

```
for (выражение; выражение; выражение)
 оператор
```

В частности, область управления в структуре цикла `for` состояла из трех определенных выше выражений, разделенных точками с запятой. Тем не менее, циклы C++ позволяют делать следующее:

```
for (int i = 0; i < 5; i++)
```

Это означает, что можно объявлять переменную в области инициализации цикла `for`. Подобное свойство может оказаться удобным, однако оно не соответствует первоначальному синтаксису языка C, поскольку объявление не является выражением. Такое незаконное поведение первоначально было согласовано благодаря определению нового вида выражения, *выражения, включающего оператор объявления*, которое представляло собой объявление без точки с запятой и которое могло использоваться только в операторе `for`. В результате синтаксис оператора `for` был видоизменен следующим образом:

```
for (объявление и инициализация
переменной цикла условие продолжения цикла;
обновление переменной цикла)
 тело цикла
```

На первый взгляд такой синтаксис выглядит странно, поскольку вместо двух точек с запятой в нем имеется лишь одна. Тем не менее, здесь все верно, поскольку *объявление и инициализация переменной цикла* идентифицируется как оператор, а любой оператор имеет собственную точку с запятой. Подобное синтаксическое правило предполагает замену выражения, за которым следует точка с запятой, оператором с собственной точкой с запятой. Все это приводит к тому, что программисты, которые пишут на C++, имеют возможность объявлять и присваивать начальное значение переменной в области инициализации цикла `for`.

Существует один практический аспект объявления переменной в области инициализации цикла, о котором следует знать. Такая переменная существует только внутри оператора `for`. Это означает, что, после того как программа покинет данный цикл, указанная переменная прекращает свое существование:

```
for (int i = 0; i < 5; i++)
 cout << "C++ knows loops.\n";
cout << i << endl; // переменная i
 // больше не существует
```

Кроме того, некоторые реализации C++ придерживаются более старого правила и рассматривают приведенный выше цикл таким образом, как будто переменная *i* определена до этого цикла, в результате чего она становится доступной после завершения цикла. Применение указанной новой возможности объявления переменной в области инициализации цикла *for* приводит к разному поведению в различных системах, по крайней мере, в настоящее время.

### ПРЕДОСТЕРЕЖЕНИЕ

На момент написания этой книги рассматриваемое правило объявления переменной в области управления циклом *for* и прекращения ее действия по завершении цикла было внедрено еще не во всех компиляторах рассматриваемого языка.

## Снова о цикле *for*

Рассмотрим циклы немного с другой точки зрения. В листинге 5.4 цикл используется для подсчета и сохранения значений первых 16 факториалов. Факториалы, которые удобны для вычисления разности, подсчитываются следующим образом. Нулевой факториал записывается как  $0!$  и, по определению, равен 1. Затем  $1!$  равен  $1 * 0!$  или 1. Далее  $2!$  равен  $2 * 1!$  или 2. Затем  $3!$  равен  $3 * 2!$  или 6 и т.д. Таким образом, факториал каждого целого числа является произведением этого целого числа на предыдущий факториал. В рассматриваемой программе для последовательного подсчета факториалов и сохранения их в массиве используется один цикл. Затем в ней используется другой цикл для отображения результатов подсчета. Кроме того, в этой программе представлено применение внешних объявлений значений переменных.

### Листинг 5.4 Программа *formore.cpp*.

```
// formore.cpp - дополнительные возможности
// цикла for
#include <iostream>
using namespace std;
const int ArSize = 16;
// пример внешнего объявления
int main()
{
 double factorials[ArSize];
 factorials[1] = factorials[0] = 1.0;
 int i;
 for (i = 2; i < ArSize; i++)
 factorials[i] = i * factorials[i-1];
 for (i = 0; i < ArSize; i++)
 cout << i << "!" = "
 << factorials[i] << "\n";
 return 0;
}
```

Ниже приведен результат выполнения указанной выше программы:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3.6288e+006
11! = 3.99168e+007
12! = 4.79002e+008
13! = 6.22702e+009
14! = 8.71783e+010
15! = 1.30767e+012
```

Как можно заметить, значения факториалов быстро увеличиваются!

### Примечания к программе

Приведенная выше программа создает массив для хранения значений факториалов. При этом нулевой элемент массива равен  $0!$ , первый элемент массива равен  $1!$  и т.д. Вследствие того что первые два факториала равны 1, данная программа устанавливает значение 1.0 для первых двух элементов массива *factorials*. (Не следует забывать, что индекс первого элемента массива равен 0.) После этого в данной программе используется цикл для задания произведения индекса массива на предыдущий факториал в качестве значения каждого факториала. В этом цикле демонстрируется возможность использовать счетчик цикла как обычную переменную в теле цикла.

В рассматриваемой программе показано также, каким образом цикл *for* действует в тесной связи с массивами, обеспечивая при этом удобные средства поочередного доступа к каждому элементу массива. Кроме того, в программе *formore.cpp* переменная типа *const* используется для создания символического представления размера массива (*ArSize*). При этом переменная *ArSize* используется всякий раз, когда дело касается размера массива, в частности, при определении массива и ограничений для циклов, которые обрабатывают массив. А теперь, если имеется желание расширить возможности данной программы, скажем, до вычисления 20 факториалов, то для этого достаточно установить в этой программе значение 20 для переменной *ArSize* и повторно скомпилировать ее. Таким образом, применение символьской константы позволяет избежать изменения констант от 16 до 20 в каждом отдельном случае.

**СОВЕТ**

Как правило, значение переменной типа **const** имеет смысл определять для представления числа элементов массива. Это значение следует использовать в объявлении массива, а также во всех других ссылках на размер массива, в частности, в цикле **for**.

Условие продолжения цикла  $i < \text{ArSize}$  отражает тот факт, что индексы массива с числом элементов, равным **ArSize**, находятся в пределах от 0 до **ArSize - 1**, поэтому индексация массива должна прекратиться за единицу до значения **ArSize**. Вместо данной проверки можно было бы оценить выражение  $i \leq \text{ArSize}$ , однако этот способ является довольно неудобным.

Одна из интересных особенностей рассматриваемой программы состоит в том, что объявление переменной **ArSize** типа **const int** осуществляется вне тела функции **main()**. Как упоминается в конце главы 4, это превращает значение **ArSize** во внешнее значение. При объявлении переменной **ArSize** подобным образом проявляются два следующих результата: переменная **ArSize** существует на протяжении всей программы, и поэтому она может быть использована во всех функциях в других программных файлах. В данном конкретном случае в программе имеется только одна функция, поэтому практическое значение внешнего объявления переменной **ArSize** невелико. Однако в программах с множеством функций нередко можно извлечь пользу из совместного использования внешних констант, далее будет показано, как это выглядит на практике.

## Изменение шага цикла

В приведенных ранее примерах переменная цикла увеличивалась или уменьшалась на 1 на каждой итерации цикла. Это приращение переменной цикла (или *шаг цикла*) можно изменить в выражении *обновления переменной цикла*. Например, в программе, приведенной в листинге 5.5, переменная цикла увеличивается на величину, выбранную пользователем. Вместо **i++** в качестве выражения обновления переменной цикла в данном случае используется выражение  $i = i + \text{by}$ , где **by** — это выбираемая пользователем величина шага цикла.

### Листинг 5.5 Программа **bigstep.cpp**.

```
// bigstep.cpp -- направленный подсчет
#include <iostream>
using namespace std;
int main()
{
 cout << "Enter an integer: ";
 int by;
 cin >> by;
 cout << "Counting by " << by << "s:\n";
 for (int i = 0; i < 100; i = i + by)
 cout << i << "\n";
 return 0;
}
```

Ниже приведен результат выполнения данной программы:

```
Enter an integer: 17
Counting by 17s:
0
17
34
51
68
85
```

Когда значение переменной **i** достигает 102, цикл завершается. Основная особенность в данном случае состоит в том, что в качестве выражения обновления цикла может служить любое действительное выражение. Например, если на каждой итерации цикла требуется возвести значение переменной **i** в квадрат и прибавить к нему 10, то для этого можно воспользоваться выражением  $i = i * i + 10$ .

## Доступ к символам строки с помощью цикла **for**

Цикл **for** предоставляет непосредственный доступ поочередно к каждому символу строки. Например, в листинге 5.6 приведена программа, которая дает возможность вводить строку, а затем отображать эту строку посимвольно в обратном порядке. При этом функция **strlen()** подсчитывает количество символов в строке, а затем это значение используется в выражении инициализации цикла для присвоения **i** значения позиции последнего символа в строке, не считая пустого символа. Для перемещения по строке в обратном порядке в данной программе используется оператор декремента (**--**), который уменьшает на 1 индекс массива на каждой итерации цикла. Кроме того, в листинге 5.6 для проверки достижения в цикле первого элемента используется оператор сравнения "больше или равно" (**>=**). Информация об операторах сравнения будет приведена далее.

### Листинг 5.6 Программа **forstr1.cpp**.

```
// forstr1.cpp -- использование цикла for
// для обработки строк
#include <iostream>
#include <cstring>
using namespace std;
const int ArSize = 20;
int main()
{
 cout << "Enter a word: ";
 char word[ArSize];
 cin >> word;

 // отображение символов строки в обратном порядке
 for (int i = strlen(word)-1; i >= 0; i--)
 cout << word[i];
 cout << "\n";
 return 0;
}
```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если в применяемой реализации рассматриваемого языка еще не введены новые заголовочные файлы, тогда, возможно, вместо файла `string.h` придется воспользоваться файлом `cstring`.

Ниже приведен результат выполнения данной программы:

```
Enter a word: animal
lamina
```

В самом деле, данная программа выводит слово `animal` (животное) в обратном порядке. Это слово было выбрано в качестве проверочного для того, чтобы яснее продемонстрировать действие данной программы.

## Операторы инкремента (++) и декремента (--)

Для C++ характерно наличие нескольких операторов, которые часто используются в циклах, поэтому уделим теперь некоторое время их рассмотрению. С двумя из них читатель уже ознакомился. Это оператор инкремента (++) , а также оператор декремента (--). Эти операторы выполняют две чрезвычайно распространенные в цикле операции: увеличение и уменьшение счетчика цикла на 1. Однако их возможности этим не ограничиваются. Существует две разновидности каждого из этих операторов. *Предфиксный* вариант выполняется до обработки операнда, как, например, в выражении `++x`. А *постфиксный* вариант выполняется после обработки операнда, как, например, в выражении `x++`. Оба варианта оказывают на operand одно и то же действие, однако происходит это у них по-разному. Их действие подобно предварительной или последующей оплате за поставки товара. Оба метода в конечном итоге оказывают одно и то же влияние на кошелек, однако деньги приходят в том и в другом случае в разное время. Это отличие в операторе инкремента демонстрируется в листинге 5.7.

### Листинг 5.7 Программа plus\_one.cpp.

```
// plus_one.cpp - оператор инкремента
#include <iostream>
using namespace std;
int main()
{
 int a = 20;
 int b = 20;

 cout << "a = " << a << ": b = "
 << b << "\n";
 cout << "a++ = " << a++ << ": ++b = "
 << ++b << "\n";
 cout << "a = " << a << ": b = "
 << b << "\n";
 return 0;
}
```

Ниже приведен результат выполнения указанной программы:

```
a = 20: b = 20
a++ = 20: ++b = 21
a = 21: b = 21
```

Запись `a++` означает примерно следующее: "Использовать текущее значение `a` при вычислении выражения, а затем увеличить значение `a`" на 1. Аналогично запись `++b` означает следующее: "Сначала увеличить значение `b` на 1, а затем использовать новое значение при вычислении выражения". Например, имеются следующие отношения:

```
int x = 5;
int y = ++x; // изменить значение x, а
 // затем присвоить это значение y
 // y равно 6 и x равно 6
int z = 5;
int y = z++; // присвоить значение y, а
 // затем изменить значение z
 // y равно 5, а z равно 6
```

Операторы инкремента и декремента представляют собой краткий и удобный способ выполнения распространенных задач увеличения или уменьшения значений на 1. Их можно использовать вместе с указателями, а также с обычными переменными. Напомним, что в результате прибавления 1 к указателю его значение увеличивается на число байтов в типе данных, на который он указывает. Это же правило справедливо и в случае применения к указателям операторов инкремента и декремента.

### ПОМНИТЕ

Увеличение и уменьшение указателей выполняется в соответствии с правилами арифметики указателей. Поэтому если указатель `pt` указывает на первый элемент массива, тогда оператор `++pt` изменяет указатель `pt` таким образом, чтобы он указывал на второй элемент массива.

Операторы инкремента и декремента являются небольшими, изящными операторами, однако при этом нельзя увлекаться, увеличивая или уменьшая одно и то же значение более одного раза в одном и том же выражении. Ведь проблема состоит в том, что правила использования и последующего изменения, а также изменения и последующего использования могут оказаться неоднозначными. Это означает, что такой оператор, как

```
x = 2 * x++ * (3 - ++x); // этого делать нельзя
```

может дать совершенно разные результаты в разных системах. Правильное поведение подобного рода оператора в C++ не определяется.

## Комбинированные операторы присваивания

В листинге 5.5 для обновления переменной цикла используется следующее выражение:

```
i = i + by
```

В C++ имеется комбинированный оператор сложения и присваивания, который дает тот же результат и является более компактным:

```
i += by
```

Оператор `+=` выполняет сложение двух своих операндов и присваивает полученный результат левому операнду. При этом подразумевается, что левый операнд должен представлять собой некоторый объект, которому может быть присвоено значение, в частности, им может быть переменная, элемент массива, компонент структуры или данные, которые идентифицируются с помощью разыменования указателя:

```
int k = 5;
k += 3; // правильно, переменной k
 // присваивается значение 8

int *pa = new int[10]; // указатель pa
 // указывает на элемент массива pa[0]
pa[4] = 12;
pa[4] += 6; // правильно, элементу массива
 // pa[4] присваивается 18
*(pa + 4) += 7; // правильно, элементу
 // массива pa[4] присваивается значение 25
pa += 2; // указатель pa указывает на
 // предыдущий элемент массива pa[2]
34 += 10; // совершенно неверно
```

Для каждой арифметической операции имеется соответствующий оператор присваивания, как следует из табл. 5.1. При этом каждый такой оператор действует аналогично оператору `+=`. Таким образом, оператор

```
k *= 10;
```

заменяет текущее значение `k` на большее в 10 раз значение.

**Таблица 5.1 Комбинированные операторы присваивания.**

| Операция        | Действие ( <i>L</i> — левый операнд,<br><i>R</i> — правый операнд) |
|-----------------|--------------------------------------------------------------------|
| <code>+=</code> | <i>L</i> + <i>R</i> присваивается <i>L</i>                         |
| <code>-=</code> | <i>L</i> - <i>R</i> присваивается <i>L</i>                         |
| <code>*=</code> | <i>L</i> * <i>R</i> присваивается <i>L</i>                         |
| <code>/=</code> | <i>L</i> / <i>R</i> присваивается <i>L</i>                         |
| <code>%=</code> | <i>L</i> % <i>R</i> присваивается <i>L</i>                         |

## Составные операторы или блоки

Формат или синтаксис оператора `for` в C++ может показаться ограниченным, поскольку тело данного цикла должно состоять из одного оператора. Это неудобно в том случае, если требуется, чтобы тело цикла содержало несколько операторов. К счастью, в C++ существует специальный синтаксический прием, позволяющий обойти это ограничение и вставить в тело цикла сколько угодно операторов. Этот прием состоит в использовании спаренных фигурных скобок для построения *составного оператора* или *блока*. Блоки состоят из спаренных фигурных скобок и операторов, которые в них заключены, причем с точки зрения синтаксиса они считаются единым оператором. Например, в программе, приведенной в листинге 5.6, фигурные скобки используются для объединения трех отдельных операторов в единый блок. Это дает возможность ввести в теле цикла приглашение для пользователя, прочитать ввод и выполнить подсчет. В данной программе выполняется подсчет текущей суммы вводимых чисел и, таким образом, вам предоставляется естественная возможность воспользоваться оператором `+=`.

**Листинг 5.8 Программа block.cpp.**

```
// block.cpp -- программа, демонстрирующая
// применение блока операторов
#include <iostream>
using namespace std;
int main()
{
 cout << "The Amazing Accounto will sum and
 average ";
 cout << "five numbers for you.\n";
 cout << "Please enter five values:\n";
 double number;
 double sum = 0.0;
 for (int i = 1; i <= 5; i++)
 {
 // здесь начинается блок
 cout << "Value " << i << ": ";
 cin >> number;
 sum += number;
 } // здесь блок оканчивается
 cout << "Five exquisite choices indeed!";
 cout << "They sum to " << sum << "\n";
 cout << "and average to " << sum / 5
 << ".\n";
 cout << "The Amazing Accounto bids you
 adieu!\n";
 return 0;
}
```

Ниже приведен результат выполнения указанной выше программы:

```
The Amazing Accounto will sum and average
five numbers for you.
Please enter five values:
Value 1: 1942
```

```

Value 2: 1948
Value 3: 1957
Value 4: 1974
Value 5: 1980
Five exquisite choices indeed! They sum to
9801
and average to 1960.2.
The Amazing Accounto bids you adieu!

```

Допустим, что отступ оставлен, однако при этом опущены фигурные скобки:

```

for (int i = 1; i <= 5; i++)
 cout << "Value " << i << ": ";
 // здесь оканчивается цикл
 cin >> number; // после цикла
 sum += number;
 cout << "Five exquisite choices indeed! ";

```

В этом случае компилятор игнорирует отступ, и поэтому в цикле окажется только один оператор. Таким образом, в цикле будет распечатано пять приглашений и больше ничего. По завершении цикла программа перейдет к следующим строкам, выполняя чтение и суммирование еще одного числа.

Составные операторы обладают еще одним интересным свойством. Если внутри блока определить новую переменную, то эта переменная будет продолжать существовать до тех пор, пока программа выполняет операторы внутри блока. А по завершении выполнения блока данная переменная становится недействительной. Это означает, что рассматриваемая переменная известна только внутри блока:

```

#include <iostream>
using namespace std;
int main()
{
 int x = 20;
 { // начало блока
 int y = 100;
 cout << x << "\n"; // правильно
 cout << y << "\n"; // правильно
 } // конец блока
 cout << x << "\n"; // правильно
 cout << y << "\n"; // неправильно,
 // компиляции не подлежит
 return 0;
}

```

Обратите внимание на то, что переменная, определенная во внешнем блоке, все еще остается определенной и во внутреннем блоке. А что произойдет, если объявить в блоке переменную с таким же именем, как и во внешнем блоке? В этом случае новая переменная скрывает из виду старую переменную до тех пор, пока данный блок не завершится. А затем старая переменная снова становится видимой.

```

int main()
{
 int x = 20; // исходная переменная x
 { // начало блока
 cout << x << "\n"; // использование
 // исходной переменной x
 int x = 100; // новая переменная x
 cout << x << "\n"; // использование
 // новой переменной x
 } // конец блока
 cout << x << "\n"; // использование
 // исходной переменной x
 return 0;
}

```

## Оператор "запятая" (или дополнительные синтаксические приемы)

Как видите, блок дает возможность вставить в обход общих правил два или более оператора в том месте, где синтаксис C++ допускает только один оператор. Оператор "запятая" делает то же самое в выражениях, допуская использование двух выражений в том месте, где синтаксис C++ допускает только одно выражение. Допустим, например, что имеется цикл, в котором одна переменная увеличивается, а вторая уменьшается на 1 на каждой итерации цикла. Обе операции было бы удобно выполнить при обновлении цикла `for`, однако синтаксис цикла допускает только одно выражение в этой части цикла. Решение подобной задачи состоит в применении оператора "запятой" для объединения двух выражений в одно:

```

j++, i-- // оба выражения считаются единым
 // выражением с точки зрения
 // синтаксиса

```

Запятая, встречающаяся в программном коде, отнюдь не всегда означает оператор "запятой". Например, запятая в объявлении

```

int i, j // здесь запятая является
 // разделителем, а не оператором

```

служит для разделения смежных имен в списке переменных.

В программе, приведенной в листинге 5.9 и переставляющей содержимое символьного массива, оператор "запятая" используется дважды. Следует заметить, что в листинге 5.6 содержимое массива отображается в обратном порядке, а в листинге 5.6 символы фактически переставляются в массиве. Кроме того, в данной программе для объединения нескольких операторов в один используется блок.

### Листинг 5.9 Программа firstr2.cpp.

```

// Программа firstr2.cpp,
// переставляющая содержимое массива
#include <iostream>
#include <cstring>

```

```

using namespace std;
const int ArSize = 20;
int main()
{
 cout << "Enter a word: ";
 char word[ArSize];
 cin >> word;

 // фактическое изменение массива
 char temp;
 int i, j;
 for (j = 0, i = strlen(word) - 1;
 j < i; i--, j++)
 { // начало блока
 temp = word[i];
 word[i] = word[j];
 word[j] = temp;
 } // конец блока
 cout << word << "\n";
 return 0;
}

```

Ниже приведен результат выполнения указанной выше программы:

```
Enter a word: parts
strap
```

### Примечания к программе

Рассмотрим область управления циклом `for`:

```
for (j = 0, i = strlen(word) - 1; j < i;
 i--, j++)
```

Сначала оператор "запятая" используется в первой части области управления циклом для объединения двух инициализаций в одном выражении. А затем оператор "запятая" используется еще раз для объединения двух обновлений в одном выражении в последней части области управления циклом.

Далее рассмотрим тело цикла. В данной программе фигурные скобки используются для объединения нескольких операторов в единое целое. В теле цикла данная программа выполняет перестановку букв в слове, заменяя первый элемент массива последним. Затем она увеличивает значение переменной `j` и уменьшает значение переменной `i` таким образом, чтобы они теперь указывали на следующий после первого элемент и предшествующий последнему элемент массива. После этого программа меняет эти элементы местами. Следует заметить, что условие проверки `j < i` приводит к завершению цикла при достижении середины массива. Если бы выполнение необходимо было продолжить далее этой точки, тогда программа начала бы переставлять элементы массива обратно в их исходное положение (рис. 5.2).

Следует также обратить внимание на местоположение объявления переменных `temp`, `i` и `j`. В данной программе переменные `i` и `j` объявляются до выполнения

цикла, поскольку объединить два объявления с помощью оператора "запятой" нельзя. Это связано с тем, что запятые уже используются в объявлении с иной целью — для разделения элементов списка. Для создания и инициализации двух переменных можно воспользоваться выражением с одним оператором объявления, однако внешне это выглядит несколько запутанно:

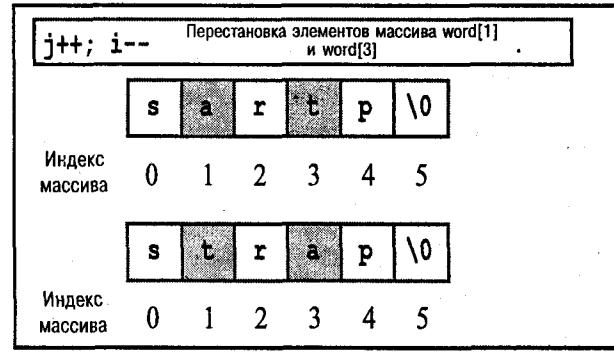
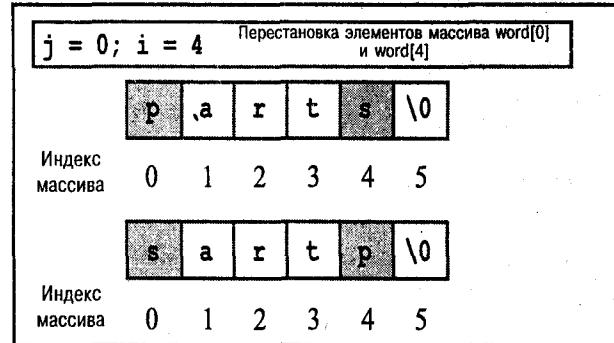
```
int j = 0, i = strlen(word) - 1;
```

В данном случае запятая служит лишь в качестве разделителя списка, а не оператора "запятой", поэтому в указанном выражении осуществляется объявление и инициализация обеих переменных — `i` и `j`. Тем не менее, создается такое впечатление, как будто объявляется только одна переменная `j`.

Кстати, переменную `temp` можно объявить и внутри цикла `for`:

```
int temp = word[i];
```

Это приводит к тому, что переменная `temp` назначается и освобождается на каждой итерации цикла. Такой способ может оказаться несколько более медленным, чем объявление переменной `temp` один раз перед началом цикла. С другой стороны, по завершении цикла переменная `temp` прекращает свое существование, если она была объявлена внутри цикла.



`j++; i--` А теперь выражение `j>1` становится ложным, и поэтому цикл завершается

РИСУНОК 5.2 Перестановка символов строки.

## Особенности применения оператора "запятая"

До сих пор оператор "запятая" применялся главным образом для объединения двух или более выражений в одно выражение, используемое в цикле `for`. Однако в C++ у этого оператора имеется два дополнительных свойства. Во-первых, он гарантирует, что первое выражение вычисляется до второго выражения. Так, безопасными являются выражения, подобные следующему:

```
i = 20, j = 2 * i // для переменной i
 // установлено значение 20,
 // а для переменной j - значение 40
```

Во-вторых, в C++ утверждается, что значение выражения с запятой равно значению второй его части. Например, значение предыдущего выражения равно 40, поскольку значение переменной `j = 2 * i`.

Оператор "запятая" обладает самым низким приоритетом среди всех операторов. Например, оператор

```
cats = 17, 240;
```

воспринимается как оператор

```
(cats = 17), 240;
```

Это означает, что для переменной `cats` устанавливается значение 17, а значение 240 нигде не используется. Однако поскольку скобки обладают более высоким приоритетом, то в результате выполнения оператора

```
cats = (17, 240);
```

для переменной `cats` устанавливается значение 240, т.е. значение выражения, размещенного справа.

## Выражения сравнения

Компьютеры представляют собой нечто большее чем средства для беспрерывного "перемалывания" чисел. Они обладают способностью сравнивать значения, и именно эта особенность служит компьютеру основанием для принятия необходимых решений. Указанная возможность воплощена в C++ в операторах сравнения. Для сравнения чисел в C++ существует шесть операторов сравнения. Вследствие того что символы представлены в коде ASCII, указанные операторы могут быть применены и к символам, однако они неприменимы к строкам, созданным в стиле С. Каждое выражение сравнения сводится к значению `true` типа `bool`, если результат сравнения оказывается истинным, и к значению `false` типа `bool`, если результат сравнения оказывается ложным. Именно поэтому данные операторы вполне пригодны для применения в *условии продолжения цикла*. (В старых реализациях языка C++ истинный результат вычисления выражений сравнения равен 1, а ложный результат равен 0.) Все рассмотренные операторы представлены в табл. 5.2.

Таблица 5.2 Операторы сравнения.

| Оператор | Значение         |
|----------|------------------|
| <        | Меньше           |
| <=       | Меньше или равно |
| ==       | Равно            |
| >        | Больше           |
| >=       | Больше или равно |
| !=       | Не равно         |

Возможности сравнения чисел в C++ исчерпываются шестью операторами сравнения. Если же требуется сравнить два значения, чтобы определить, какое из них является более подходящим, тогда подобные возможности следует искать в другом месте.

Ниже приведены некоторые примеры осуществления тестирования:

```
for (x = 20; x > 5; x--) // продолжить до
 // тех пор, пока x остается больше 5
for (x = 1; y != x; x++) // продолжить до
 // тех пор, пока y остается не равным x
for (cin >> x; x == 0; cin >> x)
 // продолжить до тех пор, пока x
 // остается равным 0
```

Операторы сравнения обладают более низким приоритетом, чем арифметические операторы. Это означает, что выражение

```
x + 3 > y - 2 // выражение 1
```

соответствует выражению

```
(x + 3) > (y - 2) // выражение 2
```

а не следующему выражению:

```
x + (3 > y) - 2 // выражение 3
```

В связи с тем что выражение `(3 > y)` равно 1 или 0 после преобразования значения из типа `bool` в тип `int`, выражения 2 и 3 являются действительными. Однако в большинстве случаев требуется, чтобы выражение 1 соответствовало выражению 2, что собственно и характерно для C++.

## Типичная ошибка программирования

Не следует путать проверку, реализуемую с помощью оператора равенства (`==`), с выражением для оператора присваивания (`=`). Так, в выражении

```
musicians == 4 // сравнение
```

выясняется, равна ли переменная `musicians` значению 4? Значение этого выражения может быть равно `true` или `false`. А в выражении

```
musicians = 4 // присваивание
```

значение 4 присваивается переменной `musicians`. В данном случае значение всего выражения равно 4, поскольку именно это значение находится в левой его части.

Гибкая конструкция цикла `for` предоставляет интересную возможность для проявления ошибки. Если случайно пропустить знак (`=`) в операторе `==` и воспользоваться выражением присваивания вместо выражения сравнения в проверочной части цикла `for`, получаемый в итоге код все равно окажется действительным. Это связано с тем, что в проверочном условии цикла `for` можно использовать любое действительное выражение C++. Напомним, что при проверке ненулевые значения соответствуют логическому значению `true`, а нулевые — логическому значению `false`. Так, значение выражения, в котором значение 4 присваивается переменной `musicians`, равно 4, и оно считается истинным (`true`). К подобным ошибкам могут быть склонны те, кто раньше пользовался языками Pascal и BASIC.

В листинге 5.10 продемонстрирована ситуация, в которой возможна подобного рода ошибка. Приведенная в этом листинге программа пытается проверить массив баллов викторины и остановиться на балле, который не равен 20. В этой программе применяется цикл, в котором показано правильное использование сравнения, а затем цикл, в условии продолжения которого ошибочно используется операция присваивания. Кроме того, в данной программе имеется еще одна виновная ошибка разработки, исправление которой будет рассмотрено далее. (Обычно учиться приходится на собственных ошибках, и в этом отношении листинг 5.10 может оказаться посильной помощь.)

### Листинг 5.10 Программа equal.cpp.

```
// equal.cpp - отличия операций сравнения
// и присваивания

#include <iostream>
using namespace std;
int main()
{
 int quizscores[10] =
 { 20, 20, 20, 20, 20, 19, 20, 18, 20, 20};

 cout << "Doing it right:\n";
 int i;
 for (i = 0; quizscores[i] == 20; i++)
 cout << "quiz " << i << " is a 20\n";
 cout << "Doing it dangerously wrong:\n";
 for (i = 0; quizscores[i] = 20; i++)
 cout << "quiz " << i << " is a 20\n";

 return 0;
}
```

Вследствие того что в этой программе имеется серьезная ошибка, возможно, лучше сначала ознакомиться с ней, а затем ее выполнять.

Ниже приведен результат выполнения данной программы:

**Doing it right:**

```
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
```

**Doing it dangerously wrong:**

```
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
quiz 5 is a 20
quiz 6 is a 20
quiz 7 is a 20
quiz 8 is a 20
quiz 9 is a 20
quiz 10 is a 20
quiz 11 is a 20
quiz 12 is a 20
quiz 13 is a 20
```

...

Первый цикл корректно прерывается после отображения первых пяти баллов викторины. Однако второй цикл начинается с отображения всего массива. Еще хуже то, что каждое значение в этом цикле выводится равным 20. И совсем плохо, что он не прерывается по достижению конца массива.

На самом деле все указанные недоразумения связаны со следующим *условием продолжения цикла*:

```
quizzescores[i] = 20
```

Во-первых, это связано с тем, что в данном выражении просто присваивается значение элементу массива, поэтому данное выражение всегда оказывается ненулевым, а следовательно, оно истинно. Во-вторых, поскольку в данном выражении элементам массива присваиваются конкретные значения, оно фактически изменяет данные. В-третьих, вследствие того что *условие продолжения цикла* остается истинным, программа продолжает изменять данные уже за пределами массива. При этом она просто продолжает вносить в память все больше и больше чисел 20! А это уже никуда не годится.

Сложность подобной ошибки состоит в том, что код с точки зрения синтаксиса оказывается правильным, поэтому компилятор не отметит его в качестве ошибочного. (Тем не менее, многие годы программисты, которые пишут на языках С и С++, совершили подобную ошибку, и в конечном итоге это привело к тому, что многие компиляторы выдают предупреждение, в котором запрашивается, действительно ли имеется в виду именно такая операция.)

**ПРЕДОСТЕРЖЕНИЕ**

Для сравнения на предмет равенства вместо оператора `=` пользуйтесь оператором `==`.

Подобно С, язык C++ предоставляет большую свободу действий, чем большинство других языков программирования. Но при этом на программиста возлагается большая ответственность. Ничто, кроме тщательного планирования программы, не может воспрепятствовать ее выходу за пределы стандартного массива C++. Тем не менее, с помощью классов C++ можно разработать защищенный тип массива, который препятствует появлению подобного рода абсурдных ситуаций. В главе 12 приведен соответствующий пример. Между тем, защиту следует встраивать в программы по мере надобности. Например, в цикл должна быть включена проверка, которая препятствует выходу за пределы последнего компонента. Это справедливо даже для "хороших" циклов. Так, если бы все баллы были равны 20, то и в этом случае произошел бы выход за пределы массива. Одним словом, в цикле необходима проверка значений массива и индекса массива. В главе 6 показано, каким образом логические операторы используются для объединения подобных проверок в одном условии.

## Сравнение строк

Допустим, что в символьном массиве требуется определить слово `mate`. Если имя массива `word`, тогда следующая проверка вряд позволит добиться предполагаемого результата:

```
word == "mate"
```

Напомним, что имя массива связано с адресом его местонахождения. Аналогично заключенная в кавычки строка связана с ее адресом. Таким образом, предыдущее выражение сравнения проверяет строки не на равенство, а на предмет хранения обеих строк по одному и тому же адресу. Подобная проверка дает отрицательный ответ, даже если в обеих строках содержатся одни и те же символы.

В связи с тем что строки в C++ рассматриваются в виде адресов, применение операторов сравнения для сравнения строк вряд принесет большое удовлетворение. Вместо этого для сравнения строк можно обратиться к библиотеке строковых функций в стиле С и воспользоваться функцией `strcmp()`. Эта функция воспринимает адреса двух строк в качестве аргументов. Это означает, что ее аргументами могут быть указатели, строковые константы или имена символьных массивов. Если обе строки одинаковы, тогда данная функция возвращает нулевое значение. Если же первая строка предшествует второй строке в алфавитном порядке, тогда функция `strcmp()` возвращает отрицательное значение, а если

первая строка следует за второй строкой в алфавитном порядке, то в этом случае функция `strcmp()` возвращает положительное значение. На самом деле расположение строк в системной сортирующей последовательности является более точным, чем в алфавитном порядке. Это означает, что символы сравниваются в соответствии с системным кодом символов. Например, в коде ASCII всем прописным буквам соответствуют меньшие значения, чем строчным буквам, поэтому прописные буквы предшествуют строчным в сортирующей последовательности. Следовательно, строка "Zoo" предшествует строке "aviary". Тот факт, что сравнение основано на значениях кодов, говорит о том, что прописные и строчные буквы отличаются друг от друга, и поэтому строка "FOO" отличается от строки "foo".

В некоторых языках, в частности в BASIC и в стандартном варианте Pascal, строки, хранящиеся в массивах разного размера, как правило, не равны друг другу. Однако строки, образованные в стиле С, определяются по завершающему нулевому символу, а не по размеру содержащего их массива. Это означает, что две строки могут быть одинаковы, даже если они содержатся в массивах разного размера:

```
char big[80] = "Daffy"; //5 букв плюс \0
char little[6] = "Daffy"; //5 букв плюс \0
```

Между прочим, несмотря на то что операторы сравнения нельзя применять для сравнения строк, их можно использовать для сравнения символов, поскольку символы на самом деле относятся к целому типу данных. Таким образом,

```
for (ch = 'a'; ch <= 'z'; ch++)
 cout << ch;
```

является действительным кодом для отображения символов алфавита, по крайней мере это касается набора символов в коде ASCII.

В листинге 5.11 функция `strcmp()` применяется в условии продолжения цикла `for`. Приведенная в этом листинге программа отображает слово, изменяет первую его букву, отображает это слово еще раз и продолжает делать это до тех пор, пока функция `strcmp()` не определит, что данное слово совпадает со строкой "mate". Следует заметить, что в данный листинг включен файл `cstring`, поскольку он предоставляет прототип функции `strcmp()`.

### Листинг 5.11 Программа compstr.cpp.

```
// Программа compstr.cpp, демонстрирующая
// сравнение строк
#include <iostream>
#include <cstring> //прототип функции strcmp()
using namespace std;
int main()
```

```

{
 char word[5] = "?ate";
 for (char ch = 'a'; strcmp(word, "mate"); ch++)
 {
 cout << word << "\n";
 word[0] = ch;
 }
 cout << "After loop ends, word is "
 << word << "\n";
 return 0;
}

```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Вместо файла `cstring`, возможно, придется воспользоваться файлом `string.h`. Кроме того, в указанной выше программе предполагается, что в системе используется набор символов в коде ASCII. В этом наборе символов коды букв от `a` до `z` расположены последовательно друг за другом, а код символа `?` непосредственно предшествует коду символа `a`.

Ниже приведен результат выполнения указанной выше программы:

```

?ate
aate
bate
cate
date
eate
fate
gate
hate
iate
jate
kate
late
After loop ends, word is mate

```

### Примечания к программе

В рассматриваемой программе имеется ряд интересных моментов. Один из них, безусловно, касается проверки. В данном случае требуется, чтобы цикл продолжался до тех пор, пока значение `word` не равно значению `mate`. Это означает, что проверка должна продолжаться до тех пор, пока функция `strcmp()` будет сообщать о том, что обе строки неодинаковы. Наиболее очевидной в этом случае является следующая проверка:

```

strcmp(word, "mate") != 0 // строки
 // неодинаковы

```

Значение данного оператора равно 1 (истинно), если строки не равны; и 0 (ложно), если они равны. А как насчет самой функции `strcmp(word, "mate")`? Ее значение не равно 0 (истинно), если строки не равны; и равно 0 (ложно), если строки равны. По существу, данная функция возвращает истинное значение, если строки отличаются друг от друга, и ложное значение, если они одинаковы. Таким образом, вместо целочисленного вы-

ражения сравнения можно воспользоваться одной лишь функцией. Результат при этом оказывается таким же, а вводить приходится меньший объем кода. Кроме того, программисты, которые пишут на C и C++, по традиции пользуются функцией `strcmp()`.

### ПОМНИТЕ

Функцию `strcmp()` следует применять для проверки равенства или порядка следования строк. При этом выражение

`strcmp(str1, str2) == 0`

истинно, если строки `str1` и `str2` одинаковы, выражение

`strcmp(str1, str2) != 0`

и

`strcmp(str1, str2)`

истинны, если строки `str1` и `str2` не равны, выражение

`strcmp(str1, str2) < 0`

истинно, если строка `str1` предшествует строке `str2`, и выражение

`strcmp(str1, str2) > 0`

истинно, если строка `str1` следует после строки `str2`. Таким образом, функция `strcmp()` может играть роль операций `==`, `!=`, `<` и `>` в зависимости от того, каким образом задано проверочное условие.

Далее в программе `compstr.cpp` применяется оператор инкремента для последовательного присваивания переменной `ch` букв всего алфавита:

`ch++`

Операторы инкремента и декремента могут применяться к символьным переменным, поскольку тип `char` на самом деле является целым типом, и поэтому подобная операция фактически изменяет код целого значения, хранящийся в переменной. Кроме того, следует заметить, что благодаря применению индекса массива изменяются отдельные символы в строке:

`word[0] = ch;`

И наконец, в отличие от большинства рассмотренных циклов `for` данный цикл не является циклом со счетчиком. Напротив, в этом цикле осуществляется отслеживание конкретной ситуации (когда значение `word` будет равно `"mate"`) для сигнализации о прекращении операции. Более распространенным в программах, написанных на C++, является применение циклов `while` для этого второго вида проверки, поэтому обратимся теперь к рассмотрению данной формы цикла.

## Цикл while

Цикл `while` представляет собой цикл `for` без частей инициализации и обновления переменной цикла, при этом у него имеется только условие продолжения цикла и тело:

**while (условие продолжения цикла)**  
**тело цикла**

Прежде всего программа проверяет *условие продолжения цикла*. Если в результате вычисления этого выражения получается логическое значение *true*, тогда программа выполняет операторы в теле цикла. Подобно циклу *for*, тело данного цикла состоит из одного оператора или блока, определяемого парой фигурных скобок. После завершения выполнения тела цикла программа возвращается к *условию продолжения цикла* и вычисляет его еще раз. Если данное условие оказывается ненулевым, тогда программа снова выполняет тело цикла. Подобный цикл проверки и выполнения продолжается до тех пор, пока в результате вычисления выражения не получится логическое значение *false* (рис. 5.3). Ясно, что, если требуется, чтобы цикл в конечном итоге был завершен, в теле цикла должно быть выполнено некоторое действие, которое могло бы оказать влияние на значение *условия продолжения цикла*. Например, в цикле может быть выполнено приращение переменной, используемой в *условии продолжения цикла*, либо прочитано введенное с клавиатуры новое значение. Подобно циклу *for*, *while* является циклом с входным *условием продолжения цикла*. Таким образом, если в результате вычисления *условия продолжения цикла* оказывается ложным с самого начала, программа вообще не выполняет тело цикла.

В листинге 5.12 цикл *while* приводится в действие. В данном цикле осуществляется циклическое обращение к каждому символу строки и отображение кода ASCII этого символа. Цикл завершается по достижении пустого символа. Такой способ пошагового обращения к символам строки вплоть до пустого символа является

типичным для обработки строк в C++. Благодаря тому что строка содержит собственный маркер окончания, программам зачастую не требуется явная информация о длине строки.

### Листинг 5.12 Программа while.cpp.

```
#include <iostream>
using namespace std;
const int ArSize = 20;
int main()
{
 char name[ArSize];

 cout << "Your first name, please: ";
 cin >> name;
 cout << "Here is your name,
 verticalized and ASCIIized:\n";
 int i = 0; //приступить к обработке с
 //начала строки и вплоть до ее конца
 while (name[i] != '\0') //программа,
 //представляющая возможности цикла while
 {
 cout << name[i] << ": " << int(name[i])
 << '\n';
 i++; // не забывайте об этом действии
 }
 return 0;
}
```

Ниже приведен результат выполнения указанной выше программы:

```
Your first name, please: Muffy
Here is your name, verticalized and
ASCIIized:
M: 77
u: 117
f: 102
f: 102
y: 121
```

(Безусловно, расположенные вертикально слова в коде ASCII нельзя считать настоящими словами или даже претендующими на полезность словами. Однако они придают интересный вид выходному результату.)

### Примечания к программе

Условие продолжения цикла *while* выглядит следующим образом:

```
while (name[i] != '\0')
```

Это условие позволяет проверить, является ли конкретный символ в массиве пустым. Для успешного выполнения такой проверки в теле цикла необходимо изменить значение переменной *i*. Вследствие пропуска этого шага цикл постоянно привязывается к одному и тому же элементу массива, выводя символ и его код до тех пор, пока не будет остановлено выполнение программы. Подобный бесконечный цикл является одной из наиболее распространенных ошибок применения циклов. Она зачастую возникает в том случае, когда забывают обновить операторы из тела цикла.

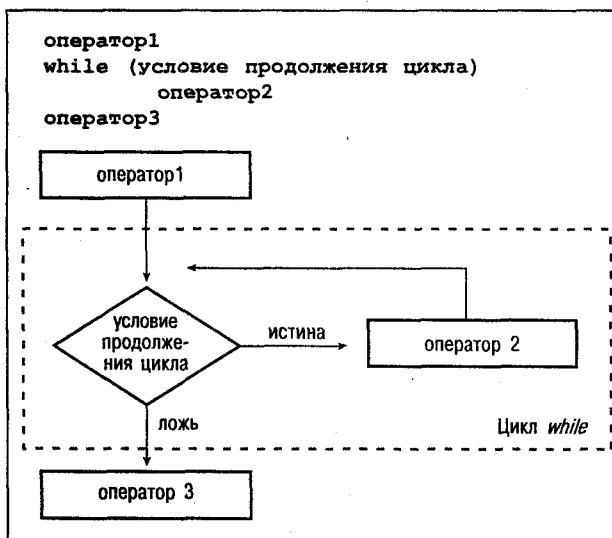


РИСУНОК 5.3 Цикл *while*.

Конструкцию **while** можно переписать следующим образом:

```
while (name[i])
```

При таком изменении программа действует как и прежде. Это связано с тем что если элемент массива `name[i]` содержит обычный символ, тогда значение его кода не равно 0, т.е. является истинным. Однако в том случае, когда элемент массива `name[i]` содержит пустой символ, значение его кода равно 0, т.е. является ложным. Подобная запись оказывается более короткой и в то же время менее ясной, чем примененная ранее. Компиляторы с неразвитыми логическими возможностями могут создать более быстрый код для второго варианта цикла, а вот компиляторы с развитыми логическими возможностями в обоих случаях создадут один и тот же код.

Для вывода кода ASCII каждого символа в рассматриваемой программе выполняется преобразование типа символа, который содержится в элементе `name[i]`, в целочисленный тип. Затем оператор `cout` выполняет вывод результата в виде целого значения, а не интерпретирует его как символьный код.

## Сравнение циклов **for** и **while**

В C++ циклы **for** и **while**, по существу, равнозначны. Например, цикл **for**:

```
for (инициализация цикла; условие продолжения цикла; обновление переменной цикла)
{
 оператор (ы)
}
```

может быть переписан следующим образом:

```
инициализация цикла;
while (условие продолжения цикла)
{
 оператор (ы)
 обновление переменной цикла;
}
```

Аналогично цикл **while**:

```
while (условие продолжения цикла)
 тело цикла
```

может быть переписан следующим образом:

```
for (; условие продолжения цикла;)
 тело цикла
```

Для цикла **for** требуется три выражения (или, выражаясь более точным техническим языком, один оператор с двумя последующими выражениями), однако эти выражения (или операторы) могут быть пустыми. В этой связи пропуск **условия продолжения** в цикле **for** истолковывается как истинное значение, и поэтому цикл

```
for (; ;)
 тело цикла
```

выполняется бесконечно.

Благодаря тому что циклы **for** и **while** практически равнозначны, их применение является делом стиля. (Тем не менее, между ними существует некоторое отличие, если в тело цикла входит оператор **continue**, который рассматривается в главе 6. Как правило, программисты применяют цикл **for** в качестве цикла со счетчиком, поскольку формат цикла **for** позволяет размещать в одном месте все необходимые вещи: начальное значение, конечное значение и способ обновления переменной цикла (счетчика цикла). А цикл **while** чаще всего применяется в том случае, когда заранее точно неизвестно, сколько раз должен быть выполнен цикл.

При разработке цикла необходимо руководствоваться следующими принципами:

1. Определить условие, которое завершает выполнение цикла.
2. Инициализировать переменную цикла до выполнения первой проверки.
3. Обновлять переменную цикла на каждой итерации цикла перед повторной проверкой условия.

Одна из замечательных особенностей цикла **for** состоит в том, что его структура дает возможность реализовать все три указанных выше руководящих принципа, что способствует их практическому усвоению.

### НЕВЕРНАЯ РАССТАНОВКА ЗНАКОВ ПРЕПИНАНИЯ

Тела обоих циклов, **for** и **while**, состоят из одного оператора. Как известно, в качестве оператора может выступать блок, содержащий несколько операторов. При этом следует иметь в виду, что границы блока определяются с помощью фигурных скобок, а не отступа. Рассмотрим, например, следующий цикл:

```
i = 0;
while (name[i] != '\0')
 cout << name[i] << "\n";
 i++;
cout << "Done\n";
```

Отступ указывает на то, что автор программы предполагал отнести оператор `i++` к телу цикла. Однако отсутствие фигурных скобок указывает компилятору на то, что тело цикла состоит только из первого оператора `cout`. Таким образом, цикл продолжает бесконечную распечатку первого символа в массиве. При этом программа вообще не доходит до оператора `i++`, поскольку он находится за пределами цикла.

В следующем примере продемонстрирована еще одна, возможная ловушка:

```
i = 0;
while (name[i] != '\0'); //ошибка,
//связанная с применением точки с запятой
{
 cout << name[i] << "\n";
 i++;
}
cout << "Done\n";
```

На сей раз фигурные скобки расположены в коде верно, однако в нем присутствует лишняя точка с запятой. Напомним, что точка с запятой завершает оператор, поэтому данная точка с запятой завершает цикл `while`. Иными словами, тело данного цикла составляет пустой (*null*) оператор, т.е. после точки с запятой как бы ничего не следует. При этом весь код в фигурных скобках оказывается расположенным после цикла. И доступ к нему вообще невозможен. Вместо этого получается бесконечный цикл, в котором ничего не выполняется. В связи с этим следует избегать подобного беспорядочного применения точки с запятой.

## Небольшая пауза

Иногда оказывается полезным вставить в программу временную задержку. Например, читателю, возможно, встречались программы, которые выводят сообщение на экран, а затем переходят к выполнению следующих операций, прежде чем это сообщение можно будет прочитать. При этом остается опасение, что была пропущена некоторая невосстановимая и весьма важная информация. Было бы намного лучше, если бы программа делала паузу в течение пяти секунд, прежде чем продолжать свою работу дальше. Для достижения подобного результата весьма удобным оказывается цикл `while`. Один из самых старых методов отсчета времени на компьютере состоял в следующем:

```
long wait = 0;
while (wait < 10000)
 wait++; // "молчаливый" отсчет
```

Недостаток данного метода состоит в том, что при изменении быстродействия компьютера приходится изменять пределы отсчета. Например, ряд игр, написанных для первоначальной модели ПК IBM PC, стали действовать невообразимо быстро при выполнении на последующих более быстродействующих моделях ПК. Более совершенный подход состоит в применении для отсчета времени системных часов.

Для этой цели в библиотеках ANSI C и C++ имеется специальная функция. Эта функция называется `clock()` и возвращает системное время, прошедшее с того момента, когда программа начала выполняться. Применение этой функции сопряжено с двумя сложностями. Во-первых, функция `clock()` отнюдь не обязательно возвращает время в секундах. А во-вторых, возвращаемым данной функцией типом данных может быть `long` в одних системах, `unsigned long` в других системах, а возможно, и какой-либо иной тип данных.

Однако в заголовочном файле `ctime` (который в более старых реализациях называется `time.h`) эта проблема решена. Во-первых, здесь определена символическая константа `CLOCK_PER_SEC`, значение которой равно числу единиц системного времени в секунду. Таким

образом, разделив системное время на указанное значение, можно получить количество секунд. С другой стороны, для получения значения времени в системных единицах можно умножить число секунд на значение `CLOCK_PER_SEC`. Во-вторых, в файле `ctime` в качестве псевдонима возвращаемого функцией `clock()` типа устанавливается `clock_t`. (Более подробные сведения об этом приведены в разделе "Псевдонимы типов"). Это означает возможность объявить переменную типа `clock_t`, и тогда компилятор преобразует ее в тип `long` или `unsigned int` либо в другой подходящий для конкретной системы тип данных.

 **ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**

В тех системах, в которых отсутствует заголовочный файл `ctime`, вместо этого может быть использован файл `time.h`. В некоторых реализациях C++ могут возникнуть затруднения с выполнением приведенной ниже программы `waiting.cpp`, если используемый в данной реализации компонент библиотеки не является полностью совместимым со стандартом ANSI C. Это связано с тем, что функция `clock()` служит в качестве дополнения традиционной библиотеки C. Кроме того, в ряде первоначальных реализаций ANSI C вместо более длинной константы `CLOCK_PER_SEC` использовались константы `CLK_TCK` или `TCK_CLK`. А в некоторых более поздних реализациях C++ ни одна из этих определенных ранее констант не распознается. В ряде сред (в частности, в MSVC++ 1.0, но не в MSVC++ 5.0) возникают затруднения с применением предупреждающего символа `\a`, а также с согласованием режима отображения и временной задержки.

В листинге 5.13 показано, каким образом функция `clock()` и заголовочный файл `ctime` применяются для создания цикла временной задержки.

### Листинг 5.13 Программа `waiting.cpp`.

```
// программа, демонстрирующая применение
// функции clock() в цикле временной задержки
#include <iostream>
#include <ctime> // определяет функцию clock()
 // и тип clock_t
using namespace std;
int main()
{
 cout << "Enter the delay time, in seconds: ";
 float secs; cin >> secs;

 // преобразовать в такты системных часов
 clock_t delay = secs * CLOCK_PER_SEC;
 cout << "starting\n";
 clock_t start = clock();

 // ожидать до тех пор, пока не пройдет время
 while (clock() - start < delay)
 ; // обратить внимание на
 // применение точки с запятой
 cout << "done\n";
 return 0;
}
```

Благодаря тому что данная программа выполняет расчет времени в системных единицах, а не в секундах, отпадает необходимость в преобразовании системного времени в секунды на каждой итерации цикла.

### ПСЕВДОНИМЫ ТИПОВ

В C++ имеется два способа установки нового имени в качестве псевдонима типа. Один из них состоит в применении препроцессора:

```
#define BYTE char // препроцессор заменяет
 // тип BYTE на тип char
```

При этом препроцессор заменяет все экземпляры типа **BYTE** на тип **char** во время компиляции программы, превращая тем самым **BYTE** в псевдоним типа **char**.

Второй способ состоит в применении ключевого слова **typedef** языка C++ (а также С) для создания псевдонима. Например, чтобы превратить **byte** в псевдоним типа **char**, необходимо выполнить следующий оператор:

```
typedef char byte; // превращает byte в
 // псевдоним типа char
```

Ниже приведена общая форма указанного преобразования:

```
typedef typeName aliasName
```

Иными словами, если требуется превратить **aliasName** в псевдоним конкретного типа, тогда следует объявить **aliasName** в качестве переменной этого типа, а затем предварить данное объявление ключевым словом **typedef**. Например, чтобы превратить **byte\_pointer** в псевдоним указателя типа **char**, следует объявить **byte\_pointer** в качестве указателя типа **char**, а затем предварить это объявление ключевым словом **typedef**:

```
typedef char * byte_pointer; // указатель
 // типа char
```

Нечто подобное можно попытаться сделать и с помощью ключевого слова **#define**, однако этот способ не годится для объявления списка переменных. Например, рассмотрим следующий фрагмент кода:

```
#define FLOAT_POINTER float *
FLOAT_POINTER pa, pb;
```

В результате подстановки, выполняемой препроцессором, данное объявление будет преобразовано в следующее:

```
float * pa, pb; // pa является указателем
 // типа float, а pb -- просто типом float
```

При использовании **typedef** подобного затруднения не возникает.

Следует заметить, что **typedef** не создает новый тип данных. При этом создается лишь новое имя старого типа данных. Если превратить **word** в псевдоним типа **int**, тогда метод **cout** будет действительно рассматривать значение типа **word** в качестве значения типа **int**.

## Цикл do while

До сих пор были рассмотрены циклы **for** и **while**. Третьим типом цикла в C++ является цикл **do while**.

От двух предыдущих циклов он отличается тем, что это цикл с *постусловием*. Это означает, что сначала выполняется тело цикла и только после этого оценивается условие продолжения цикла. Если в результате оценки получается логическое значение **false**, тогда данный цикл завершается. В противном случае начинается новый этап выполнения и проверки цикла. Подобный цикл всегда выполняется, по крайней мере, один раз, поскольку процесс выполнения программы должен пройти тело цикла, прежде чем будет осуществлена проверка условия продолжения. Ниже приведен синтаксис рассматриваемого цикла:

```
do
 тело цикла
 while (условие продолжения цикла);
```

Тело рассматриваемого цикла может состоять из одного оператора или ограниченного фигурными скобками блока операторов. Процесс выполнения программы в цикле **do while** приведен на рис. 5.4.

Как правило, лучше выбирать цикл с предусловием, чем с постусловием, поскольку в первом случае проверка осуществляется до выполнения цикла. Допустим, например, что в листинге 5.12 вместо цикла **while** использован цикл **do while**. Тогда в данном цикле пустой символ и его код выводился бы до обнаружения конца строки. Однако иногда цикл типа **do while** все же имеет

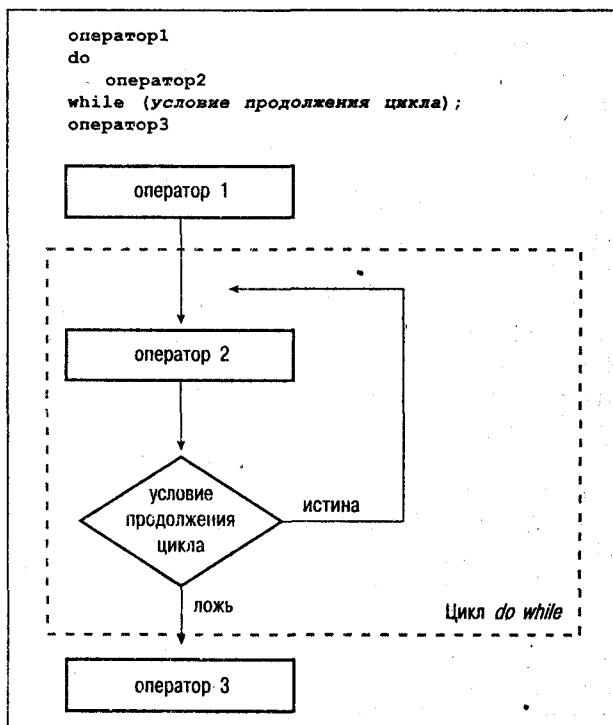


РИСУНОК 5.4 Цикл **do while**.

смысл. Например, если выполняется запрос пользовательского ввода, программа должна получить ввод до его проверки. В листинге 5.14 показано, каким образом цикл **do while** используется в подобной ситуации.

#### Листинг 5.14 Программа dowhile.cpp.

```
// dowhile.cpp -- программа,
// демонстрирующая цикл с постусловием
#include <iostream>
using namespace std;
int main()
{
 int n;
 cout << "Enter numbers in the range 1-10
 to find ";
 cout << "my favorite number\n";
 do
 {
 cin >> n; // выполнить тело цикла
 } while (n != 7); //затем осуществить проверку
 cout << "Yes, 7 is my favorite.\n" ;
 return 0;
}
```

Ниже приведен результат выполнения указанной выше программы:

```
Enter numbers in the range 1-10 to find my
favorite number
9
4
7
Yes, 7 is my favorite.
```

## Циклы и ввод текста

После того как стал понятен принцип действия циклов, обратимся к одной из наиболее важных и распространенных задач, для которых предназначены циклы: к посимвольному чтению текста из файла либо с клавиатуры. Допустим, например, что требуется написать программу, которая подсчитывает число введенных символов, строк и слов. По традиции в C++, как, собственно, и в C, для решения подобного рода задачи применяется цикл **do while**. Рассмотрим, каким образом это делается. Тем, кто знает C, не рекомендуется сразу же пропускать эту часть данной главы. Несмотря на то что цикл **do while** в C++ оказывается таким же, как и в C, возможности ввода вывода в C++ несколько иные. А это придает циклу в C++ несколько иной вид. В самом деле, объект **cin** поддерживает три разных режима ввода одинарных символов, причем в каждом случае с разным форматом ввода/вывода. Рассмотрим, каким образом все эти возможности могут быть применены в циклах **while**.

## Применение простого оператора **cin** для ввода данных

Если в программе предполагается использовать цикл для ввода текста с клавиатуры, то в этом случае необходимо знать, когда этот ввод должен быть прекращен? А каким образом можно узнать об этом? В качестве признака окончания ввода можно, в частности, воспользоваться специальным символом *сигнальной метки*. Например, в листинге 5.15 чтение потока ввода прекращается в том случае, когда в программе встречается символ **#**. Приведенная в данном листинге программа подсчитывает число прочитанных символов и отображает их. Это означает, что она повторно отображает прочитанные символы. (Ведь одного лишь нажатия клавиши на клавиатуре недостаточно для вывода символа на экран. Для этого программам приходится выполнять монотонную работу, связанную с отображением введенного символа. Как правило, эту работу берет на себя операционная система. В данном же случае воспроизведение ввода выполняется с помощью операционной системы, а также тестовой программы, приведенной в листинге 5.15.) По завершении ввода эта программа выдает отчет о числе обработанных символов.

#### Листинг 5.15 Программа textin1.cpp.

```
// textin1.cpp -- программа, считающая
// символы с помощью цикла while
#include <iostream>
using namespace std;
int main()
{
 char ch;
 int count = 0; // использование
 // основного ввода

 cin >> ch; // получение символа
 while (ch != '#') // проверка символа
 {
 cout << ch; // отображение символа
 count++; // подсчет символов
 cin >> ch; // получение следующего
 // символа
 }
 cout << "\n" << count
 << " characters read\n";
 return 0;
}
```

Ниже приведен результат выполнения указанной выше программы:

```
see ken run#really fast
saeikenrun
9 characters read
```

Очевидно, пользователь Кен так спешит ввести текст, что даже забывает о пробелах во время ввода.

## Примечания к программе

Прежде всего, следует обратить внимание на структуру данной программы. Эта программа читает первый введенный символ еще до завершения цикла. Таким образом, первый символ может быть проверен по достижении оператора цикла. Это имеет большое значение, поскольку первым символом может быть символ #. Благодаря тому что в программе `textin1.cpp` применяется цикл с входным условием, рассматриваемая программа правильно пропускает в данном случае весь цикл. А поскольку предварительно было установлено нулевое значение переменной `count`, ее значение оказывается верным.

Допустим, что первый прочитанный символ не является знаком #. Тогда рассматриваемая программа входит в цикл, отображает символ, увеличивает счетчик и читает следующий символ. Последний шаг является очень важным. Без него в цикле бесконечно повторялась бы обработка первого введенного символа. А с его помощью программа переходит к следующему символу.

Следует заметить, что конструкция данного цикла следует упомянутым выше руководящим принципам. Условие, по которому цикл завершается, выполняется в том случае, если прочитан символ #. Это условие инициализируется благодаря чтению символа еще до начала цикла. Затем это условие обновляется благодаря чтению нового символа в конце цикла.

Все это кажется благоразумным. Так почему же рассматриваемая программа исключает пробелы при выводе? Всему виной метод `cin`. При чтении значений типа `char`, как, собственно, и при чтении значений других основных типов, метод `cin` пропускает символы пробела и начала новой строки. При этом пробелы во вводимом тексте не отображаются, и поэтому они и не подсчитываются.

Дело усложняется еще и тем, что оператор `cin` буферизует ввод. А это означает, что введенные символы не передаются программе до тех пор, пока не будет нажата клавиша Enter. Именно поэтому существует возможность вводить символы после появления знака #. После нажатия клавиши Enter вся последовательность символов передается программе, однако она прекращает обработку ввода по достижении символа #.

## На помощь приходит функция `cin.get(char)`

Как правило, программам, которые выполняют посимвольное чтение вводимого текста, приходится проверять буквально каждый символ, в том числе пробелы, символы табуляции и новой строки. Класс `istream`, который определяется в заголовочном файле `iostream` и к которому принадлежит метод `cin`, включает в себя компо-

нентные функции, удовлетворяющие указанным потребностям. В частности, функция-элемент `cin.get(ch)` читает из входного потока следующий символ, даже если им является пробел, а затем присваивает его переменной `ch`. Заменив выражение `cin>>ch` указанной функцией, можно исправить ошибку в листинге 5.15. Полученный результат приведен в листинге 5.16.

### Листинг 5.16 Программа `textin2.cpp`.

```
#include <iostream>
using namespace std;
int main()
{
 char ch;
 int count = 0;

 cin.get(ch); // применить функцию
 // cin.get(ch)
 while (ch != '#')
 {
 cout << ch;
 count++;
 cin.get(ch); // применить ее еще раз
 }
 cout << "\n" << count
 << " characters read\n";
 return 0;
}
```

Ниже приведен результат выполнения указанной программы:

```
Did you use a #2 pencil?
Did you use a
14 characters read
```

Теперь программа отображает и подсчитывает каждый символ, в том числе и пробелы. Тем не менее, ввод буферизуется, поэтому все еще возможен ввод дополнительных символов по сравнению с тем их числом, которого в конечном итоге достигает программа.

Тем, кто знаком с C, указанная программа может показаться совершенно неверной! Ведь вызов функции `cin.get(ch)` заменяет значение переменной `ch`, а это значит, что изменяется значение данной переменной. Если в C требуется изменить значение переменной, то для этого необходимо передать функции адрес данной переменной. Однако при вызове функции `cin.get()` в листинге 5.16 передается сама переменная `ch`, а не ее адрес `&ch`. Такой программный код в C выполняться не будет. А вот в C++ он будет работать при условии, что аргумент функции объявлен в качестве *ссылки*. Это новый для C++ производный тип. В заголовочном файле `iostream` аргумент функции `cin.get(ch)` объявлен в качестве ссылочного типа, поэтому данная функция может изменить значение своего аргумента. Подробнее об этом будет сказано в главе 8. А между тем, знатоков С может успо-

коить тот факт, что передача аргументов в C++ осуществляется таким же образом, как и в С. Однако это не относится к функции `cin.get(ch)`.

## Выбор функции `cin.get()`

В главе 4 используется следующий код:

```
char name[ArSize];
...
cout << "Enter your name:\n";
cin.get(name, ArSize).get();
```

Последняя строка равнозначна двум последовательно выполняемым вызовам следующих функций:

```
cin.get(name, ArSize);
cin.get();
```

В одном варианте функция `cin.get()` воспринимает два аргумента: имя массива, которое является адресом строки (с технической точки зрения это тип `char*`), а также целое значение `ArSize` типа `int`. (Напомним, что имя массива представляет собой адрес первого его элемента, поэтому имя символьного массива оказывается типа `char*`.) С другой стороны, функция `cin.get()` применяется в программе без аргументов. А чаще всего она применяется следующим образом:

```
char ch;
cin.get(ch);
```

На сей раз у функции `cin.get()` имеется один аргумент типа `char`. Если в С функция воспринимает в качестве аргументов указатель типа `char` и значение типа `int`, то эту функцию нельзя с таким же успехом использовать с одним аргументом другого типа. А вот в C++ это вполне возможно, поскольку данный язык поддерживает свойство ООП, называемое *перегрузкой функции*. Перегрузка функции дает возможность создавать другие функции с тем же самым именем при условии, что у них имеется другой список аргументов. Например, если в C++ применяется функция `cin.get(name, ArSize)`, тогда компилятор обнаружит тот вариант функции `cin.get()`, в котором используются аргументы `char` и `int`. Однако если применяется функция `cin.get(ch)`, тогда компилятор выбирает тот вариант функции, в котором используется единственный аргумент типа `char`. А если в программе аргументы данной функции не предоставляются, тогда компилятор использует ее вариант `cin.get()` без аргументов. Перегрузка функции позволяет использовать одно и то же имя для связанных друг с другом функций, которые выполняют одну и ту же основную задачу разными способами или для разных типов данных. Этот вопрос также ожидает своего рассмотрения в главе 8. А между тем, освоить перегрузку функций можно, воспользовавшись примерами, которые сопутствуют классу `istream`. Чтобы провести различие между разными

вариантами функций, при обращении к ним будет включен список их аргументов. Таким образом, `cin.get()` означает вариант данной функции без аргументов, а `cin.get(char)` означает ее вариант с одним аргументом.

## Условие конца файла

Как следует из листинга 5.16, применение такого символа, как #, для обозначения конца ввода отнюдь не всегда дает удовлетворительный результат, поскольку такой символ вполне обоснованно может быть частью вводимых данных. Это же справедливо и для других произвольно выбранных знаков, в частности, для символов @ или %. Если ввод осуществляется из файла, то в этом случае можно воспользоваться намного более эффективным методом обнаружения конца файла (EOF). Средства ввода в C++ взаимодействуют с операционной системой для обнаружения момента достижения конца файла и выдачи этой информации программе.

На первый взгляд кажется, что чтение информации из файлов не имеет отношения к методу `cin` и вводу с клавиатуры, однако в двух аспектах такое отношение все же существует. Во-первых, многие операционные системы, в том числе UNIX и MS-DOS, поддерживают *переадресацию*, которая дает возможность заменить файлом ввод с клавиатуры. Допустим, например, что в MS-DOS имеется исполняемая программа под названием `gofish.exe`, а также текстовый файл под названием `fishtale`. Тогда в командной строке DOS можно ввести следующую команду:

```
gofish < fishtale
```

Эта команда приводит к тому, что ввод из файла осуществляется не с клавиатуры, а с помощью указанной программы. Знак < является оператором переадресации как в UNIX, так и в MS-DOS. Кроме того, многие операционные системы позволяют имитировать с клавиатуры условие конца файла. Так, в UNIX для этого следует использовать комбинацию клавиш `Ctrl+D` в начале строки. А в DOS для этого нужно сначала нажать клавиши `Ctrl+Z`, а затем `Enter` в любом месте строки. В некоторых реализациях поддерживается аналогичное поведение, хотя это не относится к базовой операционной системе. Принцип указания конца файла при вводе с клавиатуры на самом деле считается устаревшим и относится к средам, работающим в режиме командной строки. Тем не менее, в среде Symantec C++ для Macintosh имитируется вариант для UNIX, и поэтому комбинация клавиш `Ctrl+D` распознается в качестве имитируемого признака EOF. Среда Metrowerks Codewarrior распознает комбинацию клавиш `Ctrl+Z` в средах Macintosh и Windows. А в средах Microsoft Visual C++ 5.0 и Borland C++Builder для Windows поддержи-

вается консольный режим, в котором комбинация клавиш Ctrl+Z действует без клавиши Enter. Как ни странно, после обнаружения комбинации клавиш Ctrl+Z обе последние среды вообще не способны отображать вывод до отображения первого символа новой строки.

Если программа способна проверять наличие конца файла, тогда ее можно использовать вместе с переадресованными файлами, а также при вводе с клавиатуры, при котором имитируется конец файла. Последнее может оказаться полезным, поэтому посмотрим, как это делается.

Когда метод `cin` обнаруживает конец файла (EOF), то в этом случае он устанавливает значение 1 для двух битов (`eofbit` (бит конца файла) и `failbit` (бит отказа)). Для проверки установки бита `eofbit` можно воспользоваться функцией-элементом `eof()`. В результате вызова функции `cin.eof()` возвращается значение `true` типа `bool`, если обнаружен признак EOF, в противном же случае возвращается значение `false`. Аналогично функция-элемент `fail()` возвращает значение `true`, если установлено значение 1 для бита `eofbit` или бита `failbit`, в противном же случае возвращается значение `false`. Следует заметить, что оба метода, `eof()` и `fail()`, сообщают о самой последней попытке чтения, т.е. они сообщают о последствиях, а не действуют с упреждением. Таким образом, проверка с помощью функций `cin.eof()` и `cin.fail()` всегда должна следовать после попытки чтения. Этот факт отражен в конструкции, приведенной в листинге 5.17. Здесь вместо функции `eof()` применяется функция `fail()`, поскольку первая функция оказывается работоспособной в большем числе реализаций.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В одних системах имитация признака EOF с клавиатуры не поддерживается. А в других системах, в том числе в Microsoft Visual C++ 5.0, Metrowerks Codewarrior и Borland C++Builder, такая поддержка оказывается неидеальной. Если функция `cin.get()` применялась раньше для "замораживания" экрана до тех пор, пока не появится возможность ввода, то в данном случае она работать не будет, поскольку в результате обнаружения признака EOF исключаются дальнейшие попытки чтения ввода. Однако для того, чтобы сохранить экран видимым на некоторое время, можно воспользоваться циклом задержки времени, подобным приведенному в листинге 5.13.

### Листинг 5.17 Программа `textin3.cpp`

```
// textin3.cpp -- программа чтения символов
// до конца файла
#include <iostream>
using namespace std;
int main()
{
 char ch;
 int count = 0;
 cin.get(ch); // попытка чтения символа
```

```
while (cin.fail() == false) // проверка
 // на наличие признака EOF
{
 cout << ch; // отображение символа
 count++;
 cin.get(ch); // попытка чтения еще
 // одного символа
}
cout << "\n" << count
 << " characters read\n";
return 0;
```

Ниже приведен результат выполнения указанной выше программы. В связи с тем что эта программа была выполнена в системе Windows 95, для имитации конца файла была использована комбинация клавиш Ctrl+Z. Пользователи DOS должны для этого нажать клавиши Ctrl+Z, Enter, а пользователи UNIX и Symantec C++ для Macintosh — клавиши Ctrl+D.

```
The green bird sings in the winter.<ENTER>
The green bird sings in the winter.
Yes, but the crow flies in the dawn.<ENTER>
Yes, but the crow flies in the dawn.
<CTRL><Z>
73 characters read
```

Благодаря переадресации указанную программу можно использовать для отображения текстового файла и выдачи отчета о числе символов в этом файле. На сей раз получилась программа, которая читает, отображает и подсчитывает содержимое файла, состоящего из двух строк, в системе UNIX (знак \$ служит в UNIX в качестве приглашения):

```
$ textin3 < stuff
I am a UNIX file. I am proud
to be a UNIX file.
49 characters read
$
```

### Конец файла означает конец ввода

Напомним, что, когда метод `cin` обнаруживает конец файла, он устанавливает соответствующий признак в объекте `cin`, указывающий на условие окончания файла. При установке этого признака метод `cin` больше не выполняет чтение ввода, и поэтому дальнейшие вызовы метода `cin` не оказывают никакого влияния на суть происходящего. В этом есть определенный смысл при вводе файлов, поскольку чтение не должно происходить после достижения конца файла. Однако при вводе с клавиатуры для прекращения выполнения цикла, возможно, придется воспользоваться симулированным концом файла, но затем потребуется осуществить дополнительное чтение ввода. Метод `cin.clear()` сбрасывает признак конца файла и, таким образом, позволяет продолжить ввод. Рассмотрение этого вопроса продол-

жается в главе 16. Тем не менее, следует иметь в виду, что в Windows 95 использование комбинации клавиш Ctrl+Z в режиме имитации консоли на самом деле приводит к прекращению как ввода, так и вывода, и метод `cin.clear()` не в состоянии возобновить и тот, и другой процесс.

### Распространенные идиомы

По существу, конструкция цикла ввода имеет следующий вид:

```
cin.get(ch); // попытка чтения символа
while (cin.fail() == false) // проверка на
 // наличие признака EOF
{
 ...
 cin.get(ch); // выполнить что-либо
 // попытаться прочитать
 // еще один символ
}
```

Здесь возможны некоторые сокращения программного кода. Так, в главе 6 представлен оператор `!`, который осуществляет переключение между логическими значениями `true` и `false`. Им можно воспользоваться для изменения проверки в цикле `while` следующим образом:

```
while (!cin.fail()) // до тех пор, пока не
 // произойдет сбой при вводе
```

Метод `cin.get(char)` возвращает значение, которое соответствует объекту `cin`. Однако в классе `iostream` имеется функция, которая может выполнять преобразование объекта типа `iostream`, в частности объекта `cin`, в значение `bool`. Эта функция преобразования вызывается при появлении объекта `cin` в том месте, где предполагается значение `bool`, в частности, в условии продолжения цикла `while`. К тому же значение `bool` равно `true`, если последняя попытка чтения оказалась успешной, в противном же случае оно равно `false`. Это означает, что проверку в цикле `while` можно изменить следующим образом:

```
while (cin) // до тех пор, пока ввод
 // осуществляется успешно
```

Это несколько более распространенный вариант, чем применение выражений `!cin.fail()` и `!cin.eof()`, так как в данном случае обнаруживаются и другие сбои, в частности, отказ жесткого диска.

И наконец, в связи с тем, что метод `cin.get(char)` возвращает значение, соответствующее объекту `cin`, цикл ввода можно сократить до следующего формата:

```
while (cin.get(ch)) //до тех пор, пока ввод
 //осуществляется успешно
{
 ...
 // выполнить что-либо
}
```

Для оценки условия продолжения цикла программа сначала должна осуществить вызов метода `cin.get(char)`,

который в случае успешного выполнения присваивает введенное значение переменной `ch`. Затем программа получает возвращаемое из данной функции значение, которое соответствует объекту `cin`. После этого она выполняет по отношению к объекту `cin` преобразование типа `bool`, в результате которого получается значение `true`, если ввод оказался успешным, а противном случае получается значение `false`. При этом все три упомянутых выше руководящих принципа (обозначение условия прекращения цикла, инициализация и обновление переменной цикла) сводятся к одному условию продолжения цикла.

### Еще одна разновидность функции `cin.get()`

Некоторых пользователей С может одолевать тоска по функциям ввода/вывода `getchar()` и `putchar()`. Не беспокойтесь, они по-прежнему доступны. Для их применения достаточно воспользоваться заголовочным файлом `stdio.h` таким же образом, как и в С, либо текущим его вариантом `cstudio`. С другой стороны, можно применить функции-элементы из классов `istream` и `ostream`, которые действуют аналогичным образом. Последний метод будет рассмотрен ниже.

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторых более старых реализациях рассматриваемая здесь функция-элемент `cin.get()` (без аргументов) не поддерживается.

Функция-элемент `cin.get()` без аргументов возвращает следующий введенный символ. Она применяется следующим образом:

```
ch = cin.get();
```

(Напомним, что функция `cin.get(ch)` возвращает объект, а не прочитанный символ.) Данная функция действует подобно имеющейся в С функции `getchar()` и возвращает код символа в виде значения типа `int`. Аналогично функция `cout.put()` (рассмотренная в главе 3) может быть использована для отображения символа:

```
cout.put(ch);
```

Она действует подобно имеющейся в С функции `putchar()`, за исключением того, что ее аргумент должен быть типа `char`, а не типа `int`.

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Первоначально у функции-элемента `put()` был один прототип `put(char)`. Ей можно было передать аргумент типа `int`, который далее мог иметь тип, приводимый к типу `int`. Стандарт также требует наличия единственного прототипа. Однако во многих текущих реализациях предоставляются следующие три прототипа данной функции: `put(char)`, `put(signed char)` и `put(unsigned char)`. При использовании в подобных реализациях функции `put()` с аргументом типа `int`

формируется сообщение об ошибке, поскольку в данном случае имеется несколько вариантов преобразования типа `int`. К данным типу `int` применимо и явное приведение типа, в частности, `cin.put(char(c))`.

Для успешного применения функции `cin.get()` необходимо знать, каким образом она обрабатывает условие конца файла. Когда эта функция достигает конца файла, она не возвращает никаких символов. Напротив, функция `cin.get()` возвращает специальное значение, представленное в виде символьской константы `EOF`. Эта константа определяется в заголовочном файле `iostream`. Значение `EOF` должно отличаться от любого действительного символа, чтобы программа не спутала его с обычным символом. Как правило, значение `EOF` определяется равным `-1`, поскольку ни у одного из символов нет значения `-1` в коде ASCII, тем не менее, знать конкретное значение `EOF` отнюдь не обязательно. В программе достаточно использовать константу `EOF`. Например, основная часть листинга 5.15 имеет следующий вид:

```
cin >> ch;
while (ch != '#')
{
 cout << ch;
 count++;
 cin >> ch;
}
```

В данном случае можно осуществить замену `cin` на `cin.get()`, `cout` на `cout.put()` и '#' на `EOF`:

```
ch = cin.get();
while (ch != EOF)
{
 cout.put(ch); // применение функции
 // cout.put(char)
 // в ряде реализаций
 count++;
 ch = cin.get();
}
```

Если значение переменной `ch` является символом, тогда этот символ отображается в цикле. А если им является признак `EOF`, тогда цикл завершается.

### СОВЕТ

Следует понять, что признак `EOF` не представляет ни один из вводимых символов. Напротив, он сигнализирует об отсутствии дополнительных символов.

Существует весьма тонкий и в то же время важный момент, связанный с применением функции `cin.get()`, помимо внесенных до сих пор изменений. В связи с тем что константа `EOF` представляет значение, которое находится за пределами действительных кодов символов, вполне вероятно, что она может оказаться несовместимой с типом `char`. Например, в некоторых системах тип `char` оказывается без знака, поэтому переменная типа

`char` вообще не может иметь значение `EOF`, равное `-1`. Именно по этой причине при использовании функции `cin.get()` (без аргументов) и проверке значения `EOF` возвращаемому значению необходимо присвоить тип `int` вместо типа `char`. Кроме того, если назначить переменную `ch` типа `int` вместо типа `char`, тогда при отображении переменной `ch` придется выполнить приведение к типу `char`.

В листинге 5.18 содержится новый вариант приведенной в листинге 5.15 программы, в которой применяется функция `cin.get()`. Кроме того, код в данном случае сокращен благодаря объединению ввода символов с проверкой цикла `while`.

### Листинг 5.18 Программа `textin4.cpp`.

```
// textin4.cpp - программа чтения символов
// с помощью функции cin.get()
#include <iostream.h>
int main(void)
{
 int ch; // переменная должна иметь тип
 // int, а не char
 int count = 0;

 while ((ch = cin.get()) != EOF)
 // проверка на наличие конца файла
 {
 cout.put(char(ch));
 count++;
 }
 cout << "\n" << count
 << " characters read\n";
 return 0;
}
```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторых системах имитируемый признак `EOF` не поддерживается либо эта поддержка оказывается неидеальной, что может воспрепятствовать правильному выполнению приведенного выше примера. Если функция `cin.get()` применялась ранее для "замораживания" экрана до тех пор, пока не появится возможность ввода, то в данном случае она работать не будет, поскольку в результате обнаружения признака `EOF` исключаются дальнейшие попытки чтения ввода. Однако для того, чтобы сохранить экран видимым на некоторое время, можно воспользоваться циклом задержки времени, подобным приведенному в листинге 5.13.

Ниже приведен примерный результат выполнения указанной выше программы:

```
The sullen mackerel sulks in the
shadowy shallows.<ENTER>
The sullen mackerel sulks in the shadowy
shallows.
Yes, but the blue bird of happiness harbors
secrets.<ENTER>
Yes, but the blue bird of happiness harbors
secrets.
^Z
104 characters read
```

Проанализируем условие продолжения цикла:

```
while ((ch = cin.get()) != EOF)
```

Поскольку подвыражение `ch = cin.get()` заключено в скобки, программа сначала будет оценивать его значение. Для этого программа прежде всего вызывает функцию `cin.get()`. Далее она присваивает переменной `ch` возвращаемое данной функцией значение. Благодаря тому что значение оператора присваивания равно значению левого операнда, значение всего подвыражения сводится к значению переменной `ch`. Если это значение равно `EOF`, тогда цикл завершается, в противном же случае он продолжается. Все условие продолжения цикла должно быть заключено в скобки. Допустим, что это не сделано полностью:

```
while (ch = cin.get() != EOF)
```

Оператор `!` обладает более высоким приоритетом, чем оператор `=`, поэтому программа сравнивает с признаком `EOF` возвращаемое функцией `cin.get()` значение. В результате этого сравнения получается истинный или ложный результат, причем соответствующее значение типа `bool` преобразуется в 0 или 1, и именно это значение присваивается переменной `ch`.

С другой стороны, при использовании функции `cin.get(char)` (с одним аргументом) в процессе ввода никаких затруднений не возникает. Напомним, что функция `cin.get(char)` присваивает переменной `ch` специальное значение по достижению конца файла. На самом деле она в данном случае ничего переменной `ch` не присваивает, ведь переменная `ch` не призвана хранить отличное от символьного значение. Отличия между функциями `cin.get(char)` и `cin.get()` показаны в табл. 5.3.

Так в каких же случаях следует использовать функцию `cin.get()`, а в каких — `cin.get(char)`? Форма данной функции с одним аргументом более полно вписывается в объектный подход, поскольку значение, которое возвращает эта функция, является объектом класса `istream`. Это, в частности, означает возможность применять данную функцию несколько раз подряд. Например,

Таблица 5.3 Сравнение функций `cin.get(ch)` и `cin.get()`.

| Свойство                                                   | Функция <code>cin.get(ch)</code>                                                                                             | Функция <code>cin.get()</code>                                                                |
|------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| Способ передачи введенного символа                         | Присваивание значения аргументу <code>ch</code>                                                                              | Использование возвращаемого функцией значения для его присваивания переменной <code>ch</code> |
| Возвращаемое функцией значение при вводе символа           | Объект класса <code>istream</code> (значение <code>true</code> , получающееся после преобразования типа <code>bool</code> )  | Код символа в виде значения типа <code>int</code>                                             |
| Возвращаемое функцией значение при обнаружении конца файла | Объект класса <code>istream</code> (значение <code>false</code> , получающееся после преобразования типа <code>bool</code> ) | Признак <code>EOF</code>                                                                      |

представленный ниже код означает чтение следующего символа, введенного в переменную `ch1`, и последующего за ним символа, введенного в переменную `ch2`:

```
cin.get(ch1).get(ch2);
```

Такой способ оказывается действенным потому, что при вызове функции `cin.get(ch1)` возвращается объект `cin`, действующий затем в качестве объекта, к которому присоединяется функция `get(ch2)`.

Вероятно, основное применение формы `get()` состоит в возможности выполнять на скорую руку переход от функций `getchar()` и `putchar()` из заголовочного файла `stdio.h` к методам `cin.get(char)` и `cout.put()` из заголовочного файла `iostream`. Для этого достаточно заменить один заголовочный файл другим и выполнить глобальную замену функций `getchar()` и `putchar()` эквивалентными им методами. (Если в старом коде для ввода используется переменная типа `int`, тогда при наличии в данной реализации нескольких прототипов функции `put()` придется выполнить дополнительную настройку этого кода.)

## Вложенные циклы и двумерные массивы

Итак, цикл `for` естественным образом подходит для обработки массивов. Продвинемся в этом направлении еще на один шаг и посмотрим, каким образом один цикл `for` применяется в другом цикле `for` для обработки двумерных массивов (речь идет о вложенных циклах).

Прежде всего посмотрим, что собой представляет двумерный массив. Применявшиеся до сих пор массивы называются одномерными, поскольку каждый такой массив можно наглядно представить в виде одной строки данных. А вот двумерный массив может быть наглядно представлен в таком виде, который в большей степени подобен таблице, имеющей строки и столбцы данных. Двумерный массив можно, например, использовать для представления величин продаж за квартал в шести отдельных районах, причем каждый район представлен одной строкой данных. С другой стороны, двумерный

массив можно использовать для представления положения фигуры RoboDork в компьютеризированной настольной игре.

Специального типа двумерного массива в C++ не существует. Вместо этого создается массив, каждый элемент которого является собственно массивом. Допустим, например, что требуется сохранить данные о максимальной температуре воздуха в пяти городах за четырехлетний период. В этом случае можно объявить следующий массив:

```
int maxtemps[4][5];
```

Это объявление означает, что `maxtemps` является массивом из четырех элементов. При этом каждый элемент данного массива представляет собой массив из пяти целых значений (рис. 5.5). Таким образом, массив `maxtemps` можно рассматривать как состоящий из четырех строк, в каждой из которых приведено пять значений.

Выражение `maxtemps[0]` является первым элементом массива `maxtemps`, следовательно, `maxtemps[0]`, собственно, является массивом из пяти значений типа `int`. Первым элементом массива `maxtemps[0]` является `maxtemps[0][0]`, причем этот элемент содержит единственное значение типа `int`. Таким образом, для доступа к элементам массива типа `int` приходится использовать два индекса. Первый индекс можно рассматривать в качестве элемента, представляющего строку, а второй индекс — в качестве элемента, представляющего столбец (рис. 5.6).

Допустим, что требуется вывести содержимое всего массива. В этом случае для изменения строк можно воспользоваться одним циклом `for`, а для изменения столбцов — вторым, вложенным циклом `for`:

`maxtemps` является массивом из четырех элементов

```
int maxtemps[4][5];
```

Каждый элемент данного массива представляет собой массив из пяти целых значений

Массив `maxtemps`

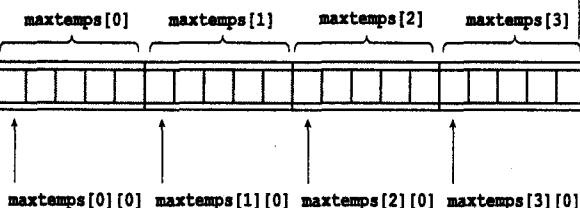


РИСУНОК 5.5 Двумерный массив.

```
for (int row = 0; row < 4; row++)
{
 for (int col = 0; col < 5; col++)
 cout << maxtemps[row][col] << "\t";
 cout << "\n";
}
```

Для каждого значения переменной `row` во внутреннем цикле `for` осуществляется поочередное обращение ко всем значениям переменной `col`. В данном примере символ табуляции (имеющий в C++ обозначение `\t`) выводится после каждого значения, а символ новой строки распечатывается после каждой полной строки.

## Инициализация двумерного массива

При создании двумерного массива имеется возможность инициализировать каждый его элемент. В основу данного метода положена инициализация одномерного массива. Напомним, что данный метод состоит в представлении разделенного запятыми списка значений, заключенных в фигурные скобки:

```
// инициализация одномерного массива
int btus[5] = { 23, 26, 24, 31, 28};
```

Что касается двумерного массива, то каждый его элемент, собственно, является массивом, поэтому каждый его элемент может быть инициализирован в приведенном выше виде. Таким образом, инициализация такого массива состоит из ряда разделенных запятыми инициализированных одномерных массивов, причем все они заключены в целый ряд фигурных скобок:

```
int maxtemps[4][5] = // Двумерный массив
{
 // Значения массива maxtemps[0]
 { 94, 98, 87, 103, 101 },
 // Значения массива maxtemps[1]
 { 98, 99, 91, 107, 105 },
 // Значения массива maxtemps[2]
 { 93, 91, 90, 101, 104 },
 // Значения массива maxtemps[3]
 { 95, 100, 88, 105, 103 }
};
```

| int maxtemps[4][5];                                           |   |                |                |                |                |                |
|---------------------------------------------------------------|---|----------------|----------------|----------------|----------------|----------------|
| массив <code>maxtemps</code> , рассматриваемый в виде таблицы |   |                |                |                |                |                |
|                                                               | 0 | 1              | 2              | 3              |                |                |
| maxtemps[0]                                                   | 0 | maxtemps[0][0] | maxtemps[0][1] | maxtemps[0][2] | maxtemps[0][3] | maxtemps[0][4] |
| maxtemps[1]                                                   | 1 | maxtemps[1][0] | maxtemps[1][1] | maxtemps[1][2] | maxtemps[1][3] | maxtemps[1][4] |
| maxtemps[2]                                                   | 2 | maxtemps[2][0] | maxtemps[2][1] | maxtemps[2][2] | maxtemps[2][3] | maxtemps[2][4] |
| maxtemps[3]                                                   | 3 | maxtemps[3][0] | maxtemps[3][1] | maxtemps[3][2] | maxtemps[3][3] | maxtemps[3][4] |

РИСУНОК 5.6 Доступ к элементам массива с помощью индексов.

Так, терм {94, 98, 87, 103, 101} инициализирует первую строку, представленную элементом `maxtemps[0]`. Ради соблюдения стиля каждую строку данных следует по возможности размещать в отдельной строке кода, что упрощает чтение данных.

В программе, приведенной в листинге 5.19, применяются двумерный массив и вложенный цикл. На сей раз циклы используются в данной программе в обратном порядке, т.е. цикл обработки столбцов (индекс городов) оказывается внешним, а цикл обработки строк (индекс лет) — внутренним. Кроме того, здесь применяется распространенный в C++ метод инициализации массива указателей на ряд строковых констант. Это означает, что `cities` объявляется в виде массива указателей типа `char`. При этом каждый элемент данного массива, в частности `cities[0]`, является указателем типа `char`, который может быть инициализирован адресом строки. Так, рассматриваемая программа инициализирует элемент `cities[0]` адресом строки "Gribble City" и т.д. Таким образом, данный массив указателей, по существу, является массивом строк.

#### Листинг 5.19 Программа nested.cpp.

```
// программа nested.cpp, демонстрирующая
// возможности циклов и вложенных массивов
#include <iostream>
using namespace std;
const int Cities = 5;
const int Years = 4;
int main()
{
 const char * cities[Cities] = // массив
 { // указателей на пять строк
 "Gribble City",
 "Gibbleton",
 "New Gribble",
 "San Gribble",
 "Gribble Vista"
 };

 int maxtemps[Years][Cities] = // двумерный
 // массив
 {
 // Значения массива maxtemps[0]
 { 94, 98, 87, 103, 101 },
 // Значения массива maxtemps[1]
 { 98, 99, 91, 107, 105 },
 // Значения массива maxtemps[2]
 { 93, 91, 90, 101, 104 },
 // Значения массива maxtemps[3]
 { 95, 100, 88, 105, 103 }
 };

 cout << "Maximum temperatures
 for 1995 - 1998\n\n";
 for (int city = 0; city < Cities; city++)
 {
 cout << cities[city] << ":";
```

```
for (int year = 0; year < Years;
 year++)
 cout << maxtemps[year][city] << "\t";
 cout << "\n";
}
return 0;
```

Ниже приведен результат выполнения указанной выше программы:

|                | Maximum temperatures for 1995 - 1998 |
|----------------|--------------------------------------|
| Gribble City:  | 94 98 93 95                          |
| Gibbleton:     | 98 99 91 100                         |
| New Gribble:   | 87 91 90 88                          |
| San Gribble:   | 103 107 101 105                      |
| Gribble Vista: | 101 105 104 103                      |

Благодаря применению при выводе символов табуляции данные располагаются более равномерно, чем при использовании пробелов. Более точные и вместе с тем более сложные методы форматирования представлены в главе 16.

## Резюме

В C++ предоставляются следующие три разновидности циклов: цикл `for`, цикл `while` и цикл `do while`. Цикл осуществляет неоднократно повторяющееся выполнение одних и тех же команд до тех пор, пока условие продолжения цикла будет истинным (ненулевым), причем выполнение цикла завершается в том случае, когда вычисляется ложное, т.е. нулевое значение условия продолжения цикла. Циклы `for` и `while` относятся к циклам с предусловием, означающим, что в них проверка условия осуществляется до выполнения операторов в теле цикла. А цикл `do while` является циклом с постусловием, означающим, что проверка условия в нем осуществляется после выполнения операторов в теле цикла.

Синтаксис каждого цикла требует, чтобы тело цикла состояло из одного оператора. Однако этим оператором может быть составной оператор или блок, образованный благодаря тому, что несколько операторов заключены в пару фигурных скобок.

Выражения сравнения, которые сравнивают два значения, нередко используются в качестве условий продолжения циклов. При этом выражения сравнения образуются с помощью одного из следующих шести операторов сравнения: `<`, `<=`, `==`, `>=`, `>` или `!=`. В результате вычисления выражений сравнения получаются значения `true` или `false` типа `bool`.

Во многих программах текст набирается или вводится из текстовых файлов посимвольно. Для этого в классе `istream` предоставляется несколько способов. Так, если `ch` является переменной типа `char`, тогда оператор

```
cin >> ch;
```

выполняет чтение следующего введенного символа в переменную *ch*. Однако при этом пропускаются символы пробела, новой строки и табуляции.

При вызове функции-элемента

```
cin.get(ch);
```

чтение следующего введенного символа выполняется независимо от его значения, а затем он размещается в переменной *ch*. Функция-элемент *cin.get()* возвращает следующий введенный символ, в том числе символы пробела, новой строки и табуляции, поэтому она может быть использована следующим образом:

```
ch = cin.get();
```

В результате вызова функции-элемента *cin.get(char)* сообщение о встретившемся условии конца файла выдается благодаря возврату значения *false* типа *bool*, тогда как при вызове функции *cin.get()* сообщение о конце файла выдается благодаря возврату признак *EOF*, который определяется в файле заголовка *iostream*.

Вложенный цикл представляет собой такой цикл, который находится внутри другого цикла. Вложенные циклы естественным образом применяются для обработки двумерных массивов.

## Вопросы для повторения

1. В чем отличие между циклом с предусловием и циклом с постусловием? К какому из указанных видов можно отнести каждый цикл C++?

2. Что бы вывел на печать следующий фрагмент кода, если бы он был включен в действительную программу:

```
int i;
for (i = 0; i < 5; i++)
 cout << i;
 cout << "\n";
```

3. Что бы вывел на печать следующий фрагмент кода, если бы он был включен в действительную программу:

```
int j;
for (j = 0; j < 11; j += 3)
 cout << j;
 cout << "\n" << j << "\n";
```

4. Что бы вывел на печать следующий фрагмент кода, если бы он был включен в действительную программу:

```
int j = 5;
while (++j < 9)
 cout << j++ << "\n";
```

5. Что бы вывел на печать следующий фрагмент кода, если бы он был включен в действительную программу:

```
int k = 8;
do
 cout << " k = " << k << "\n";
while (k++ < 5);
```

6. Создайте цикл *for*, который выводит на печать значения 1, 2, 4, 8, 16, 32, 64 при увеличении на 2 значения счетчика на каждом шаге цикла.

7. Каким образом можно включить в тело цикла несколько операторов?

8. Является ли следующий оператор действительным? Если нет, то почему? А если да, то что он выполняет?

```
int x = (1,024);
```

А как насчет следующего?

```
int y;
y = 1,024;
```

9. Чем выражение *cin>>ch* отличается от выражений *cin.get(ch)* и *ch=cin.get()* в отношении ввода?

## Упражнения по программированию

1. Напишите программу, которая запрашивает ввод двух целых чисел со стороны пользователя. Затем эта программа должна выполнять расчет и выдачу суммы всех целых чисел, находящихся в пределах между двумя введенными целыми числами. При этом предполагается, что первым вводится меньшее целое число. Например, если пользователь вводит 2 и 9, тогда программа сообщает, что сумма всех целых чисел от 2 до 9 составляет 44.

2. Напишите программу, которая запрашивает ввод чисел. После ввода каждого числа сообщается накопительная сумма введенных до сих пор чисел. Программа завершается после ввода нуля.

3. Дафна сделала вклад на сумму \$100 под простые проценты, которые составляют 10%, т.е. ежегодно ее вклад дает доход в виде 10% от первоначального вклада, или \$10, причем каждый год:

*проценты = 0.10 x начальный остаток*

В то же время Клео сделала вклад на сумму \$100 под сложные проценты, которые составляют 5%, т.е. 5% от текущего остатка, в том числе и предыдущие проценты:

*проценты = 0.05 x текущий остаток*

Доход Клео в виде 5% от суммы вклада \$100 за первый год составит \$105. В следующем году ее доход в виде 5% от суммы \$105 составит \$5.25 и т.д. Напишите программу, которая определяет, сколько лет потребуется для того, чтобы сумма вклада Клео превысила сумму вклада Дафны, а затем отображает сумму обоих вкладов в этот момент.

4. Допустим, что читатель занимается продажей книги "C++ для начинающих" (*C++ For Fools*). Напишите программу, которая требует ввода объема ежемесячных продаж этой книги в течение года (в экземплярах книг, а не в денежном выражении). В этой программе должен быть использован цикл, приглашающий ввести данные продаж за каждый месяц. Для этого используется массив указателей типа `char *`, инициализированных для указания на строки названий месяцев года, а введенные данные сохраняются в массиве значений типа `int`. После этого программа должна найти сумму содержимого массива и выдать отчет об общем объеме продаж за год.
5. Выполните упражнение 4, однако на сей раз воспользуйтесь двумерным массивом для хранения введенных данных о продажах в течение трех лет. Выдайте отчет об общем объеме продаж за каждый год в отдельности и в целом за все годы.

6. Разработайте структуру под названием `car`, в которой хранится следующая информация об автомобиле: его марка в виде строки в символьном массиве, а также год его выпуска в виде целого числа. Напишите программу, которая запрашивает пользователя, сколько автомашин следует ввести в каталог. Затем программа должна использовать метод `new` для создания нового динамического массива в соответствии с указанным количеством структур `car`. Далее она должна выдать приглашение на ввод со стороны пользователя марки (которая может состоять из нескольких слов) и года выпуска автомашины для каждой структуры. Следует заметить, что этот процесс требует некоторого внимания, поскольку при этом поочередно осуществляется чтение строковых и числовых данных (подробнее об этом сказано в главе 4). И наконец, программа должна отображать содержимое каждой структуры. Результат выполнения такой программы должен выглядеть следующим образом:

```
How many cars do you wish to catalog? 2
Car #1:
Please enter the make: Hudson Hornet
Please enter the year made: 1952
Car #2:
Please enter the make: Kaiser
Please enter the year made: 1951
Here is your collection:
1952 Hudson Hornet
1951 Kaiser
```

# Операторы ветвления и логические операции

**В этой главе рассматривается следующее:**

- Оператор `if`
- Оператор `if else`
- Логические операции: `&&`, `||` и `!`
- Библиотека символьных функций `cctype`
- Условный оператор `? :`
- Оператор `switch`
- Операторы `continue` и `break`
- Циклы считывания чисел

При разработке эффективных программ очень важно наделить их способностью принимать решения. В главе 5 был продемонстрирован один вид принятия решений — организация циклов, когда программа решает, нужно ли продолжать цикл. Теперь мы рассмотрим, как язык C++ позволяет использовать операторы ветвления для выбора одного из взаимоисключающих действий. Какое средство защиты от вампиров должно применяться в программе: чеснок или крест? Какой пункт меню выбрал пользователь? Ввел ли пользователь значение 0? Для принятия решений в языке C++ предлагаются операторы `if` и `switch`, и именно они являются основными темами этой главы. Читатели познакомятся также с условным оператором, который предоставляет еще один способ осуществления выбора, и с логическими операциями, которые позволяют объединять две проверки условий в одну.

## Оператор `if`

Когда программе C++ нужно решить, выполнять ли заданное действие, можно использовать оператор `if`. Этот оператор имеет две формы: `if` и `if else`. Давайте вначале рассмотрим более простую форму — `if`. Она подобна обычной языковой конструкции типа "Если у вас есть карточка капитана Кука, вы получите бесплатное печенье". Оператор `if` приводит к тому, что программа выполняет оператор или группу операторов, если проверочное условие истинно, и пропускает этот оператор или группу операторов, если условие ложно. Синтаксис этой формы оператора `if` аналогичен синтаксису оператора `while`:

`if (проверочное условие)`  
    `оператор`

Истинное или ненулевое *проверочное условие* приводит к выполнению программой *оператора*, которым может быть отдельный оператор или блок операторов. Ложное или нулевое *проверочное условие* приводит к тому, что *оператор* пропускается программой (рис. 6.1). Вся конструкция `if` обрабатывается как единый оператор.

Чаще всего *проверочное условие* — это выражение сравнения, подобное тем, что используются для управления циклами. Предположим, например, что требуется программа, подсчитывающая количество пробелов и общее количество символов ввода. Можно использовать

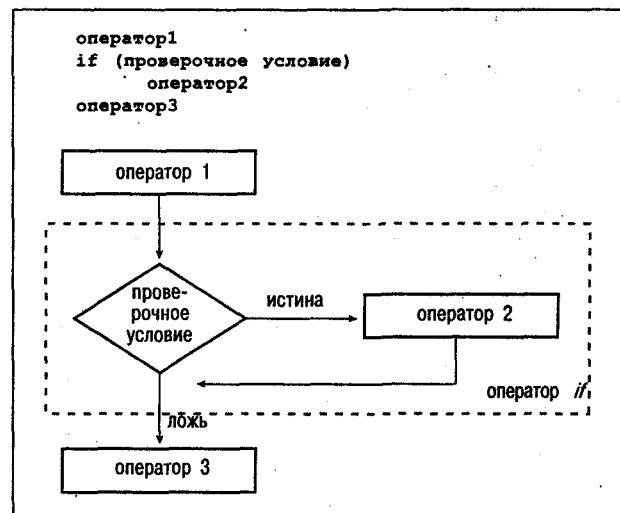


РИСУНОК 6.1 Оператор `if`.

функцию `cin.get(char)` в цикле `while` для считывания символов, а затем применить оператор `if` для выявления и подсчета символов пробела. Программа, приведенная в листинге 6.1, выполняет именно это, используя точку для распознавания конца предложения.

### Листинг 6.1 Программа if.cpp.

```
// if.cpp - использование оператора if
#include <iostream>
using namespace std;
int main()
{
 char ch;
 int spaces = 0;
 int total = 0;
 cin.get(ch);
 while (ch != '.') // выход в конце предложения
 {
 if (ch == ' ') // проверка, является
 // ли ch пробелом
 spaces++;
 total++; // выполняется каждый раз
 cin.get(ch);
 }
 cout << spaces << " spaces, " << total;
 cout << " characters total in sentence\n";
 return 0;
}
```

Результаты выполнения программы:

The balloonist was an airhead  
with lofty goals.  
6 spaces, 46 characters total in sentence

Как видно из комментариев, оператор `spaces++`; выполняется только в том случае, если `ch` — пробел. А оператор `total++`; выполняется при каждой итерации цикла, поскольку он находится вне оператора `if`. Обратите внимание, что при подсчете общего количества символов учитывается и символ новой строки, генерируемый при нажатии клавиши `Enter`.

### Оператор if else

Оператор `if` позволяет программе решить, должен ли выполняться конкретный оператор или блок. Оператор `if else` позволяет программе решить, какой из двух операторов или блоков должен выполняться. Этот оператор незаменим для создания взаимоисключающих последовательностей действий. Оператор `if else` языка C++ моделирует языковую конструкцию типа "Если у вас есть карточка капитана Кука, вы получите Cookie Plus Plus, в противном случае вы получите Cookie d'Ordinaire". Общая форма этого оператора выглядит следующим образом:

```
if (проверочное условие)
 оператор1
else
 оператор2
```

Если проверочное условие является истинным или ненулевым, программа выполняет `оператор1` и пропускает `оператор2`. В противном случае, когда значение проверочного условия является ложным или нулевым, программа пропускает `оператор1` и выполняет `оператор2`. Таким образом, фрагмент программы

```
if (answer == 1492)
 cout << "That's right!\n";
else
 cout << "You'd better review
 Chapter 1 again.\n";
```

выводит первое сообщение, если значение `answer` равно 1492, и второе сообщение — в противном случае. Каждый оператор может быть одиночным оператором или блоком операторов, ограниченным фигурными скобками (рис. 6.2). Синтаксически вся конструкция `if else` обрабатывается в качестве единого оператора.

Например, предположим, что нужно изменить поступающий текст, перепутывая буквы, но оставляя при этом символы новой строки неизменными. В результате каждая строка ввода преобразуется в строку вывода равной длины. Это означает, что программа должна выполнять одну последовательность действий по отношению к символам новой строки и другую последовательность действий по отношению ко всем остальным символам. Как видно из листинга 6.2, оператор `if else` делает эту задачу простой.

Обратите внимание, что в одном из комментариев программы говорится об интересном эффекте в случае изменения `++ch` на `ch+1`. Можете ли вы догадаться, в чем он будет заключаться? Если нет, то выполните это изменение и попытайтесь объяснить, что при этом происходит. (Подсказка: подумайте над тем, как оператор `cout` обрабатывает различные типы данных.)

```
оператор1
if (проверочное условие)
 оператор2
else
 оператор3
 оператор4
```

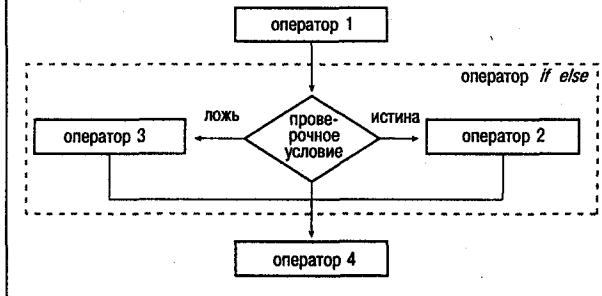


РИСУНОК 6.2 Оператор `if else`.

**Листинг 6.2 Программа ifelse.cpp.**

```
//ifelse.cpp - использование оператора if else
#include <iostream>
using namespace std;
int main()
{
 char ch;
 cout << "Type, and I shall repeat.\n";
 cin.get(ch);
 while (ch != '.')
 {
 if (ch == '\n')
 cout << ch; //выполняется, если ch -
 //символ новой строки
 else
 cout << ++ch; //выполняется в
 //противном случае
 cin.get(ch);
 }
 //для получения интересного эффекта вместо
 //++ch попытайтесь применить ch + 1
 cout << "\nPlease excuse the slight
 confusion.\n";
 return 0;
}
```

Результаты выполнения программы:

```
Type, and I shall repeat.
I am extraordinarily pleased
J!bn!fyusbpsejobsjmz!qmfbtf
to use such a powerful computer.
up!vtf!tvdi!b!qpxfsgvm!dpnqvufs
Please excuse the slight confusion.
```

**Форматирование операторов if else**

Имейте в виду, что две ветви оператора if else должны быть одиночными операторами. Если требуется более одного оператора, следует использовать фигурные скобки, чтобы объединить операторы в единый блок. В отличие от некоторых языков, таких как BASIC или FORTRAN, C++ не считает автоматически блоком все, что расположено между if и else, поэтому для заключения операторов в блок необходимо использовать фигурные скобки. Например, следующий фрагмент программы вызывает ошибку компилятора. Компилятор считает его простым оператором if, который заканчивается оператором zorrot++; Затем следует оператор cout. Пока все хорошо. Но затем следует ключевое слово else, которое компилятор воспринимает в качестве несвязанного и помечает как синтаксическую ошибку.

```
if (ch == 'Z')
 zorrot++; //оператор if
 //заканчивается здесь
 cout << "Another Zorro candidate\n";
else //ошибка
 dull++;
 cout << "Not a Zorro candidate\n";
```

Добавьте фигурные скобки, чтобы придать программному фрагменту нужный вид:

```
if (ch == 'Z')
{ //блок, выполняемый, если условие истинно
 zorrot++;
 cout << "Another Zorro candidate\n";
}
else
{ //блок, выполняемый, если условие ложно
 dull++;
 cout << "Not a Zorro candidate\n";
}
```

Поскольку язык C++ допускает свободную форму, фигурные скобки можно размещать как угодно, пока они заключают в себе операторы. В предшествующем примере приведен один из популярных форматов. А вот еще один:

```
if (ch == 'Z') {
 zorrot++;
 cout << "Another Zorro candidate\n";
}
else {
 dull++;
 cout << "Not a Zorro candidate\n";
}
```

Первая форма делает более очевидной блочную структуру операторов, в то время как вторая теснее связывает блоки с ключевыми словами if и else. Обе формы одинаково пригодны, если только вы не являетесь фанатичным приверженцем какого-либо конкретного стиля.

**Конструкция if else if else**

Компьютерные программы, как и реальная жизнь, могут ставить вас перед необходимостью выбора более чем из двух возможных вариантов. Для удовлетворения этой потребности можно расширить возможности оператора if else C++. Как было показано ранее, за ключевым словом else должен следовать единый оператор, который может быть блоком. Поскольку if else сам является единственным оператором, он может следовать за else:

```
if (ch == 'A')
 a_grade++; //альтернатива # 1
else
 if (ch == 'B') //альтернатива # 2
 b_grade++; //подальтернатива # 2a
 else
 soso++; //подальтернатива # 2b
```

Если ch — не равно 'A', программа переходит к else. Здесь второй оператор if else разветвляет эту альтернативу еще на две возможности. Свободное форматирование C++ позволяет располагать эти элементы в удобном для чтения формате:

```

if (ch == 'A')
 a_grade++;
else if (ch == 'B')
 b_grade++;
else
 soso++;

```

Это выглядит подобно новой управляющей структуре `if else if else`. Но в действительности это один оператор `if else`, содержащийся внутри другого. Этот измененный формат выглядит гораздо понятнее и позволяет просматривать код для выбора различных альтернатив. Вся конструкция по-прежнему считается одним оператором.

В листинге 6.3 только что описанное форматирование используется для создания небольшой шуточной программы.

### Листинг 6.3 Программа `ifelseif.cpp`.

```

// ifelseif.cpp -- использование конструкции
// if else if else
#include <iostream>
using namespace std;
const int Fave = 27;
int main()
{
 int n;

 cout << "Enter a number in the
 range 1-100 to find ";
 cout << "my favorite number: ";
 do
 {
 cin >> n;
 if (n < Fave)
 cout << "Too low - guess again: ";
 else if (n > Fave)
 cout << "Too high - guess again: ";
 else
 cout << Fave << " is right!\n";
 } while (n != Fave);
 return 0;
}

```

Результаты выполнения программы:

```

Enter a number in the range 1-100 to find
my favorite number: 50
Too high - guess again: 25
Too low - guess again: 37
Too high - guess again: 31
Too high - guess again: 28
Too high - guess again: 27
27 is right!

```

## Логические выражения

Часто приходится проверять более одного условия. Например, чтобы символ был строчной буквой, его значение должно быть больше или равно значению 'a' и меньше или равно 'z'. Или, если вы просите пользователя ответить вводом символа `u` или `n`, желательно, чтобы

прописные буквы (`Y` и `N`) воспринимались наряду со строчными. В целях удовлетворения подобной необходимости языка C++ предоставляет три логические операции для объединения или изменения существующих выражений. Этими операциями являются логическое ИЛИ, записываемое как `||`; логическое И, записываемое как `&&`; и логическое НЕ, записываемое как `!`. Давайте их рассмотрим.

### Операция логического ИЛИ: `||`

В разговорном языке слово `или` может указывать на то, что одно или оба условия удовлетворяют предъявляемому требованию. Например, вы можете попасть на пикник компании MegaMicro, если вы `или` ваша супруга работает в компании MegaMicro, Inc. Эквивалентом этой конструкции в языке C++ является операция логического ИЛИ, записываемая как `||`. Эта операция объединяет два выражения в одно. Если любое или оба исходных выражения имеют значение `true`, или ненулевое, результирующее выражение имеет значение `true`. В противном случае выражение имеет значение `false`. Ниже приведено несколько примеров:

```

5 == 5 || 5 == 9 //истинно, поскольку
 //первое выражение истинно
5 > 3 || 5 > 10 //истинно, поскольку
 //первое выражение истинно
5 > 8 || 5 < 10 //истинно, поскольку
 //второе выражение истинно
5 < 8 || 5 > 2 //истинно, поскольку оба
 //выражения истинны
5 > 8 || 5 < 2 //ложно, поскольку оба
 //выражения ложны

```

Поскольку операция `||` имеет более низкий приоритет, чем операции сравнения, в этих выражениях не требуется использовать скобки. Общее описание операции `||` приведено в табл. 6.1.

Таблица 6.1 Операция `||`.

Значение выражения `expr1 || expr2`

|                             | <code>expr1 == true</code> | <code>expr2 == false</code> |
|-----------------------------|----------------------------|-----------------------------|
| <code>expr2 == true</code>  | <code>true</code>          | <code>true</code>           |
| <code>expr2 == false</code> | <code>true</code>          | <code>false</code>          |

В соответствии с правилами C++ оператор `||` является *точкой последовательности*, т.е. любые изменения значения, указанные в левой части, происходят до вычисления правой части. Например, давайте рассмотрим следующее выражение:

```
i++ < 6 || i == j
```

Предположим, что первоначально `i` имеет значение, равное 10. К моменту сравнения с `j` переменная `i` полу-

чает значение, равное 11. Кроме того, C++ не станет утруждать себя вычислением выражения в правой части, если выражение в левой части истинно, поскольку для того, чтобы все логическое выражение было истинным, достаточно истинности одной части выражения. (Точка с запятой и оператор "запятая" также являются точками последовательности.)

В программе, приведенной в листинге 6.4, операция || в операторе if выполняется для проверки как строчных, так и прописных версий символа. Кроме того, в ней свойство конкатенации строк языка C++ (глава 4) используется для распределения единой строки по трем строкам.

#### Листинг 6.4 Программа or.cpp.

```
// or.cpp - использование логического или
#include <iostream>
using namespace std;
int main()
{
 cout << "This program may reformat
 your hard disk\n"
 "and destroy all your data.\n"
 "Do you wish to continue? <y/n> ";
 char ch;
 cin >> ch;
 if (ch == 'y' || ch == 'Y') //y или Y
 cout << "You were warned!\a\a\n";
 else if (ch == 'n' || ch == 'N') //n или N
 cout << "A wise choice ... bye\n";
 else
 cout << "That wasn't a y or an n,
 so I guess I'll "
 "trash your disk anyway.\n";
 return 0;
}
```

Результаты выполнения программы:

```
This program may reformat your hard disk
and destroy all your data.
Do you wish to continue? <y/n> N
A wise choice ... bye
```

Программа считывает только один символ, поэтому в ответе имеет значение только первый символ. Это означает, что пользователь мог бы ответить NO! вместо N. Программа прочла бы только N. Но если бы впоследствии программа попыталась продолжить считывание, она начала бы считывание с символа O.

#### Операция логического И: &&

Операция логического И, записываемая как &&, также объединяет два выражения в одно. Результирующее выражение имеет значение true только в том случае, если оба исходные выражения имеют значения true.

Вот несколько примеров:

```
5 == 5 && 4 == 4 //истинно, поскольку оба
 //выражения истинны
5 == 3 && 4 == 4 //ложно, поскольку первое
 //выражение ложно
5 > 3 && 5 > 10 //ложно, поскольку второе
 //выражение ложно
5 > 8 && 5 < 10 //ложно, поскольку первое
 //выражение ложно
5 < 8 && 5 > 2 //истинно, поскольку оба
 //выражения истинны
5 > 8 && 5 < 2 //ложно, поскольку оба
 //выражения ложны
```

Поскольку операция && имеет более низкий приоритет, чем операторы сравнения, в этих выражениях скобки не используются. Подобно операции ||, операция && действует в качестве точки последовательности, и поэтому левая часть выражения вычисляется, оказывая всяческие воздействия на программу, до вычисления правой части выражения. Если левая часть ложна, то должно и все выражение; поэтому в таком случае C++ не утруждает себя вычислением правой части. Краткое описание работы операции && приведено в табл. 6.2.

Таблица 6.2 Операция &&.

Значение выражения expr1 && expr2

|                | expr1 == true | expr2 == false |
|----------------|---------------|----------------|
| expr2 == true  | true          | false          |
| expr2 == false | false         | false          |

В листинге 6.5 продемонстрировано использование операции && в обычной ситуации прерывания цикла while по двум различным причинам. В программе этого листинга цикл while считывает значения в массив. Одно проверочное условие (*i* < ArSize) прерывает цикл, когда массив полон. Второе проверочное условие (*temp* >= 0) предоставляет пользователю возможность досрочно выйти из цикла, введя отрицательное число. Операция && позволяет объединить две проверки в единое условие. В программе используются также два оператора if, оператор if else и цикл for; поэтому она может служить иллюстрацией к нескольким темам этой и предыдущей главы.

#### Листинг 6.5 Программа and.cpp.

```
// and.cpp - использование логического И
#include <iostream>
using namespace std;
const int ArSize = 6;
int main()
{
 float naaq[ArSize];
 cout << "Enter the NAAQs (New Age
 Awareness Quotients) "
 << "of\your neighbors. Program
 terminates "
```

```

<< "when you make\n" << ArSize
<< " entries "
<< "or enter a negative value.\n";

int i = 0;
float temp;
cin >> temp;
while (i < ArSize && temp >= 0) //два
 //критерия выхода
{
 naaq[i++] = temp;
 if (i < ArSize) //в массиве остается
 //свободное место,
 cin >> temp; //поэтому необходимо
 //получить следующее значение
}
if (i == 0)
 cout << "No data-bye\n";
else
{
 cout << "Enter your NAAQ: ";
 float you;
 cin >> you;
 int count = 0;
 for (int j = 0; j < i; j++)
 if (naaq[j] > you)
 count++;
 cout << count;
 cout << " of your neighbors have
 greater awareness of\n"
 << "the New Age than you do.\n";
}
return 0;
}

```

Обратите внимание, что программа помещает результаты ввода во временную переменную `temp`. Только убедившись, что введенное значение является допустимым, программа присваивает значение массиву.

Ниже приведено несколько примеров выполнения программы. Выполнение первого примера прерывается после ввода шести значений, а второго — после ввода отрицательного значения:

Enter the NAAQs (New Age Awareness Quotients) of  
your neighbors. Program terminates when you make  
6 entries or enter a negative value.

28 72 19

6

130 145

Enter your NAAQ: 50

3 of your neighbors have greater awareness of  
the New Age than you do.

Enter the NAAQs (New Age Awareness Quotients) of  
your neighbors. Program terminates when you make  
6 entries or enter a negative value.

123 119

4

89

-1

Enter your NAAQ: 123.027

0 of your neighbors have greater awareness of  
the New Age than you do.

## Примечания к программе

Обратите внимание на часть программы, обеспечивающую ввод данных:

```

cin >> temp;
while (i < ArSize && temp >= 0) //два
 //критерия выхода
{
 naaq[i++] = temp;
 if (i < ArSize) //в массиве остается
 //свободное место
 cin >> temp; //поэтому необходимо
 //получить следующее значение
}

```

Программа начинает работу со считывания первого введенного значения во временную переменную, названную `temp`. Затем проверочное условие цикла `while` проверяет, остается ли еще свободное место в массиве (`i < ArSize`) и не является ли введенное значение отрицательным (`temp >= 0`). Если это условие выполняется, программа копирует значение `temp` в массив и увеличивает индекс массива на 1. На этот момент, поскольку нумерация массива начинается с 0, значение `i` равно общему числу записей. Другими словами, если `i` начинается с 0, то в результате первой итерации цикла присваивается значение элементу массива `naaq[0]`, а затем устанавливается значение `i`, равное 1.

Цикл прерывается, когда массив заполнен или когда пользователь вводит отрицательное число. Обратите внимание, что цикл считывает следующее значение в переменную `temp`, только если `i` меньше `ArSize`, т.е. только если в массиве есть свободное место.

После получения данных программа использует оператор `if else` для отображения комментария, если никакие новые данные не были введены (т.е. если первое введенное значение было отрицательным числом), и для обработки данных, если таковые имеются.

## Определение диапазонов с помощью операции &&

Операция `&&` позволяет также устанавливать последовательности операторов `if else if else`, каждая ветвь которых соответствует конкретному диапазону значений. Этот подход демонстрируется в листинге 6.6. В этой программе приводится также полезная методика обработки ряда сообщений. Аналогично тому как переменная указателя-на-`char` может идентифицировать отдельную строку, указывая на ее начало, массив указателей-на-`char` может идентифицировать последовательность строк. Для этого достаточно просто присвоить адрес каждой строки отдельному элементу массива. В программе из листинга 6.6 массив `qualify` используется для хранения адресов четырех строк. Например, элемент `qualify[1]` содержит адрес строки "mud tug-of-war\n". Затем программа может использовать `qualify[1]` подобно любому

другому указателю на строку, например, с оператором `cout` либо с функцией `strlen()` или `strcmp()`. Использование спецификатора `const` позволяет защитить эти строки от случайного изменения.

#### Листинг 6.6 Программа more\_and.cpp.

```
// more_and.cpp - использование логического И
#include <iostream>
using namespace std;
const char * qualify[4] = // массив указателей
{ //на строки
 "10,000-meter race.\n",
 "mud tug-of-war.\n",
 "masters canoe jousting.\n",
 "pie-throwing festival.\n"
};
int main()
{
 int age;
 cout << "Enter your age in years: ";
 cin >> age;
 int index;
 if (age > 17 && age < 35)
 index = 0;
 else if (age >= 35 && age < 50)
 index = 1;
 else if (age >= 50 && age < 65)
 index = 2;
 else
 index = 3;
 cout << "You qualify for the "
 << qualify[index];
 return 0;
}
```

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Читатели могут припомнить, что в некоторых реализациях C++ требуется использовать ключевое слово `static` в объявлении массива, чтобы его можно было инициализировать. Это ограничение, как описано в главе 8, применяется к массивам, объявленным внутри тела функции. Когда массив, подобно массиву `qualify` в листинге 6.6, объявляется вне тела функции, он называется **внешним массивом** и может быть инициализирован даже в реализациях языка, предшествовавших ANSI C.

Результаты выполнения программы:

```
Enter your age in years: 87
You qualify for the pie-throwing festival.
```

Введенный возраст не соответствует ни одному из проверяемых диапазонов, поэтому программа устанавливает значение индекса равным 3, а затем выводит соответствующую строку.

#### Примечания к программе

С помощью выражения `age > 17 && age < 35` проверяется значение возраста, заключенного между двумя границами, а именно в диапазоне 18–34. В выражении

`age >= 35 && age < 50` операция `<=` используется для включения значения 35 в указанный диапазон, которым является 35–49. Если бы в программе использовалось выражение `age > 35 && age < 50`, значение 35 не охватывалось бы ни одним тестом. При выполнении тестирования диапазона нужно проверять, нет ли пробелов между диапазонами и не накладываются ли они один на другой. Кроме того, следует правильно определить каждый диапазон (см. примечание "Проверки диапазонов").

Оператор `if else` служит для выбора индекса массива, который, в свою очередь, идентифицирует конкретную строку.

#### ПРОВЕРКИ ДИАПАЗОНОВ

Обратите внимание, что каждая часть проверки диапазона должна использовать оператор `И` для объединения двух завершенных выражений сравнения:

```
if (age > 17 && age < 35) // Правильно
```

Не следует использовать следующее математическое выражение:

```
if (17 < age < 35) // Не делайте этого!
```

Если допустить эту ошибку, компилятор ее не заметит, поскольку этот синтаксис все же допускается в C++. Оператор `<` связывает выражения слева направо, поэтому предыдущее выражение означает следующее:

```
if ((17 < age) < 35)
```

Однако `17 < age` имеет значение либо `true` (1), либо `false` (0). В любом случае значение выражения `17 < age` меньше `35`, поэтому все проверочное выражение всегда истинно!

#### Операция логического НЕ: !

Операция `!` отрицает или инвертирует следующее за ним выражение сравнения. Таким образом, если `выражение` имеет значение `true`, то `!выражение` имеет значение `false` и наоборот. Точнее говоря, если `выражение` имеет значение `true` или ненулевое, то `!выражение` имеет значение `false`, или нулевое. Кстати, многие называют восклицательный знак **ударом**, читая запись `!x` как "удар-икс", а `!!x` — как "удар-удар-икс".

Как правило, отношение можно выразить понятнее, не прибегая к этому оператору:

```
if (!(x > 5)) // if (x <= 5) понятней
```

Однако операция `!` может быть удобной при работе с функциями, которые возвращают значение типа "истинно-ложно" или значения, которые могут интерпретироваться таким образом. Например, `strcmp(s1,s2)` возвращает ненулевое значение (значение `true`), если две строки, `s1` и `s2`, отличаются одна от другой, и нулевое значение, если они одинаковы. Следовательно, `!strcmp(s1,s2)` имеет значение `true`, если две строки одинаковы.

В программе, приведенной в листинге 6.7, эта методика (выполнение операции `!` по отношению к возвращаемому значению функции) используется для скрытия численного ввода, чтобы вводимые значения можно было присваивать переменной типа `int`. Определяемая пользователем функция `is_int()`, которая будет описана далее, возвращает значение `true`, если ее аргумент находится в диапазоне значений, допустимых для типа `int`. Затем программа выполняет тестирование `while(!is_int(num))` для отсечения значений, не соответствующих этому диапазону.

### Листинг 6.7 Программа not.cpp.

```
// not.cpp - реализация операции отрицания
#include <iostream>
#include <climits>
using namespace std;
bool is_int(double);
int main()
{
 double num;

 cout << "Yo, dude! Enter an integer value: ";
 cin >> num;
 while (!is_int(num)) //продолжается до тех
 //пор, пока значение num не
 //становится допустимым для типа int
 {
 cout << "Out of range-please try again: ";
 cin >> num;
 }
 int val = num;
 cout << "You've entered the integer "
 << val << "\n";
 return 0;
}

bool is_int(double x)
{
 if (x <= INT_MAX && x >= INT_MIN) //использование значений climits
 return true;
 else
 return false;
}
```

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если система не поддерживает `climits`, используйте `limits.h`.

Ниже приведен пример выполнения программы при использовании 32-разрядного типа `int`:

```
Yo, dude! Enter an integer value: 6234128679
Out of range - please try again: -8000222333
Out of range - please try again: 99999
You've entered the integer 99999
```

#### Примечания к программе

При вводе значения, превышающего допустимые значения типа `int`, многие реализации языка просто усекают значения до допустимого размера, не сообщая о потере

данных. В приведенной программе этот недостаток устраняется путем считывания потенциального значения `int` в качестве значения типа `double`. Тип `double` имеет более чем достаточную точность для хранения типичного значения `int`, а его диапазон допустимых значений намного больше.

Булева функция `is_int()` использует две символьные константы (`INT_MAX` и `INT_MIN`), определенные в файле `climits` (глава 3), для определения того, находится ли значение ее аргумента в допустимых пределах. Если да, то программа возвращает значение `true`; в противном случае она возвращает значение `false`.

Программа `main()` использует цикл `while` для отклонения недопустимых вводимых значений до тех пор, пока пользователь не введет правильное значение. Программу можно сделать более дружественной, отображая допустимые пределы типа `int`, когда вводимое значение выходит за пределы этого диапазона. Как только введенное значение подтверждено, программа присваивает его переменной типа `int`.

### Немного о логических операциях

Как уже говорилось, операции логического И и логического ИЛИ в C++ имеют более низкий приоритет, чем операции сравнения. Это означает, что выражение типа

`x > 5 && x < 10`

читается следующим образом:

`(x > 5) && (x < 10)`

С другой стороны, операция `!` имеет более высокий приоритет, чем любые операции сравнения или арифметические операции. Следовательно, чтобы выполнить отрицание выражения, его нужно заключить в скобки:

`!(x > 5) //выражение ложно, если x больше 5`  
`!x > 5 //выражение истинно, если !x больше 5`

Кстати говоря, второе выражение всегда ложно, поскольку `!x` может иметь только значения `true` или `false`, которые преобразуются в значения 1 или 0 соответственно.

Операция логического И имеет более высокий приоритет, чем операция логического ИЛИ. Следовательно, выражение

`age > 30 && age < 45 || weight > 300`

означает следующее:

`(age > 30 && age < 45) || weight > 300`

Таким образом, одно условие состоит в проверке принадлежности переменной `age` диапазону от 31 до 44, а второе в том, чтобы `weight` было больше 300. Все выражение имеет значение `true`, если одно из этих условий или они истинны.

Конечно, для указания программе требуемого порядка действий можно использовать скобки. Например, предположим, что нужно использовать операцию `&&` для объединения условия "age больше 50 или weight больше 300" с условием "donation больше 1000". Для этого часть ИЛИ необходимо заключить в скобки:

```
(age > 50 || weight > 300) && donation > 1000
```

В противном случае компилятор объединит условие `weight` с условием `donation`, а не с условием `age`. Для группирования условий проще всего использовать скобки, независимо от того, нужны они или нет. Это облегчает чтение текста программы.

C++ гарантирует вычисление логического выражения слева направо и прекращение вычисления, как только ответ становится известен. Например, предположим, что имеется следующее условие:

```
x != 0 && 1.0 / x > 100.0
```

Если первое условие ложно, то и все выражение должно быть ложно. Это следует из того, что для истинности этого выражения каждое отдельное условие должно быть истинным. Зная, что первое условие ложно, программа не утруждает себя вычислением второго условия. В приведенном примере это и к лучшему, поскольку вычисление второго условия привело бы к операции деления на 0, что не допускается.

## Библиотека символьных функций ctype

Язык C++ унаследовал от С весьма удобный пакет функций, связанных с обработкой символов, прототипы которых описаны в заголовочном файле `cctype` (в файле `ctype.h` в более ранних версиях языка). Эти функции упрощают выполнение таких задач, как определение вида символа (строчная или прописная буква, цифра или знак препинания) и т.п. Например, функция `isalpha(ch)` возвращает ненулевое значение, если `ch` — буква, и нуль — во всех остальных случаях. Аналогично функция `ispunct(ch)` возвращает значение `true`, если `ch` — символ пунктуации, такой как запятая или точка. (Возвращаемые значения этих функций имеют тип `int`, а не `bool`, но обычные преобразования типа `bool` позволяют работать с ними как с типом `bool`.)

Эти функции использовать удобнее, чем операторы И и ИЛИ. Например, ниже показано, как операции И и ИЛИ можно было бы использовать для проверки того, является ли символ `ch` алфавитным:

```
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A'
&& ch <= 'Z'))
```

Сравните это с использованием функции `isalpha()`:

```
if (isalpha(ch))
```

Функция `isalpha()` не только проще в использовании, она также является более общей. Форма с использованием операторов И и ИЛИ предполагает, что коды символов от A до Z образуют непрерывную последовательность и что коды никаких других символов не попадают в этот диапазон. Это допущение справедливо для кода ASCII, но не в общем случае.

Использование некоторых функций из этого семейства продемонстрировано в листинге 6.8. В частности, в приведенной в нем программе используются следующие функции: `isalpha()`, которая выполняет проверку на предмет принадлежности символов к алфавитным; `isdigit()`, которая проверяет наличие таких цифр, как 3; `isspace()`, которая осуществляет проверку символов пробелов, таких как символы новой строки, пробелы между буквами и символы табуляции; `ispunct()`, которая проверяет наличие символов пунктуации. В программе также используются структура `if else if` и цикл `while` совместно с функцией `cin.get(char)`.

### Листинг 6.8 Программа cctypes.cpp.

---

```
//cctypes.cpp—использование библиотеки ctype.h
#include <iostream>
#include <cctype> //прототипы символьных функций
using namespace std;
int main()
{
 cout << "Enter text for analysis, and type @"
 " to terminate input.\n";
 char ch;
 int whitespace = 0;
 int digits = 0;
 int chars = 0;
 int punct = 0;
 int others = 0;
 cin.get(ch); //получение первого символа
 while(ch != '@') //проверка на принад-
 { //лежность к зарезервированным символам
 // этот символ — алфавитный?
 if(isalpha(ch))
 chars++;
 // этот символ — пробел?
 else if(isspace(ch))
 whitespace++;
 // этот символ — цифра?
 else if(isdigit(ch))
 digits++;
 // этот символ — знак пунктуации?
 else if(ispunct(ch))
 punct++;
 // получение следующего символа
 else
 others++;
 cin.get(ch);
 }
 cout << chars << " letters, "
 << whitespace << " whitespace, "
 << others << " others.";
```

```

<< digits << " digits,
<< punct << " punctuations,
<< others << " others.\n";
return 0;
}

```

Ниже приведен пример выполнения программы; обратите внимание, что в число символов пробелов включаются и символы новой строки:

```

Enter text for analysis, and type @ to
terminate input.
Jody "Java-Java" Joystone, noted restaurant critic,
celebrated her 39th birthday with a carafe of 1982
Chateau Panda.@
89 letters, 16 whitespace, 6 digits,
6 punctuations, 0 others.

```

В табл. 6.3 приведено краткое описание функций, доступных в пакете `cctype`. В некоторых системах могут быть представлены не все из этих функций или включены дополнительные функции.

## Оператор ?:

В языке C++ имеется оператор, который часто может использоваться вместо оператора `if else`. Этот оператор называется **условным** и записывается как `?:`. В большинстве случаев это единственный оператор C++, требующий использования трех operandов. Его общая форма выглядит следующим образом:

```
выражение1 ? выражение2 : выражение3
```

Если `выражение1` истинно, то значением всего условного выражения является значение `выражение2`.

**Таблица 6.3 Символьные функции `cctype`.**

| Имя функции             | Возвращаемое значение                                                                                                                                                             |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>isalnum()</code>  | True, если аргумент является алфавитно-цифровым символом, т.е. буквой или цифрой                                                                                                  |
| <code>isalpha()</code>  | True, если аргумент является алфавитным символом                                                                                                                                  |
| <code>iscntrl()</code>  | True, если аргумент является управляющим символом                                                                                                                                 |
| <code>isdigit()</code>  | True, если аргумент является десятичной цифрой (0–9)                                                                                                                              |
| <code>isgraph()</code>  | True, если аргумент является любым печатаемым символом, отличным от пробела                                                                                                       |
| <code>islower()</code>  | True, если аргумент является строчной буквой                                                                                                                                      |
| <code>ispunct()</code>  | True, если аргумент является любым печатаемым символом, включая пробел                                                                                                            |
| <code>isspace()</code>  | True, если аргумент является символом пунктуации                                                                                                                                  |
| <code>isupper()</code>  | True, если аргумент является прописной буквой                                                                                                                                     |
| <code>isxdigit()</code> | True, если аргумент является символом шестнадцатиричной цифры, т.е. 0–9, a–f или A–F                                                                                              |
| <code>tolower()</code>  | Если аргумент является прописной буквой, функция <code>tolower()</code> возвращает строчную версию этого символа; в противном случае она возвращает неизменное значение аргумента |
| <code>toupper()</code>  | Если аргумент является строчной буквой, функция <code>toupper()</code> возвращает прописную версию этого символа; в противном случае она возвращает неизменное значение аргумента |

В противном случае значением всего выражения является значение `выражения3`. Ниже приведено несколько примеров, в которых демонстрируется работа этого оператора:

```

5 > 3 ? 10 : 12 //5 > 3 истинно, поэтому
//значением выражения является 10
3 == 9? 25 : 18 //3 == 9 ложно, поэтому
//значением выражения является 18

```

Первый пример можно перефразировать следующим образом: если 5 больше 3, выражению присваивается значение 10; в противном случае ему присваивается значение 12. Конечно, в реальных программах в выражениях использовались бы переменные.

В программе, приведенной в листинге 6.9, условный оператор используется для определения большего из двух значений.

### Листинг 6.9 Программа condit.cpp.

```

//condit.cpp—использование условного оператора
#include <iostream>
using namespace std;
int main()
{
 int a, b;
 cout << "Enter two integers: ";
 cin >> a >> b;
 cout << "The larger of " << a
 << " and " << b;
 int c = a > b ? a : b; //если a > b,
 //то c = a; в противном случае c = b
 cout << " is " << c << "\n";
 return 0;
}

```

Результаты выполнения программы:

```
Enter two numbers: 25 27
The larger of 25 and 27 is 27
```

Основным в этой программе является оператор

```
int c = a > b ? a : b;
```

Он приводит к такому же результату, что и следующие операторы:

```
int c;
if (a > b)
 c = a;
else
 c = b;
```

По сравнению с последовательностью `if else` условный оператор более лаконичен, но в то же время имеет менее понятный синтаксис. Одно из различий между этими двумя подходами состоит в том, что условный оператор создает выражение и, следовательно, генерируется единственное значение, которое может быть присвоено или вставлено в большее выражение (здесь это было сделано путем присвоения значения условного выражения переменной `c`). Лаконичная форма, необычный синтаксис и общий "загадочный" вид условного оператора делают его весьма привлекательным для тех программистов, которые ценят эти качества. Один из распространенных приемов достижения достойной всяческого порицания цели скрытия назначения кода заключается во внедрении условных выражений одного внутрь другого, как показано в следующем кратком примере:

```
const char x[2] [20] = {"Jason ",
 "at your service\n"};
const char * y = "Quillstone ";
for (int i = 0; i < 3; i++)
 cout << ((i < 2)? i? x [i] : y x[1]);
```

Это всего лишь немного замысловатый способ вывода трех строк в следующем порядке:

```
Jason Quillstone at your service
```

## Оператор `switch`

Предположим, что нужно создать экранное меню, которое предлагает пользователю выбрать один из пяти пунктов, например, Cheap, Moderate, Expensive, Extravagant и Excessive. Последовательность `if else if else` можно расширить для обработки пяти взаимоисключающих вариантов, но оператор `switch` C++ больше подходит для выполнения выбора из расширенного списка. Общая форма оператора `switch` выглядит следующим образом:

```
switch (целочисленное выражение)
{
 case метка1 : оператор(и)
 case метка2 : оператор(и)
 ...
 default : оператор(ы)
}
```

Оператор `switch` в языке C++ действует в качестве переключателя, который указывает компьютеру, какую строку программы нужно выполнять следующей. Встретив ключевое слово `switch`, программа переходит к строке, помеченной значением, соответствующим значению **целочисленного выражения**. Например, если **целочисленное выражение** имеет значение, равное 4, программа переходит к строке с меткой `case 4`: Значением **целочисленного выражения**, как следует из его названия, должно быть выражение, при вычислении дающее целочисленное значение. Кроме того, каждая метка должна быть выражением целочисленной константы. Чаще всего метками являются просто константы типа `int` или `char`, такие как 1 или q, либо перечислители. Если **целочисленное выражение** не соответствует ни одной из меток, программа переходит к строке, помеченной меткой `default`. Эта метка необязательна. Если ее опустить и при этом совпадение с метками отсутствует, программа переходит к оператору, следующему за `switch` (рис. 6.3).

Оператор `switch` существенно отличается от аналогичных операторов таких языков, как Pascal. В языке C++ каждая метка `case` действует только в качестве метки строки, а не разграничителя между вариантами. Иначе говоря, после того, как программа переходит к конкретной строке в операторе `switch`, она по порядку выполняет все операторы, которые следуют за этой строкой, если только явно не указано иное. Выполнение операции `НЕ` останавливается автоматически на следующем варианте. Для остановки выполнения в конце конкретной группы операторов необходимо использовать оператор `break`. В результате выполнение продолжится с оператора, следующего за `switch`.

Совместное использование операторов `switch` и `break` для реализации простого меню взаимоисключающих вариантов показано в листинге 6.10. Для отображения набора вариантов в программе используется функция `showmenu()`. Затем оператор `switch` выбирает действие, исходя из ответа пользователя.

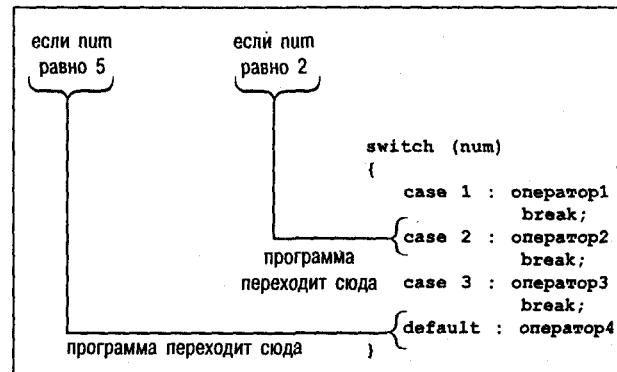


РИСУНОК 6.3 Оператор `switch`.

## ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторых реализациях управляющая последовательность \a обрабатывается как символ блокировки вывода.

### Листинг 6.10 Программа switch.cpp.

```
// switch.cpp - использование оператора switch
#include <iostream>
using namespace std;
void showmenu(); // прототипы функций
void report();
void comfort();
int main()
{
 showmenu();
 int choice;
 cin >> choice;
 while (choice != 5)
 {
 switch(choice)
 {
 case 1 : cout << "\a\n";
 break;
 case 2 : report();
 break;
 case 3 : cout << "The boss was
 `in all day.\n";
 break;
 case 4 : comfort();
 break;
 default : cout << "That's not a choice.\n";
 }
 showmenu();
 cin >> choice;
 }
 cout << "Bye!\n";
 return 0;
}

void showmenu()
{
 cout << "Please enter 1, 2, 3, 4, or 5:\n"
 "1) alarm 2) report\n"
 "3) alibi 4) comfort\n"
 "5) quit\n";
}
void report()
{
 cout << "It's been an excellent week
 `for business.\n"
 "Sales are up 120%. Expenses
 `are down 35%.\n";
}
void comfort()
{
 cout << "Your employees think you are
 `the finest CEO\n"
 "in the industry. The board of
 `directors think\n"
 "you are the finest CEO in the
 `industry.\n";
}
```

Ниже приведен пример выполнения программы меню выбора взаимоисключающих вариантов.

```
Please enter 1, 2, 3, 4, or 5:
1) alarm 2) report
3) alibi 4) comfort
5) quit
4
Your employees think you are the finest CEO
in the industry. The board of directors think
you are the finest CEO in the industry.
Please enter 1, 2, 3, 4, or 5:
1) alarm 2) report
3) alibi 4) comfort
5) quit
2
It's been an excellent week for business.
Sales are up 120%. Expenses are down 35%.
Please enter 1, 2, 3, 4, or 5:
1) alarm 2) report
3) alibi 4) comfort
5) quit
6
That's not a choice.
Please enter 1, 2, 3, 4, or 5:
1) alarm 2) report
3) alibi 4) comfort
5) quit
5
Bye!
```

Выполнение цикла while прерывается, когда пользователь вводит 5. Ввод чисел от 1 до 4 приводит к выбору соответствующего варианта из списка switch, а ввод числа 6 приводит к переходу к операторам, определенным по умолчанию.

Как уже отмечалось ранее, для ограничения выполнения определенной частью переключателя программе требуются операторы break. Чтобы убедиться в этом, можно удалить операторы break из листинга 6.10 и посмотреть, как программа будет выполняться после этого. В частности, выяснится, что ввод значения, равного 2, приводит к выполнению программой всех операторов, связанных с метками 2, 3, 4 и default. В C++ такое поведение программы определено умышленно, поскольку оно может быть полезным, в первую очередь тем, что упрощается использование нескольких меток. Например, предположим, что в листинге 6.10 в качестве пунктов меню и меток используются символы, а не целые числа. Тогда для одних и тех же операторов можно было бы использовать как прописные, так и строчные символы меток:

```
char choice;
cin >> choice;
while (choice != 'Q' && choice != 'q')
{
 switch(choice)
 {
 case 'a':
```

```

case 'A': cout << "\a\n";
 break;
case 'r':
case 'R': report();
 break;
case 'l':
case 'L': cout << "The boss was
 in all day.\n";
 break;
case 'c':
case 'C': comfort();
 break;
default : cout << "That's not a
 choice.\n";
}
showmenu();
cin >> choice;
}

```

Поскольку непосредственно за меткой `case 'a'` не стоит оператор `break`, выполнение программы продолжается со следующей строки, которой является оператор, расположенный сразу после `case 'A'`.

### Использование перечислителей в качестве меток

В листинге 6.11 иллюстрируется использование `enum` для определения набора связанных констант и последующее использование констант в операторе `switch`. В общем случае функция `cin` не распознает нумерованные типы (она не может знать, как пользователь определит их), поэтому программа считывает варианты как значения

### Листинг 6.11 Программа enum.cpp.

```

// enum.cpp - использование enum
#include <iostream>
using namespace std;
// создание именованных констант
enum { red, orange, yellow, green, blue, violet, indigo} ;

int main()
{
 cout << "Enter color code (0-6): ";
 int code;
 cin >> code;
 while (code >= red && code <= indigo)
 {
 switch (code)
 {
 case red : cout << "Her lips were red.\n"; break;
 case orange : cout << "Her hair was orange.\n"; break;
 case yellow : cout << "Her shoes were yellow.\n"; break;
 case green : cout << "Her nails were green.\n"; break;
 case blue : cout << "Her sweatsuit was blue.\n"; break;
 case violet : cout << "Her eyes were violet.\n"; break;
 case indigo : cout << "Her mood was indigo.\n"; break;
 }
 cout << "Enter color code (0-6): ";
 cin >> code;
 }
 cout << "Bye\n";
 return 0;
}

```

типа `int`. Когда оператор `switch` сравнивает значение типа `int` с меткой варианта перечислителя, она повышает тип перечислителя до `int`. Тип перечислителей также повышается до `int` в проверочном условии цикла `while`.

Ниже приведены результаты выполнения программы:

```

Enter color code (0-6): 3
Her nails were green.
Enter color code (0-6): 5
Her eyes were violet.
Enter color code (0-6): 2
Her shoes were yellow.
Enter color code (0-6): 8
Bye

```

### Операторы switch и if else

И оператор `switch`, и оператор `if else` позволяют программе осуществить выбор из списка взаимоисключающих вариантов. Оператор `if else` более гибок. Например, он может обрабатывать диапазоны, как показано в следующем примере:

```

if (age > 17 && age < 35)
 index = 0;
else if (age >= 35 && age < 50)
 index = 1;
else if (age >= 50 && age < 65)
 index = 2;
else
 index = 3;

```

Однако оператор **switch** не предназначен для обработки диапазонов. Каждая метка варианта оператора **switch** должна быть отдельным значением. Кроме того, это значение должно быть целым (включая тип **char**), поэтому оператор **switch** не будет выполнять проверку значений с плавающей точкой. И значение метки варианта должно быть константой. Если реализация вариантов требует использования диапазонов, осуществления проверки значений с плавающей точкой или сравнения двух переменных, следует использовать оператор **if else**.

Однако, если все варианты могут быть идентифицированы целочисленными константами, можно использовать как оператор **switch**, так и оператор **if else**. Поскольку оператор **switch** разработан именно для обработки таких ситуаций, обычно его использование более эффективно с точки зрения размера программы и времени выполнения, если только подлежащие выбору варианты не ограничиваются единицами.

#### СОВЕТ

Если можно использовать как последовательность **if else if**, так и оператор **switch**, то оператор **switch** следует использовать при наличии трех и более возможных вариантов.

## Операторы **break** и **continue**

Операторы **break** и **continue** позволяют программе пропускать фрагменты кода. Оператор **break** можно использовать в операторе **switch** и в любых циклах. Этот оператор вызывает переход к оператору, следующему за оператором **switch** или циклом. Оператор **continue** используется в циклах и приводит к тому, что программа пропускает остальную часть тела цикла и начинает новую итерацию цикла (рис. 6.4).

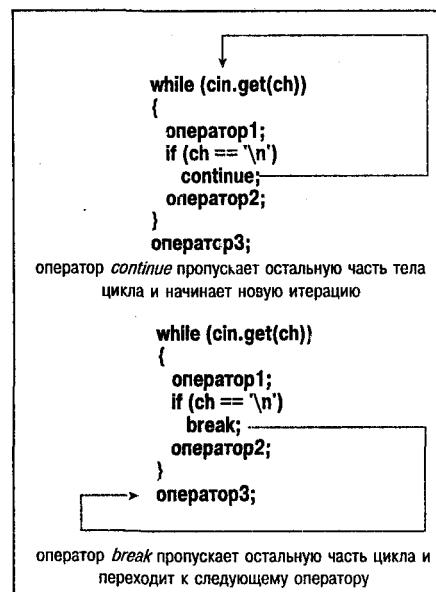


РИСУНОК 6.4  
Операторы **break** и **continue**.

Выполнение обоих операторов показано в листинге 6.12. Программа позволяет вводить строку текста. Цикл повторяет каждый символ и использует оператор **break** для прерывания цикла, если символ является точкой. Этот пример демонстрирует, как можно использовать оператор **break** для прерывания цикла изнутри, если некоторое условие становится истинным. Затем программа подсчитывает количество пробелов, но не остальных символов. Оператор **continue** используется для пропуска связанный с подсчетом части цикла, когда символ не является пробелом.

#### Листинг 6.12 Программа **jump.cpp**.

```

// jump.cpp - использование операторов
// continue и break
#include <iostream>
using namespace std;
const int ArSize = 80;
int main()
{
 char line[ArSize];
 int spaces = 0;

 cout << "Enter a line of text:\n";
 cin.get(line, ArSize);
 for (int i = 0; line[i] != '\0'; i++)
 cout << line[i]; // отображение символа
 if (line[i] == '.') // выход, если это точка
 break;
 if (line[i] != ' ') // пропуск остальной
 // части цикла
 continue;
 spaces++;
 }
 cout << "\n" << spaces << " spaces\n";
 return 0;
}

```

Результаты выполнения программы:

```

Let's do lunch today. You can pay!
Let's do lunch today.
3 spaces

```

#### Примечания к программе

Обратите внимание, что, хотя оператор **continue** заставляет программу пропустить остальную часть тела цикла, он не препятствует выполнению выражения обновления цикла. В цикле **for** оператор **continue** заставляет программу перейти непосредственно к выражению обновления, а затем к проверочному выражению. Однако в цикле **while** оператор **continue** заставляет программу перейти непосредственно к проверочному выражению. Таким образом, любое выражение обновления в теле цикла **while**, следующее за **continue**, будет пропущено. В ряде случаев это может создавать проблему.

В приведенной программе использование оператора **continue** было необязательным. Вместо него можно было бы использовать следующий программный текст:

```
if (line[i] == ' ')
 spaces++;
```

Однако оператор `continue` может делать программу более читабельной, когда за `continue` следует несколько операторов. В результате все эти операторы не должны быть частью оператора `if`.

C++, подобно языку C, имеет оператор `goto`. Оператор типа

```
goto paris
```

означает переход к точке программы, помеченной меткой `paris`. Это значит, что можно использовать программный текст, подобный следующему:

```
char ch;
cin >> ch;
if (ch == 'P')
 goto paris;
cout < ...
...
paris: cout << "You've just arrived at
 Paris.\n";
```

Однако в большинстве случаев не рекомендуется использовать оператор `goto`, для управления ходом выполнения программы следует использовать такие структурированные операторы управления, как `if else`, `switch`, `continue` и т.п.

## Циклы считывания чисел

Предположим, что была создана программа для считывания ряда чисел в массив. Пользователю необходимо предоставить возможность прервать ввод до заполнения массива. Один из возможных способов достижения этого — использование особенностей метода `cin`. Давайте рассмотрим следующий программный фрагмент:

```
int n;
cin >> n;
```

Что происходит, если пользователь вместо числа вводит слово? В этом случае возможны четыре варианта:

- Значение `n` остается неизменным.
- Несоответствующие входные данные остаются в очереди ввода.
- Флаг ошибки устанавливается для объекта `cin`.
- При вызове метода `cin` в случае преобразования в тип `bool` возвращается значение `false`.

Возвращение методом значения `false` свидетельствует о том, что для прерывания цикла считывания чисел нельзя вводить нечисловые значения. Установка флага ошибки при нечисловом вводе означает, что, прежде чем программа сможет продолжить считывание ввода, необходимо переустановить флаг. Метод `clear()`, с по-

мощью которого также переустанавливается условие конца файла (глава 5), позволяет переустановить флаг неверного ввода. (И неверный ввод, и конец файла могут привести к тому, что метод `cin` вернет значение `false`. Различие между этими двумя случаями описано в главе 16.) Давайте рассмотрим несколько примеров, демонстрирующие эти методики.

Предположим, что нужно создать программу, которая подсчитывает средний вес выловленной за день рыбы. Количество пойманной рыбы ограничивается пятью единицами, поэтому все данные могут быть размещены в пятиэлементном массиве, но возможно, что рыбы было поймано и меньше пяти штук. В программе из листинга 6.13 используется цикл, который прерывается, если массив полон или если вводится отличающееся от числового значение.

### Листинг 6.13 Программа `cinfish.cpp`.

```
// cinfish.cpp - нечисловой ввод прерывает
// выполнение цикла
#include <iostream>
using namespace std;
const int Max = 5;
int main()
{
 // получение данных
 double fish[Max];
 cout << "Please enter the weights
 of your fish.\n";
 cout << "You may enter up to " << Max
 << " fish <q to terminate>.\n";
 cout << "fish #1: ";
 int i = 0;
 while (i < Max && cin >> fish[i]) {
 if (++i < Max)
 cout << "fish #" << i+1 << ": ";
 }
 // вычисление среднего значения
 double total = 0.0;
 for (int j = 0; j < i; j++)
 total += fish[j];
 // вывод результатов
 if (i == 0)
 cout << "No fish\n";
 else
 cout << total/i << " = average weight of "
 << i << " fish\n";
 return 0;
}
```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Некоторые компиляторы Borland выводят предупреждение, что в строке

```
cout << "fish #" << i+1 << ': ';
```

"сомнительные" операторы требуют применения скобок. Не беспокойтесь. Это предупреждение сообщает всего лишь о возможной ошибке группирования при использовании `<<` в его первоначальном значении оператора сдвига влево.

Выражение `cin >> fish[i]` в действительности представляет собой вызов функции метода `cin`, и функция возвращает `cin`. Если значение `cin` является частью проверочного условия, оно преобразуется в тип `bool`. Значение преобразования равно `true`, если ввод успешен, и `false` — в противном случае. Значение выражения, равное `false`, прерывает цикл. Вот как выглядит результат выполнения программы:

```
Please enter the weights of your fish.
You may enter up to 5 fish <q to
terminate>.
fish #1: 30
fish #2: 35
fish #3: 25
fish #4: 40
fish #5: q
32.5 = average weight of 4 fish
```

Обратите внимание на следующую строку:

```
while (i < Max && cin >> fish[i]) {
```

Вспомните, что C++ не вычисляет правую часть логического выражения И, если левая часть ложна. В данном случае вычисление правой части означает использование `cin` для помещения результатов ввода в массив. Если `i` равно `Max`, цикл прерывается без попытки размещения значения в ячейке, следующей за концом массива.

В последнем примере не предпринималась попытка считать какие-либо входные данные после ввода нечислового значения. Давайте рассмотрим пример, в котором это делается. Предположим, что в программу C++ нужно ввести ровно пять значений набранных в гольфе очков для получения среднего значения. Если пользователь вводит значение, не являющееся числовым, программа должна вывести сообщение, настаивая на вводе числового значения. Как было показано, для проверки результатов нечислового ввода можно использовать значение вводимого выражения `cin`. Предположим, выяснилось, что пользователь ввел неверные данные. Необходимо предпринять три действия:

- Переустановить `cin` для приема нового вводимого значения.
- Избавиться от неверно введенного значения.
- Предложить пользователю повторить попытку.

Обратите внимание, что, прежде чем избавляться от неверно введенного значения, необходимо переустановить `cin`. Возможный способ выполнения этих задач показан в листинге 6.14.

#### Листинг 6.14 Программа cingolf.cpp.

```
// cingolf.cpp - пропуск нечислового ввода
#include <iostream>
using namespace std;
const int Max = 5;
int main()
{
 // получение данных
 int golf[Max];
 cout << "Please enter your golf scores.\n";
 cout << "You must enter " << Max
 << " rounds.\n";
 int i;
 for (i = 0; i < Max; i++)
 {
 cout << "round #" << i+1 << ": ";
 while (!(cin >> golf[i])) {
 cin.clear(); //переустановка ввода
 while (cin.get() != '\n')
 continue; //удаление результатов
 //ввода неверного значения
 cout << "Please enter a number: ";
 }
 // вычисление среднего значения
 double total = 0.0;
 for (i = 0; i < Max; i++)
 total += golf[i];
 // вывод результатов
 cout << total / Max << " = average score "
 << Max << " rounds\n";
 }
 return 0;
}
```

#### ЗАМЕЧАНИЕ ГОЛОВОДУ СОВМЕСТИМОСТИ

Некоторые компиляторы Borland выводят предупреждение, что в строке

```
cout << "fish #" << i+1 << ': ';
```

"сомнительные" операторы требуют применения скобок. Не беспокойтесь. Это предупреждение сообщает всего лишь о возможной ошибке группирования при использовании символа `<<` в его первоначальном значении операции сдвига влево.

Результаты выполнения программы:

```
Please enter your golf scores.
You must enter 5 rounds.
round #1: 88
round #2: 87
round #3: must i?
Please enter a number: 103
round #4: 94
round #5: 86
91.6 = average score 5 rounds
```

## Примечания к программе

Ядром кода обработки ошибок является следующий фрагмент:

```
while (!(cin >> golf[i])) {
 cin.clear(); // переустановка ввода
 while (cin.get() != '\n')
 continue; // избавление от результатов
 // неверного ввода
 cout << "Please enter a number: ";
}
```

Если пользователь вводит 88, то значением выражения `cin` является `true`, вводимое значение помещается в массив, значением выражения `!(cin >> golf[i])` становится `false` и выполнение внутреннего цикла прерывается. Но если пользователь вводит `must i?`, то выражение `cin` получает значение `false`, никакое значение не помещается в массив, выражение `!(cin >> golf[i])` оказывается истинным и программа переходит во внутренний цикл `while`. Первый оператор цикла использует метод `clear()` для переустановки результатов ввода. Если опустить этот оператор, программа откажется продолжить считывание вводимых значений. Затем программа использует функцию `cin.get()` в цикле `while` для считывания остальных вводимых данных до конца строки. Таким образом она избавляется от неверно вводимых данных и всех данных, находящихся в этой же строке. Еще один возможный подход — считывание следующего пробела, в результате чего программа избавлялась от результатов неверного ввода по одному слову, а не по одной строке. И наконец, программа предлагает пользователю ввести число.

## Резюме

Программы и процесс программирования становятся более интересными при использовании операторов, которые позволяют программе выбирать из нескольких вариантов действий. В качестве средств управления выбором вариантов в C++ используются операторы `if`, `if else` и `switch`. Оператор `if` языка C++ позволяет программе выполнять оператор или блок операторов условно, т.е. программа выполняет оператор или блок в случае удовлетворения определенного условия. Оператор `if else` позволяет программе выбирать для выполнения один из двух операторов или блоков операторов. Для реализации последовательности нескольких операций выбора к оператору можно добавлять дополнительные операторы `else if`. Оператор `switch` C++ предоставляет программе конкретный вариант из имеющихся в списке.

C++ предлагает вам также операторы, помогающие принять нужное решение. В главе 5 рассматриваются выражения сравнения, которые позволяют сравнить два значения. Используя логические операции C++ (`&&`, `||` и `!`), можно объединять или изменять выражения отно-

шения, создавая более сложные проверочные условия. Условный оператор `(?:)` обеспечивает компактный способ выбора одного из двух условий.

Библиотека символьных функций `cctype` обеспечивает удобный и мощный набор инструментальных средств для анализа результатов символьного ввода.

С помощью циклов и операторов принятия решений C++ программист может создавать интересные, инструментальные и мощные программы. Но пока мы только начали исследовать реальные возможности языка C++. Далее мы рассмотрим функции.

## Вопросы для повторения

1. Рассмотрите следующие программные фрагменты, в которых выполняется подсчет пробелов и новых строк:

```
// Версия 1
while (cin.get(ch)) // выход по достижению
 // символа eof
{
 if (ch == ' ')
 spaces++;
 if (ch == '\n')
 newlines++;
}
// Версия 2
while (cin.get(ch)) // выход по достижению
 // символа eof
{
 if (ch == ' ')
 spaces++;
 else if (ch == '\n')
 newlines++;
}
```

Какие преимущества, если таковые имеются, имеет вторая форма по сравнению с первой?

2. Какой эффект окажет замена `ch++` на `ch+1` в листинге 6.2?

3. Внимательно рассмотрите следующую программу:

```
#include <iostream>
using namespace std;
int main()
{
 char ch;
 int ct1, ct2;
 ct1 = ct2 = 0;
 while ((ch = cin.get()) != '$')
 {
 cout << ch;
 ct1++;
 if (ch == '$')
 ct2++;
 cout << ch;
 }
 cout << "ct1 = " << ct1
 << ", ct2 = " << ct2 << "\n";
 return 0;
}
```

Предположим, что вводится следующая последовательность символов, причем ↵ представляет нажатие клавиши Enter:

```
hi!↵
Send $10 or $20 now!↵
```

Каким будет результат выполнения программы?  
(Вспомните, что ввод буферизуется.)

4. Создайте логические выражения, представляющие следующие условия:

- a. **weight** больше или равно 115, но меньше 125.
- b. **ch** является **q** или **Q**.
- c. **x** является четным, но не **26**.
- d. **donation** находится в диапазоне 1000-2000 или **quest** является **1**.
- e. **ch** является строчной или прописной буквой (предполагается, что и строчные, и прописные буквы кодируются последовательно, но между кодами строчных и прописных букв имеется пробел).

5. В разговорной речи утверждение "Я не буду говорить" равносильно утверждению "Я буду говорить". Равнозначны ли в языке C++ выражения **!!x** и **x?**

6. Создайте условное выражение, которое равно абсолютному значению переменной. Иначе говоря, если значение переменной **x** положительно, значением выражения должно быть просто **x**, но, если значение **x** отрицательно, результатом будет положительное значение **-x**.

7. Перепишите следующий фрагмент, используя оператор **switch**:

```
if (ch == 'A')
 a_grade++;
else if (ch == 'B')
 b_grade++;
else if (ch == 'C')
 c_grade++;
else if (ch == 'D')
 d_grade++;
else
 f_grade++;
```

8. Какое преимущество дало бы использование в пунктах меню и вариантах перехода листинга 6.10 символьных меток типа **a** и **c** вместо номеров? (Подсказка: подумайте, что происходит, если в любом случае пользователь вводит **q**, и что происходит, если в любом случае он вводит **5**.)

9. Рассмотрите следующий фрагмент программы:

```
int line = 0;
char ch;
while (cin.get(ch))
{
 if (ch == 'Q')
 break;
 if (ch != '\n')
 continue;
 line++;
}
```

Перепишите этот фрагмент, не используя операторы **break** или **continue**.

## Упражнения по программированию

1. Напишите программу, которая считывает ввод с клавиатуры в символ @ и при выводе повторяет ввод, за исключением цифр, преобразуя каждый прописной символ в строчный и наоборот. (Не забудьте о семействе **cctype**.)
2. Напишите программу, которая считывает до десяти значений "пожертвований" в массив значений типа **double**. Программа должна прерывать ввод при вводе нечислового значения. Она должна сообщать о средней величине "пожертвований", а также о том, сколько значений в массиве превышают среднее значение.
3. Напишите заготовку программы, управляемой меню. Программа должна отображать меню, состоящее из четырех пунктов, каждый из которых помечен буквой. Если пользователь отвечает вводом буквы, отличающейся от одного из допустимых пунктов, программа должна предлагать ему ввести допустимый ответ до тех пор, пока это не будет сделано. Затем программа должна использовать переключатель для выбора простого действия, исходя из ответа пользователя. Результат выполнения программы мог бы выглядеть примерно так:
 

```
Please enter one of the following choices:
c) carnivore p) pianist
t) tree g) game
f
Please enter a c, p, t, or g: q
Please enter a c, p, t, or g: t
A maple is a tree.
```
4. Вступив в Благотворительный орден программистов (*Benevolent Order of Programmers*), вы можете быть известны на собраниях ВОР под своим действительным именем, по названию должности или по тайной кличке ВОР. Напишите программу, которая может перечислять членов ордена по действительным именам, по должностям, по тайным кличкам или по привилегиям. В основу программы положите следующую структуру:

```
// Структура имен членов Benevolent
// Order of Programmers
struct bop {
 char fullname[strsize]; //действительное
 //имя
 char title[strsize]; //название
 //должности
 char bopname[strsize]; //секретная
 //кличка BOP
 int preference; // 0 = fullname,
 // 1 = title,
 // 2 = bopname
}
```

В программе создайте небольшой массив таких структур и инициализируйте его подходящими значениями. Программа должна выполнять цикл, который позволяет пользователю выбирать различные возможности:

- a. отображение по имени
- b. отображение по должности
- c. отображение по тайной кличке
- d. отображение по привилегиям
- q. выход

Результат выполнения программы может выглядеть примерно так:

```
Benevolent Order of Programmers Report
a. display by name
b. display by title
c. display by bopname
d. display by preference
q. quit
```

```
Enter your choice: a
Wimp Macho
Raki Rhodes
Celia Laiter
Hoppy Hipman
Pat Hand
Next choice: d
Wimp Macho
Junior Programmer
MIPS
Analyst Trainee
LOOPY
Next choice: q
Bye!
```

5. В королевстве Нейтронии, в которой денежной единицей является tvarg, установлены следующие ставки подоходного налога:

первые 5000 tvarg: 0%  
 следующие 10000 tvarg: 10%  
 следующие 20000 tvarg: 15%  
 свыше 35000: 20%

Например, кто-либо получающий 38000 tvarg должен был бы уплатить налог величиной  $5000 \times 0.00 + 10000 \times 0.10 + 20000 \times 0.15 + 3000 \times 0.20$ , или 4600 tvarg. Напишите программу, которая использует цикл для запроса о доходах и для отображения суммы налога. Цикл прерывается, когда пользователь вводит отрицательное число или нечисловое значение.

# ФУНКЦИИ ЯЗЫКА C++

**В этой главе рассматривается следующее:**

- Функции (обзор)
- Прототипы функций
- Передача функциям аргументов по значению
- Разработка функций, предназначенных для обработки массивов
- Использование указателей со статусом `const` в качестве аргументов
- Проектирование функций для обработки текстовых строк
- Проектирование функций для обработки структур
- Функции, которые вызывают сами себя (рекурсия)
- Указатели на функции

Язык C++ поставляется с обширными библиотеками полезных функций (стандартная библиотека языка ANSI C плюс несколько классов языка C++), однако истинное удовольствие от программирования вы получите, написав свои собственные функции. В этой и следующей главах вы узнаете, как определять функции, как передавать им и получать от них информацию. Ознакомившись с тем, как работают функции, вы узнаете из этой главы, как следует использовать функции при работе с массивами, строками и структурами. И наконец, в этой главе состоится ваше первое знакомство с рекурсией и с указателями на функции. Если вы не пожалели усилий на изучение языка C, то многое из того, что содержится в данной главе, вам покажется знакомым. Однако не доверяйте возникающему ощущению, что вы уже все знаете. В языке C++ возможности функций существенно расширены, и в следующей главе речь пойдет главным образом об этом. Однако обратимся сначала к фундаментальным понятиям.

## Обзор функций

Прежде всего подведем итоги того, что мы уже знаем о функциях. Чтобы воспользоваться функцией в языке C++, необходимо выполнить следующее:

- Предусмотреть определение функции.
- Предусмотреть прототип функции.
- Вызвать функцию.

Если вы используете функцию из библиотеки, значит, эта функция уже определена и скомпилирована. Кроме того, можно воспользоваться стандартным библиотечным заголовочным файлом, чтобы получить поддержку прототипа. Теперь остается только корректно вызвать функцию. В примерах, приводимых в данной книге, это делается многократно. Например, в стандартной библиотеке языка C имеется функция `strlen()`, предназначенная для определения длины строки. Ассоциированный с ней стандартный заголовочный файл `cstring` содержит прототип функции `strlen()`, а также несколько других функций, предназначенных для обработки строк. Эта предварительно проделанная работа позволяет вам совершенно спокойно пользоваться в своих программах функцией `strlen()`.

Однако, когда вы создаете собственную функцию, необходимо уделить внимание всем трем аспектам — определению, разработке прототипов функции и вызову функции. При этом ответственность за принятие решений ложится исключительно на вас. В листинге 7.1 все эти аспекты проиллюстрированы коротким примером.

### Листинг 7.1 Программа calling.cpp.

```
// calling.cpp -- определение, создание
// прототипов функции и ее вызов
#include <iostream>
using namespace std;

void simple(); // прототип функции

int main()
{
 cout << "main() will call the simple()\n"
 "function:\n";
 simple(); // обращение к функции
 return 0;
}

// определение функции
void simple()
{
 cout << "I'm but a simple function.\n";
}
```

Результаты выполнения программы:

```
main() will call the simple() function:
I'm but a simple function.
```

Теперь приступим к более подробному изучению функций.

## Определение функции

Функции можно сгруппировать в две следующие категории: функции, которые не имеют возвращаемых значений, и функции, которые имеют возвращаемые значения. Функции без возвращаемых значений называются функциями типа **void**, они имеют следующую общую форму:

```
void имяфункции(списокАргументов)
{
 оператор(ы)
 return; // необязательная конструкция
}
```

Здесь параметр **списокАргументов** описывает типы и число аргументов, передаваемых функции. Этот список рассматривается далее в настоящей главе более подробно. Необязательный оператор возврата завершает функцию. В противном случае тело функции оканчивается замыкающей скобкой. Функции категории **void** соответствуют процедурам в языке Pascal, подпрограммам в языке FORTRAN и подпрограммам современного языка программирования BASIC. Обычно функция **void** используется для того, чтобы выполнить какое-то действие. Например, функция, в задачу которой входит напечатать приветствие *Cheers!* заданное число (*n*) раз имеет следующий вид:

```
void cheers(int n) // возвращаемое
 // значение отсутствует
```

```
{
 for (int i = 0; i < n; i++)
 cout << "Cheers!";
 cout << "\n";
}
```

Список аргументов **int n** означает, что при обращении к функции *cheers()* ей необходимо передать в качестве аргумента значение типа **int**.

Функция с возвращаемым значением вычисляет значение, которое она возвращает вызывающей ее функции. Другими словами, если функция возвращает квадратный корень из числа 9.0 (**sqrt(9.0)**), то эта функция возвращает в качестве результата число 3.0. Считается, что такая функция имеет тот же тип, что и значение, которое она возвращает. Общая форма такой функции имеет вид:

```
имятипа имяфункции(списокАргументов)
{
 операторы
 return значение; // тип возвращаемого
 // значения определяется типом функции
}
```

Функции с возвращаемыми значениями требуют использования оператора возврата, обеспечивающего возврат полученного значения вызывающей функции. Само значение может быть константой, переменной или выражением более общего типа. К нему предъявляется единственное требование — чтобы это выражение приводилось к значению, которое имеет тип **имяТипа** или могло быть преобразовано к этому типу. (Если объявленным типом возвращаемого значения является, скажем, **double**, а функция возвращает выражение типа **int**, то значение типа **int** преобразуется к типу **double**.) Затем функция возвращает окончательное значение той функции, которая ее вызвала. Язык C++ накладывает ограничение на типы, которые могут быть использованы для возвращаемого значения: массив не может быть возвращаемым значением. Все остальное годится для применения — целые числа, числа с плавающей точкой, даже структуры и объекты! (Интересно отметить, что, хотя функция в C++ не может возвратить массив непосредственно, она может возвратить его как часть структуры или объекта.)

Как программисту вам не обязательно знать, каким образом функция возвращает значение, однако знание соответствующего метода может помочь получить четкое представление об идее, положенной в его основу (а также дать пищу для размышлений вашим друзьям или членам семьи). Как правило, функция возвращает значение путем копирования этого значения в специальный регистр центрального процессора или в специально отведенную для этой цели ячейку памяти. Затем вызывающая программа обращается к этой ячейке. Как возвращающая так и вызывающая функция должны прийти к со-

глашению относительно типа данных, хранящихся в этой ячейке. Прототип функции сообщает вызывающей программе, что ей следует ожидать, а определение вызываемой функции сообщает вызываемой программе, что ей следует возвратить (рис. 7.1). Размещение одних и тех же данных в прототипе функции и в ее определении может на первый взгляд показаться ненужным дублированием информации, однако в этом есть определенный смысл. Так, если вы хотите, чтобы курьер взял что-то с вашего стола в офисе, то вы существенно повысите вероятность того, что задача будет выполнена правильно, если оставите описание того, что именно он должен взять.

Выполнение функции завершается после выполнения оператора возврата. Если функция включает более одного оператора возврата, например в качестве альтернатив выбора в операторе `if else`, выполнение функции прекращается сразу после того, как она выполнит первый встреченный оператор возврата.

```
int bigger(int a, int b)
{
 if (a > b)
 return a; // если a > b, выполнение
 // функции завершается здесь
 else
 return b; // иначе выполнение
 // функции завершается здесь
}
```

В рассматриваемом случае оператор `else` не нужен, однако он помогает непредубежденному читателю понять суть дела.

Функции с возвращаемыми значениями во многом похожи на функции языков программирования Pascal, FORTRAN и BASIC. Они возвращают значения вызывающей программе, которая затем может присвоить это значение какой-нибудь переменной, отобразить это значение на экране или использовать ее каким-либо дру-

гим способом. Далее рассматривается простой пример, возвращающий значение куба числа типа `double`:

```
double cube(double x) // x умножить на x
 // умножить на x
{
 return x * x * x; // значение типа
 // double
}
```

Например, в результате вызова функции `cube(1.2)` возвращается значение 1.728. Обратите внимание на то обстоятельство, что этот оператор использует выражение. Эта функция вычисляет значение выражения (в данном случае 1.728) и возвращает его.

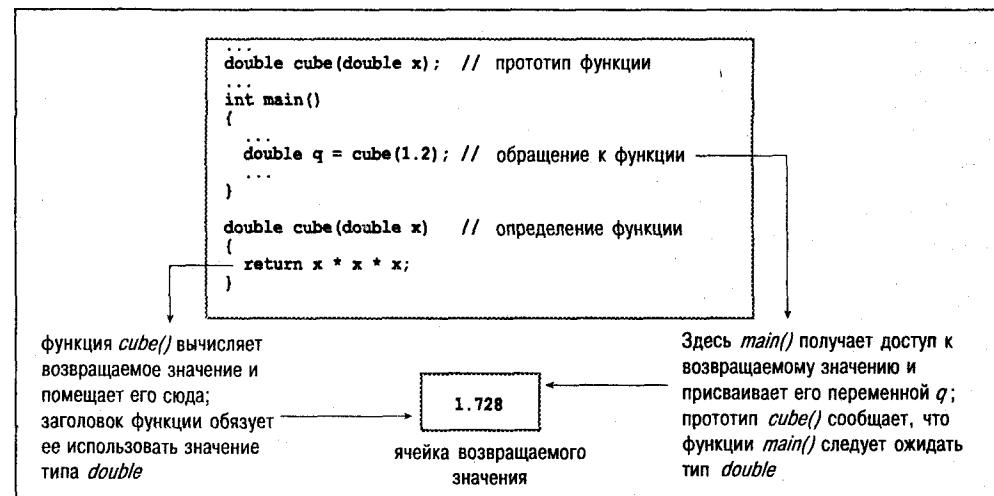
## Прототипирование и вызов функций

На данный момент вы имеете представление только о том, как обращаться к функциям (вызывать функции), но значительно меньше знакомы с тем, как выполнять прототипирование функций, поскольку эти задачи выполняли файлы включения. Попробуем воспользоваться функциями `cheers()` и `cube()` в программе (листинг 7.2); обратите внимание на прототипы этих функций.

### Листинг 7.2 Программа protos.cpp.

```
// protos.cpp -- использование прототипов и
// вызовов функций
#include <iostream>
using namespace std;
void cheers(int); // прототип: возвращаемое
 // значение отсутствует
double cube(double x); // прототип: возвращает
 // значение типа double
int main(void)
{
 cheers(5); // вызов функции
 cout << "Give me a number: ";
 double side;
 cin >> side;
```

**РИСУНОК 7.1**  
Типичный механизм  
возврата значений.



```

double volume = cube(side); // вызов
 // функции
cout << "A " << side << "-foot cube has a
 ↪volume of ";
cout << volume << " cubic feet.\n";
cheers(cube(2)); // защита прототипа в
 // действии
return 0;
}

void cheers(int n)
{
 for (int i = 0; i < n; i++)
 cout << "Cheers! ";
 cout << "\n";
}

double cube(double x)
{
 return x * x * x;
}

```

Результаты выполнения программы:

```

Cheers! Cheers! Cheers! Cheers! Cheers!
Give me a number: 5
A 5-foot cube has a volume of 125 cubic
feet.
Cheers! Cheers! Cheers! Cheers! Cheers!
 Cheers! Cheers! Cheers!

```

Обратите внимание на то, что `main()` обращается к функции `cheers()` типа `void`, используя для этой цели имя функции и аргументы, за которыми следует точка с запятой: `cheers(5);`. Это пример оператора обращения (или вызова) к функции. В то же время, поскольку функция `cube()` имеет возвращаемое значение, `main()` может использовать ее в качестве оператора присваивания:

```
double volume = cube(side);
```

Но, как уже было сказано выше, необходимо сосредоточиться на изучении прототипов. Что вам следует знать о прототипах? Прежде всего, необходимо понимать, почему в языке C++ возникла необходимость в прототипах. Далее, поскольку в C++ имеется необходимость в прототипах, нужно уметь правильно пользоваться соответствующим синтаксисом. И наконец, необходимо четко представлять себе, какие преимущества предоставляет прототип. Рассмотрим все эти вопросы по очереди на примере листинга 7.2.

### Почему именно прототипы?

Прототип описывает интерфейс функции при работе с компилятором. Другими словами, он уведомляет компьютер о том, какой тип должно иметь возвращаемое значение функции, если таковое имеется, и сообщает компилятору, сколько функция имеет аргументов и какого типа. Рассмотрим, например, какое влияние оказывает

прототип на вызов функции в программе, представленной в листинге 7.2:

```
double volume = cube(side);
```

Во-первых, прототип сообщает компилятору, что функция `cube()` должна иметь один аргумент типа `double`. Если программа не сможет обеспечить передачу такого аргумента, прототипирование позволяет компилятору зафиксировать соответствующую ошибку. Во-вторых, когда функция `cube()` закончит свои вычисления, она поместит возвращаемое значение в специально отведенное для этого место, возможно, в какой-либо регистр центрального процессора или в ячейку оперативной памяти. Далее вызывающая функция (в рассматриваемом случае это `main()`) находит его в этом месте. Поскольку прототип утверждает, что `cube()` имеет тип `double`, компилятор знает, сколько байтов нужно выбрать и как их интерпретировать. Без этой информации компилятору остается только догадываться обо всем этом.

Вы, возможно, все еще не убеждены в необходимости использования компилятором прототипа. Разве он не может заглянуть глубже в соответствующий файл и изучить определение функции? Одним из недостатков такого подхода является его недостаточная эффективность. Компилятор должен приостановить компиляцию функции `main()` на время просмотра остальной части файла. Еще более серьезные проблемы возникают в том случае, когда функции в этом файле нет. Язык C++ позволяет вам разместить программу в нескольких файлах, которые можно компилировать независимо друг от друга, чтобы объединить их в единый модуль позднее. В таком случае компилятор может потерять доступ к программному коду функции во время компиляции функции `main()`. То же самое может случиться, если функция является частью некоторой библиотеки. Единственный способ, позволяющий избежать использования прототипа, состоит в том, чтобы поместить определение функции в таком месте программы, где оно будет предшествовать первому вызову функции. Однако это не всегда возможно. Кроме того, стиль программирования, принятый в C++, требует, чтобы функция `main()` всегда была первой, ибо она в общем случае определяет структуру всей программы.

### Синтаксис прототипа

Прототип функции является оператором, поэтому за ним должна следовать точка с запятой. Простейший способ получения прототипа заключается в том, что из определения функции копируется ее заголовок и добавляется точка с запятой. Именно эти действия и выполняет программа для функции `cube()`:

```
double cube(double x); // добавить точку с
// запятой, чтобы получить прототип
```

Однако прототип функции не требует, чтобы вы снабдили переменные именами, списка типов вполне достаточно. Программа создает прототип функции `cheers()`, используя для этой цели только тип аргумента:

```
void cheers(int); // в прототипе имена
// переменных игнорируются
```

В общем случае вы можете как включать имена переменных в список аргументов для прототипа, так и исключать их из этого списка. Имена переменных в прототипе проявляют себя как заполнители, они вовсе не должны совпадать с именами, содержащимися в определении функции.

### ПРОТОТИПИРОВАНИЕ В ЯЗЫКАХ C++ И ANSI C

Язык ANSI C позаимствовал прототипирование из C++, но эти языки несколько отличаются друг от друга. Основное отличие состоит в том, что язык ANSI C (в целях обеспечения совместимости с классическим языком C) рассматривает прототипирование как необязательное программное средство, в то время как в C++ прототипирование обязательно. В качестве примера рассмотрим объявление функции:

```
void say_hi();
```

В языке C++ пустое пространство, заключенное в круглые скобки, означает то же самое, что и ключевое слово `void` в круглых скобках: функция не имеет аргументов. В языке ANSI C отсутствие параметров в круглых скобках означает, что вы отказываетесь объявлять, какие аргументы использует функция. Другими словами, в программе прототипирование предшествует списку аргументов. В C++ отсутствие идентификатора списка аргументов эквивалентно использованию многоточия:

```
void say_bye(...); // отказ от ответственности
// со стороны языка C++
```

Обычно это необходимо разве что для интерфейса с функциями языка C, имеющими переменное число аргументов.

### Польза прототипов

Вы уже видели, что прототипы облегчают работу компиляторов. Но какую пользу они приносят вам? Они существенно уменьшают вероятность ошибок. В частности, прототипы помогают компилятору в следующем:

- Правильно обращаться с возвращаемым значением функции.
- Следить за тем, чтобы использовалось правильное число аргументов функции.
- Контролировать, чтобы применялись корректные типы аргументов функции. Если при этом возникают ошибки, компилятор по возможности приводит аргументы к правильному типу.

Мы уже говорили о том, как правильно использовать возвращаемое значение. Рассмотрим теперь, что случится, если вы неправильно указали число аргументов. Например, предположим, что вы обратились к функции следующим образом:

```
double z = cube();
```

В условиях, когда не выполняется прототипирование, компилятор не находит здесь ошибки. При вызове функции он определяет, куда в результате вызова функции `cube()` помещается число и при этом передает туда значение, какое бы оно ни было. Так в принципе и работал C до тех пор, пока ANSI C не позаимствовал прототипирование из C++. В связи с тем что в ANSI C прототипирование не является обязательным, некоторые программы, написанные на C, спроектированы согласно старой идеологии. Но в C++ прототипирование обязательно, так что в этом случае вам гарантирована защита от подобного рода ошибок.

Далее предположим, что используется некоторый аргумент, однако для него выбран неподходящий тип. В языке C это повлечет за собой фатальные ошибки. Например, если функция ожидает значения типа `int` (предположим, что это 16-разрядное значение), а вы передаете ей значение типа `double` (предположим также, что это 64-разрядное значение), то функция воспринимает только 16 первых разрядов из 64 и делает попытку интерпретировать их как значение типа `int`. Однако C++ автоматически приводит значение, которое вы передали, к типу, указанному в прототипе, при условии, что оба они являются арифметическими типами. Например, программа, представленная в листинге 7.2, способна преодолеть два несоответствия типов в одном операторе:

```
cheers(cube(2));
```

Сначала программа передает значение типа `int`, равное 2, функции `cube()`, которая ожидает получить величину типа `double`. Компилятор, убедившись в том, что прототип функции `cube()` описывает аргумент типа `double`, преобразует 2 в 2.0, т.е. в значение типа `double`. Затем функция `cube()` возвращает значение типа `double` (8.0), чтобы оно затем было использовано в качестве аргумента функции `cheers()`. И снова компилятор обращается к прототипам и обнаруживает, что функция `cheers()` запрашивает тип `int`. Он преобразует возвращаемое значение в целое число 8. В общем случае прототипирование обеспечивает автоматическое приведение величин к заданному типу. (Избыточное число параметров в функции (см. главу 8) может вызвать ситуации неопределенности, когда автоматическое приведение типов становится невозможным.)

Автоматическое преобразование типов отнюдь не препятствует возникновению всех возможных ошибок. Например, если вы передаете значение 8.33E27 функции, которая ожидает получить величину типа `int`, то такое большое значение просто не может быть правильно преобразовано в величину типа `int`. Некоторые компиляторы предупреждают о возможной потере данных, когда выполняется автоматическое преобразование большего типа к меньшему.

Наряду с этим прототипирование позволяет осуществлять преобразование типов только в тех случаях, когда это имеет смысл. Например, не выполняется преобразование целого числа в структуру или указатель.

Прототипирование осуществляется во время компиляции и относится к категории *статического контроля типа*. Статический контроль типа, как мы только что смогли убедиться, позволяет обнаружить многие ошибки из числа тех, которые намного труднее выявить во время выполнения программы.

## Аргументы функции и передача по значению

Теперь настало время уделить изучению аргументов функции больше внимания. Обычно в языке C++ аргументы передаются по значению, т.е. числовое значение аргумента передается функции, где оно присваивается новой переменной. Например, в программе, представленной в листинге 7.2, продемонстрировано обращение к функции:

```
double volume = cube(side);
```

Здесь `side` — это переменная, которая в процессе выполнения программы получает значение 5.

Напомним, что заголовок функции `cube()` был таким:

```
double cube(double x)
```

Эта функция при обращении к ней создает новую переменную типа `double` с именем `x` и присваивает ей

значение 5. Благодаря этому приему данные в функции `main()` изолируются от действий, которые происходят в функции `cube()`, так как `cube()` использует в своей работе всего лишь копию переменной `side`, а не саму эту переменную. Вы вскоре ознакомитесь с примером такого рода защиты. Переменная, которая используется для приема передаваемого значения, называется *формальным аргументом* или *параметром*. Величина, передаваемая функции, называется *фактическим аргументом* или *параметром*. Следовательно, передача аргумента представляет собой присвоение фактического аргумента формальному аргументу (рис. 7.2).

Переменные, в том числе и формальные параметры, объявленные внутри какой-либо функции, являются приватными по отношению к этой функции. Во время вызова функции компьютер выделяет память, необходимую для этих переменных. Когда выполнение функции прекращается, компьютер высвобождает память, которая была отведена этим переменным. (В некоторых книгах по C++ выделение памяти под эти переменные и освобождение ее рассматривается как создание и уничтожение переменных. Это звучит более драматично.) Такие переменные называются *локальными*, поскольку сфера их действия не выходит за пределы функции. Как мы уже отмечали, это позволяет сохранить целостность данных. Это также означает, что если вы объявляете переменную с именем `x` в функции `main()` и еще одну переменную с именем `x` в какой-либо другой функции, то это абсолютно разные, ничем не связанные между собой переменные, подобно тому как город Олбани в Калифорнии отличается от города с таким же названием в штате Нью-Йорк (рис. 7.3).

## Функции с несколькими аргументами

Функции могут иметь не один, а несколько аргументов. В обращении к такой функции аргументы отделяйте друг от друга запятыми:

```
n_chars('R', 25);
```

РИСУНОК 7.2 Передача по значению.



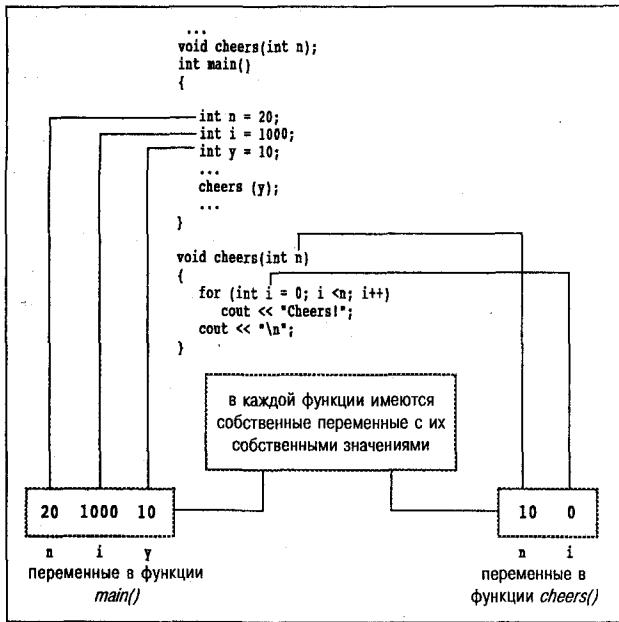


РИСУНОК 7.3 Локальные переменные.

В результате вызова функции `n_chars()`, определение которой будет приведено далее, ей передаются два аргумента.

Аналогично при определении функции вы используете в заголовке функции список объявлений, отделенных друг от друга запятыми:

```
void n_chars(char c, int n) // два
 // аргумента
```

Этот заголовок функции свидетельствует о том, что функция `n_chars()` принимает один аргумент типа `char` и один аргумент типа `int`. Переменным `c` и `n` присваиваются значения, передаваемые функции. Если функция получает два аргумента одного и того же типа, необходимо описать тип каждого аргумента отдельно. Вы не можете использовать сочетания описаний типов аргументов так, как это принято при объявлении переменных.

```
void fifi(float a, float b) // каждая
 // переменная должна быть описана отдельно
void fufu(float a, b) // такое описание
 // НЕПРАВИЛЬНО
```

Как и в случае с другими функциями, чтобы получить прототип, добавьте точку с запятой:

```
void n_chars(char c, int n); // прототип,
 // стиль 1
```

Как и в случае с использованием одного аргумента, не следует применять в прототипе те же имена переменных, которые были использованы в определении функ-

ции, вы можете просто пропустить имена переменных в прототипе:

```
void n_chars(char, int); // прототип,
 // стиль 2
```

Тем не менее, присвоение имен переменным помогает сделать прототип более понятным, особенно если два параметра имеют один и тот же тип. Кроме того, имена могут служить вам напоминанием, для чего предназначены тот или иной аргумент:

```
double melon_density(double weight,
 double volume);
```

Листинг 7.3 представляет собой пример реализации функции с двумя аргументами. Он является иллюстрацией того, что изменение значения формального аргумента функции не влияет на данные в вызывающей программе.

### Листинг 7.3 Программа twoarg.cpp.

```
// twoarg.cpp -- функция с двумя аргументами
#include <iostream>
using namespace std;
void n_chars(char, int);
int main()
{
 int times;
 char ch;

 cout << "Enter a character: ";
 cin >> ch;
 while (ch != 'q') // q для выхода из
 // программы
 {
 cout << "Enter an integer: ";
 cin >> times;
 n_chars(ch, times); // функция с
 // двумя аргументами
 cout << "\nEnter another character
 or press the"
 " q-key to quit: ";
 cin >> ch;
 }
 cout << "The value of times is " <<
 times << ".\n";
 cout << "Bye\n";
 return 0;
}

void n_chars(char c, int n) // Отображает с n
 // раз
{
 while (n-- > 0) // продолжается, пока n
 // не станет равным 0
 cout << c;
}
```

Ниже приводятся результаты выполнения этого примера:

```
Enter a character: W
Enter an integer: 50
```

```
Enter another character or press the q-key
to quit: a
Enter an integer: 20
aaaaaaaaaaaaaaaaaa
Enter another character or press the q-key
to quit: q
The value of times is 20.
Bye
```

### Примечания к программе

Функция `main()` использует цикл `while`, чтобы вы не забывали об этих конструкциях. Обратите внимание на тот факт, что функция, чтобы прочитать символ, использует конструкцию `cin >> ch`, а не `cin.get(ch)` или `cin = cin.get()`. Для этого есть веские причины. Пара функций `cin.get()`, как вам уже известно, читает все вводимые символы, включая пробелы и символы новой строки, в то время как конструкция `cin >>` пропускает пробелы и символы новой строки. Отвечая на приглашение программы, необходимо нажимать клавишу `Enter` в конце каждой строки, в результате чего генерируется очередной символ новой строки. Конструкция `cin >> ch` позволяет игнорировать символы новой строки, однако родственная ей конструкция `cin.get()` читает символы новой строки, которые следуют за каждым вводимым числом, и отображает их. Вы можете обойти это неудобство, но гораздо проще воспользоваться конструкцией `cin`, что, собственно говоря, и делает программа.

Функция `n_chars()` требует двух аргументов, которыми являются символ `c` и целое число `n`. Затем она использует цикл, чтобы выводить на экран заданный символ столько раз, сколько задается этим целым числом.

```
while (n-- > 0) // продолжается, пока n
 // не станет равным 0
 cout << c;
```

Обратите внимание на то обстоятельство, что программа ведет счет вводимых символов, уменьшая значение переменной `n`, где `n` — формальный параметр из списка аргументов. Этой переменной присваивается значение переменной `times` в функции `main()`. Значение переменной `n` уменьшается в цикле `while` до нуля, однако, как показывает выполнение рассматриваемого примера, изменение величины `n` не оказывает никакого влияния на переменную `times`.

### Еще одна функция с двумя аргументами

Построим более показательную функцию, такую, которая выполняет нетривиальные вычисления. Эта функция проиллюстрирует использование локальных переменных, отличных от функциональных формальных переменных.

Многие штаты в США организуют лотерею, придавая ей ту или иную форму игры Лото. Эта игра заклю-

чается в том, что вы произвольно выбираете некоторое число из карточки. Например, вы выбрали шесть номеров из карточки, содержащей 51 номер. Затем ведущие игры Лото случайным образом выбирают шесть номеров. Если ваш выбор точно совпадает с их выбором, то вы выигрываете сразу несколько миллионов или что-то вроде этого. Наша функция вычисляет вероятность того, что будет выбрано именно выигрышное сочетание номеров. (Разумеется, от функции, которая предсказывает выигрышное сочетание номеров, было бы куда больше пользы, однако языку C++ при всем его теперешнем могуществе и совершенстве эта задача пока не под силу.)

Прежде всего нам нужна подходящая формула. Предположим, что требуется выбрать шесть значений из 51. Математики утверждают, что у вас есть R шансов на выигрыш, при этом R вычисляется по формуле:

$$R = \frac{51 \times 50 \times 49 \times 48 \times 47 \times 46}{6 \times 5 \times 4 \times 3 \times 2 \times 1}$$

Для шести выборок знаменатель представляет собой произведение шести первых натуральных чисел, или шесть!. Числитель также представляет собой произведение шести последовательных чисел, которые на этот раз начинаются с 51 и следуют в порядке уменьшения. В более общей форме это звучит так: если вы производите picks выборок из numbers номеров, то знаменатель равен picks!, а числитель есть произведение picks целых чисел, которые начинаются со значения numbers и следуют в порядке уменьшения на единицу. Чтобы выполнить соответствующие вычисления, можно воспользоваться циклом `for`:

```
long double result = 1.0;
for (n = numbers, p = picks; p > 0; n--, p--)
 result = result * n / p;
```

Вместо того чтобы сначала перемножить значения в числителе, цикл начинается с того, что 1.0 умножается на первое значение в числителе, а затем делится на первое значение в знаменателе. На следующем этапе цикл выполняет умножение и деление на вторые значения соответственно числителя и знаменателя. Благодаря такому порядку выполнения операций текущее произведение всегда меньше, чем если бы вы сначала выполнили все операции умножения. Сравните, например,

$(10 * 9) / (2 * 1)$

со следующим выражением:

$(10 / 2) * (9 / 1)$

В первом случае сначала получается 90, а затем 45, в то время как во втором случае сначала получается 5 \* 9, а затем 45. Оба дают один и тот же результат, одна-

ко в условиях первого метода получаем больший промежуточный результат (90) по сравнению со вторым методом. Чем больше сомножителей, тем больше эта разница. При больших числах метод чередования умножения и деления позволяет избежать превышения максимально допустимого значения с плавающей точкой.

В программе, представленной в листинге 7.4, эта формула используется в функции `odds()`. Поскольку число выборок и общее число исходов — это положительные числа, программа использует для этих величин тип `unsigned int` (для краткости — `unsigned`). При умножении нескольких целых чисел сразу можно получить очень большую величину произведения, поэтому в программе `lotto.cpp` для представления возвращаемого значения используется тип `long double`. Кроме того, такие операции, как `49 / 6`, привносят в значения целого типа ошибку округления.

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторых версиях языка C++ тип `long double` не поддерживается. Если версия C++, с которой вы работаете, принадлежит этой категории, используйте обычный тип `double` для этой цели.

#### Листинг 7.4 Программа `lotto.cpp`.

```
// lotto.cpp -- вероятность упустить выигрыш
#include <iostream>
using namespace std;
// Примечание. Некоторые версии C++ требуют
// применения типа double вместо long double
long double odds(unsigned numbers,
 unsigned picks);
int main()
{
 double total, choices;
 cout << "Enter total number of game card
 choices and\n"
 "number of picks allowed:\n";
 while ((cin >> total >> choices) &&
 choices <= total)
 {
 cout << "You have one chance in ";
 cout << odds(total, choices);
 // вычислите вероятность
 cout << " of winning.\n";
 cout << "Next two numbers (q to
 quit): ";
 }
 cout << "bye\n";
 return 0;
}

// представленная ниже функция вычисляет
// вероятность выбора выигрышного сочетания
// номеров
long double odds(unsigned numbers,
 unsigned picks)
{
 long double result = 1.0;
```

```
// далее следуют несколько локальных переменных
long double n;
unsigned p;

for (n = numbers, p = picks; p > 0; n--, p--)
 result = result * n / p;
return result;
}
```

Ниже приведены результаты выполнения программы. Обратите внимание на то обстоятельство, что увеличение числа номеров в игровой карточке стремительно уменьшает шансы выигрыша.

```
Enter total number of game card choices and
number of picks allowed:
49 6
You have one chance in 1.39838e+007 of
winning.
Next two numbers (q to quit): 51 6
You have one chance in 1.80095e+007 of
winning.
Next two numbers (q to quit): 38 6
You have one chance in 2.76068e+006 of
winning.
Next two numbers (q to quit): q
bye
```

#### Примечания к программе

Функция `odds()` иллюстрирует применение двух возможных видов локальных переменных. Так, в ней используются формальные параметры (`numbers` и `picks`), которые объявлены в заголовке функции, предшествующем открывающей фигурной скобке. Далее идут локальные переменные другого типа (`result`, `n` и `p`). Они объявлены внутри фигурных скобок, ограничивающих определение функции. Главное отличие между формальными параметрами и другим типом локальных переменных заключается в том, что формальные параметры получают свои значения из обращения к этой функции, имеющего вид `odds()`, в то время как другие переменные получают значения из самой функции.

#### ФУНКЦИИ И МАССИВЫ

До сих пор примеры рассматриваемых нами функций были простыми, в них в качестве аргументов возвращаемых величин использовались только основные типы данных. Но в то же время функции могут быть ключевым средством для использования более сложных типов, таких как массивы и структуры. Теперь рассмотрим, как массивы и функции "уживаются" друг с другом.

Предположим, что вы используете массив, чтобы проследить, сколько булочек съел каждый участник пикника. (Каждый индекс массива соответствует конкретному участнику, а значение элемента массива показывает число съеденных этим участником булочек.) Теперь

вы хотите знать, сколько булочек было съедено всеми участниками. Это легко сделать, достаточно воспользоваться циклом, чтобы просуммировать все элементы массива. Но сложение элементов массива — настолько часто встречающаяся операция, что имеет смысл разработать специальную функцию, способную выполнить эту процедуру. В таком случае у вас не будет необходимости каждый раз приспособливать цикл для суммирования элементов массивов.

Рассмотрим, каким должен быть интерфейс функции. Ввиду того что функция вычисляет сумму, она должна возвращать ответ. Если ваши булочки остаются нетронутыми, можно употреблять для описания этой ситуации функцию с возвращаемым значением типа `int`. Следовательно, чтобы эта функция знала, какой массив подлежит суммированию, необходимо передать ей имя массива в качестве аргумента. И чтобы придать этой функции большую универсальность, ей следует передать размеры массива. Единственной новой составляющей в этом случае является то, что вы должны каким-то образом объявить, что одним из формальных параметров является имя массива. Теперь выясним, как должны выглядеть эта, а также все остальные части заголовка функции:

```
int sum_arr(int arr[], int n) // arr = имя
 // массива, n = размер массива
```

Это выглядит вполне реально. Квадратные скобки указывают на то, что `arr` — это массив, а тот факт, что скобки пусты, свидетельствует о том, что вы можете использовать эту функцию с массивом любых размеров. Однако не всегда обстоятельства являются такими, какими они нам кажутся на первый взгляд: `arr` вовсе не массив, это указатель! Хорошо уже то, что вы можете задавать остальную часть кода функции, как если бы переменная `arr` была массивом. Сначала посмотрим, является ли этот подход действенным, а затем выясним, в чем его суть.

Листинг 7.5 иллюстрирует использование указателя, как если бы это было имя массива. Программа инициализирует массив с некоторыми значениями и использует функцию `sum_arr()` для вычисления суммы. Обратите внимание на тот факт, что функция `sum_arr()` использует `arr`, как если бы это было имя массива.

### Листинг 7.5 Программа arrfun1.cpp.

```
// arrfun1.cpp -- функции с аргументом типа
// array
#include <iostream>
using namespace std;
const int ArSize = 8;
int sum_arr(int arr[], int n); // прототип
int main()
{
```

```
 int cookies[ArSize] =
 {1, 2, 4, 8, 16, 32, 64, 128};
 // некоторые системы требуют, чтобы int
 // предшествовало ключевое слово static,
 // благодаря чему становится возможной
 // инициализация массива

 int sum = sum_arr(cookies, ArSize);
 cout << "Total cookies eaten: " << sum
 << endl;
 return 0;
}

// возвращает сумму элементов массива целых
// чисел
int sum_arr(int arr[], int n)
{
 int total = 0;

 for (int i = 0; i < n; i++)
 total = total + arr[i];
 return total;
}
```

Результаты выполнения программы:

```
Total cookies eaten: 255
```

Как вы сами можете убедиться, программа работает. Теперь узнаем, почему она работает.

### Массивы и указатели (продолжение)

Суть заключается в том, что язык C++, подобно C, в большинстве контекстов рассматривает имя массива так, как если бы это был указатель. К этому можно добавить, что C++ интерпретирует имя массива как адрес его первого элемента:

```
cookies == &cookies[0] // именем массива
 // является адрес его первого элемента
```

(Имеются два исключения из этого правила. Первое — при объявлении массива его имя используется для того, чтобы пометить соответствующую область памяти. Второе — если применить оператор `sizeof` к имени массива, то в результате получим размер всего массива в байтах.)

Программа, представленная в листинге 7.5, выполняет вызов функции:

```
int sum = sum_arr(cookies, ArSize);
```

Здесь `cookies` — это имя массива, поэтому в соответствии с соглашениями, принятыми в C++, `cookies` — это адрес его первого элемента. Функция передает адрес. Поскольку элементы массива имеют тип `int`, массив `cookies` должен иметь тип указатель-на-`int`, или тип `int*`. Отсюда следует, что правильный заголовок функции должен быть таким:

```
int sum_arr(int * arr, int n) // arr = имя
 // массива, n = размер массива
```

Здесь `int *arr` используется вместо `int arr[]`. оказывается, что оба заголовка правильны, так как в C++ обозначения `int *arr` и `int arr[]` имеют одно и то же значение тогда и только тогда, когда они используются в заголовке функции или в прототипе функции. Обе записи означают, что `arr` представляет собой указатель на `int`. Следует отметить, что обозначение массива вида (`int arr[]`) в символической форме напоминает нам, что `arr` указывает не только на `int`, но и на первый элемент `int` в массиве элементов `int`. Мы используем обозначение массива, когда указатель ссылается на первый элемент массива, и обозначение указателя, когда указатель ссылается на изолированное значение. Помните, что обозначения `int *arr` и `int arr[]` не идентичны во всех других контекстах. Например, вы не можете воспользоваться обозначением `int tip[]` для объявления указателя в теле функции.

Остальная часть функции имеет смысл при условии, что переменная `arr` фактически является указателем. Как уже говорилось в главе 4, в которой рассматривались динамические массивы, вы с таким же успехом можете использовать обозначение "массив со скобками", именем массива, и указателями для получения доступа к элементам массива. Независимо от того, является ли `arr` указателем или именем массива, выражение `arr[3]` обозначает четвертый элемент массива. И, по-видимому, будет полезно напомнить сейчас о том, что имеют место следующие два тождества:

```
arr[i] == *(arr + i) // два обозначения
 // одной и той же величины
&arr[i] == arr + i // два обозначения
 // одного и того же адреса
```

Помните, что добавление `I` к указателю, содержащему имя массива, фактически означает добавление величины, равной размеру в байтах типа значения, на которое ссылается указатель. Увеличение значения указателя и изменение индексов элементов массива — это два эквивалентных способа отсчета элементов от начала массива.

## Трудности, возникающие при использовании массивов в качестве аргументов

Давайте посмотрим, с какими трудностями сталкивается программа, представленная в листинге 7.5. В результате вызова функции `sum_arr(cookies, ArSize)` передается адрес первого элемента массива `cookies` и число элементов этого массива функции `sum_arr()`. Функция `sum_arr()` присваивает адрес массива `cookies` переменной `arr` типа `int` и значение `ArSize` — переменной `n` типа `int`. Это значит, что программа из листинга 7.5 на самом деле не передает указанной функции содержимое

массива. Вместо этого она сообщает функции, где находится массив (т.е. его адрес), что представляют собой его элементы (т.е. их тип) и сколько таких элементов содержит массив (переменная `n`) (рис. 7.4). Вооруженная этой информацией, функция далее работает с исходным массивом. Передайте функции обычную переменную, и она будет работать с ее копией, но если функции передать массив, то она будет работать с его оригиналом. Но, по сути дела, такое различие не компрометирует подход "передавать по значению", принятый в языке C++. Функция `sum_arr()` фактически передает значение, которое затем присваивается новой переменной. Однако это значение является одиночным адресом, а не содержимым массива.

Приносит ли соответствие между именами массивов и указателями какую-либо пользу? Приносит, и немалую. Дизайнерское решение, предусматривающее использование адресов массивов в качестве аргументов позволяет экономить время и память, необходимые для получения копии всего массива. Дополнительные расходы ресурсов для получения копий массивов могут оказаться недопустимо высокими, если вы работаете с большими массивами. Программа потребует не только дополнительных объемов оперативной памяти, ей придется потратить время для копирования больших блоков данных. С другой стороны, работа с исходными данными повышает вероятность непреднамеренного искажения данных. Эта проблема стоит достаточно остро в классическом C, однако в ANSI C и C++ спецификатор `const` обеспечивает соответствующую защиту. Мы вскоре получим возможность в этом убедиться. Но сначала внесем в листинг 7.5 изменения, которые позволят нам проиллюстрировать особенности функций, работающих с массивами. В листинге 7.6 показана ситуация, когда массивы `cookies` и `arr` имеют одно и то же значение. Он также демонстрирует, что понятие указателя придает функции `sum_arr` большую гибкость, чем может показаться на первый взгляд.

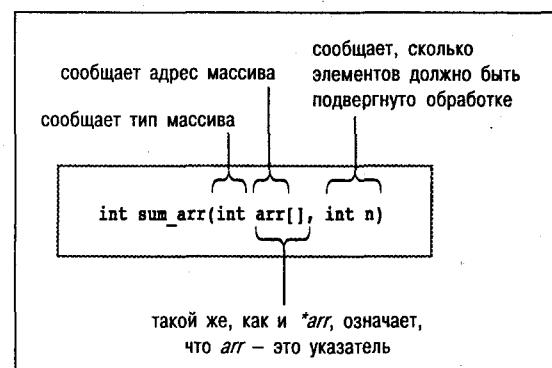


РИСУНОК 7.4 Сведения о массиве, передаваемые функции.

### Листинг 7.6 Программа arrfun2.cpp.

```
// arrfun2.cpp -- функция с массивом в качестве аргумента
#include <iostream>
using namespace std;
const int ArSize = 8;
int sum_arr(int arr[], int n);
int main()
{
 int cookies[ArSize] = {1,2,4,8,16,32,64,128};
// некоторые системы требуют, чтобы int предшествовало ключевое слово static,
// благодаря чему становится возможной инициализация массива

 cout << cookies << " = array address, ";
// некоторые системы требуют, чтобы выполнялось преобразование типов: unsigned (cookies)

 cout << sizeof cookies << " = sizeof cookies\n";
 int sum = sum_arr(cookies, ArSize);
 cout << "Total cookies eaten: " << sum << "\n";
 sum = sum_arr(cookies, 3); // обман
 cout << "First three eaters ate " << sum << " cookies.\n";
 sum = sum_arr(cookies + 4, 4); // еще один обман
 cout << "Last four eaters ate " << sum << " cookies.\n";
 return 0;
}

// возвращает сумму массива целых чисел
int sum_arr(int arr[], int n)
{
 int total = 0;
 cout << arr << " = arr, ";
// некоторые системы требуют, чтобы выполнялось преобразование типов: unsigned (arr)

 cout << sizeof arr << " = sizeof arr\n";
 for (int i = 0; i < n; i++)
 total = total + arr[i];
 return total;
}
```

Ниже представлены результаты выполнения программы (адресные величины, а также размеры массивов и целых чисел в разных системах могут быть различными):

```
0068FDD4 = array address, 32 = sizeof
cookies
0068FDD4 = arr, 4 = sizeof arr
Total cookies eaten: 255
0068FDD4 = arr, 4 = sizeof arr
First three eaters ate 7 cookies.
0068FDE4 = arr, 4 = sizeof arr
Last four eaters ate 240 cookies.
```

### Примечания к программе

Листинг 7.6 иллюстрирует некоторые наиболее интересные моменты использования функций, выполняющих обработку массивов. Прежде всего обратите внимание на то, что `cookies` и `arr` отображают один и тот же адрес с абсолютной точностью. Но `sizeof cookies` равно 16, в то время как `sizeof arr` равно всего лишь 4. Это объясняется тем, что `sizeof cookies` определяет размер всего мас-

сива, в то время как `sizeof arr` — величину переменной типа указатель. (Выполнение этой программы осуществлялось в системе с 4-байтовой адресацией.) Между прочим, именно поэтому нужно передать функции явное значение размера массива, а не использовать для этой цели конструкцию `sizeof arr` в `sum_arr()`.

Единственный способ для функции `sum_arr()`, позволяющий узнать, из какого числа элементов состоит массив, заключается в том, что вы сообщаете ей эти сведения посредством второго аргумента. В связи с этим можно непреднамеренно, так сказать, ввести ее в заблуждение. Например, при втором использовании этой функции программа вызывает ее с помощью следующего обращения:

```
sum = sum_arr(cookies, 3);
```

Сообщив функции, что массив `cookies` содержит не три элемента, вы заставляете ее вычислить сумму первых трех элементов.

Однако зачем ограничиваться только этим? Можно "ввести в заблуждение" функцию и таким образом:

```
sum = sum_arr(cookies + 4, 4);
```

Поскольку `cookies` выступает в качестве адреса первого элемента, `cookies + 4` представляет собой адрес пятого элемента. Этот оператор суммирует пятый, шестой, седьмой и восьмой элементы массива. Просматривая выходные данные, обратите внимание на то, что при третьем обращении эта функция присваивает `arr` адрес, который отличается от адреса при первых двух обращениях. Еще следует добавить, что в качестве аргумента можно использовать `&cookies[4]` вместо `cookies + 4`, поскольку оба эти выражения эквивалентны.

### ПОМНИТЕ

Чтобы указать вид массива и число содержащихся в нем элементов функции, которая выполняет обработку массива, передавайте ей эти сведения в виде двух отдельных аргументов:

```
// прототип
void fillArray(int arr[], int size);
```

Не пытайтесь использовать скобки, чтобы передать сведения о размере массива:

```
// НЕТ – неправильный прототип
void fillArray(int arr[size]);
```

## Другие виды функций, выполняющих обработку массивов

Если вы собираетесь использовать массив для представления некоторых данных, значит вы принимаете проектное решение. Однако проектные решения не определяют метод хранения данных, они должны определять, как эти данные будут использованы. Довольно часто оказывается целесообразным создавать специальные функции, выполняющие специфические операции над данными. (Извлекаемая при этом выгода выражается в более высокой степени надежности программы, в простоте внесения изменений и в удобстве отладки.) Кроме того, когда вы, обдумывая программу, начинаете связывать в единое целое свойства памяти и операции, вы начинаете мыслить категориями ООП, а это обещает принести пользу в будущем.

Рассмотрим простой случай. Предположим, что вы приняли решение использовать массив, чтобы отслеживать стоимость в долларах вашего недвижимого имущества. (В случае отсутствия такового вообразите, что оно у вас есть.) Необходимо принять решение, какой тип данных использовать. Естественно, тип `double` предоставляет большие возможности в смысле диапазона представления величин, чем `int` или `long`, к тому же он обеспечивает достаточно значащих цифр, чтобы представить величину с необходимой точностью. Далее нужно решить, сколько элементов должен содержать массив. (Если речь идет о динамических массивах, созданных

с помощью оператора `new`, можно отложить принятие подобного решения на более поздний срок, но мы сейчас рассматриваем упрощенную картину.) Допустим, у вас имеется не более пяти видов недвижимого имущества, так что можно использовать массив из пяти чисел типа `double`.

Теперь рассмотрим возможные операции, которые вы, по всей вероятности, захотите выполнять над массивом недвижимого имущества. Двумя самыми главными из них являются чтение значений в массив и отображение содержимого массива. Добавим еще одну операцию в этот список: переоценку стоимости недвижимости. Чтобы упростить задачу, предположим, что стоимость всех видов недвижимости увеличивается или уменьшается в одной и той же пропорции. (Не забывайте, что эта книга предназначена для изучения языка C++, а не вопросов управления недвижимостью.) Теперь предусмотрим функцию для каждой операции и создадим соответствующий программный код. Затем мы выполним все описанные выше действия.

### Заполнение массива

Поскольку функция, в качестве аргумента которой используется имя массива, работает с исходным массивом, а не с его копией, вы можете прибегнуть к помощи функции, чтобы присвоить соответствующие значения элементам массива. Одним из аргументов такой функции будет имя массива, который подлежит заполнению. В общем случае программа может отслеживать стоимость недвижимого имущества сразу нескольких лиц, поэтому необходимо будет иметь несколько массивов, так что вы вряд ли захотите жестко задавать в функции размерность массива. Вместо этого передавайте размер массива в качестве второго аргумента, как это было сделано в предыдущем примере. Кроме того, вполне возможно, что у вас появится желание выйти из программы еще до того, как массив будет заполнен, для этого потребуется встроить в функцию соответствующее программное средство. Поскольку у вас появится возможность вводить такие значения числа элементов, которые будут меньше максимального, целесообразно предусмотреть, чтобы функция могла возвращать фактическое число введенных значений. Принимая во внимание приведенные выше соображения, получим следующий прототип функции:

```
int fill_array(double ar[], int limit);
```

Рассматриваемая функция принимает аргумент в виде имени массива и аргумент, определяющий максимальное число элементов, подлежащих считыванию, и возвращает фактическое число считанных элементов. Например, если вы используете эту функцию при работе с массивом, состоящим из пяти элементов, то вы должны передать ей в качестве второго аргумента число 5.

Если вы ввели только три значения, функция возвратит число 3.

Чтобы считать в массив следующие один за другим значения, можно воспользоваться циклом, однако как вы сможете остановить выполнение цикла? Один из способов состоит в использовании специального значения, обозначающего конец ввода. Поскольку ни один из видов собственности не может иметь отрицательную цену, для указания завершения ввода можно использовать какое-либо отрицательное число. С учетом сделанных замечаний можно представить функцию в виде следующего программного кода:

```
int fill_array(double ar[], int limit)
{
 double temp;
 for (int i = 0; i < limit; i++)
 {
 cout << "Enter value #"
 << (i + 1) << ": ";
 cin >> temp;
 if (temp < 0) // сигнал к
 // завершению
 break;
 ar[i] = temp;
 }
 return i;
}
```

Обратите внимание на тот факт, что программный код содержит подсказку для пользователя программы. Если пользователь вводит неотрицательное значение, то это значение присваивается очередному элементу массива. В случае ввода отрицательного значения выполнение цикла прекращается. Если пользователь вводит только допустимые значения, выполнение цикла заканчивается после считывания значения под номером *limit*. Последняя операция, выполняемая в цикле, — это увеличение значения *i* на единицу. Когда выполнение цикла закончится, *i* будет на 1 больше последнего индекса массива, следовательно, оно равно числу введенных элементов. Функция возвращает это значение вызывающей программе.

### **Отображение массива и его защита с помощью спецификатора const**

Создание функции, отображающей содержимое массива на экране, не сопряжено с трудностями. Вы передаете функции имя массива и число введенных элементов, а функция, в свою очередь, прибегает к помощи цикла, чтобы отобразить каждый элемент. Но при этом необходимо соблюсти еще одно условие — отображающая функция не должна вносить изменения в исходный массив. До тех пор пока перед функцией не поставлена задача изменять переданные ей данные, нужно гарантировать, что она это не будет делать. Такая защита обес-

печивается автоматически в случае поступления обычных аргументов, поскольку C++ передает их по значению, а функция работает с их копиями. Но функция, которая использует массивы, работает с оригиналами этих массивов. В конечном итоге именно благодаря этому функция заполнения *fill\_array()* способна выполнять свою задачу. Чтобы предотвратить случайное изменение аргумента типа массив, можно воспользоваться ключевым словом *const* (глава 3) при объявлении формального аргумента:

```
void show_array(const double ar[], int n);
```

Смысль этого объявления состоит в том, что указатель *ar* указывает на неизменяемые данные. Это значит, что вы не можете использовать *ar*, чтобы изменять данные. Другими словами, можно использовать такие значения, как *ar[0]*, но нельзя изменять эти значения. Это свидетельствует не о том, что исходный массив должен оставаться постоянным, а о том, что вы не можете использовать *ar* в функции *show\_array()* для изменения данных. Таким образом, функция *show\_array()* рассматривает массив как данные, предназначенные только для чтения. Предположим, вы случайно нарушили это ограничение, выполнив в функции *show\_array()* действия, подобные следующим:

```
ar[0] += 10;
```

В этом случае компилятор положит конец вашим неправильным действиям. Язык C++ компании Borland, например, выдает в подобной ситуации сообщение об ошибке такого типа (слегка изменено):

```
Cannot modify a const object in function
show_array(const double *, int)
```

Это сообщение напоминает о том, что в интерпретации C++ описание *const double ar[]* означает *const double \*ar*. Таким образом, это описание по существу утверждает, что *ar* указывает на постоянное значение. Мы обсудим этот вопрос несколько подробнее сразу после того, как закончим рассмотрение текущего примера. Между тем, программный код функции *show\_array()* имеет следующий вид:

```
void show_array(const double ar[], int n)
{
 for (int i = 0; i < n; i++)
 {
 cout << "Property #" << (i + 1)
 << ": $";
 cout << ar[i] << "\n";
 }
}
```

### **Модификация элементов массива**

Третья операция, выполняемая над нашим массивом, — это операция умножения каждого элемента массива на

один и тот же коэффициент переоценки. Вам необходимо передать функции три аргумента: коэффициент, массив и число элементов. При этом возвращаемые значения не нужны, так что функция примет вид:

```
void reassess(double r, double ar[], int n)
{
 for (int i = 0; i < n; i++)
 ar[i] *= r;
}
```

Поскольку эта функция предназначена для изменения значений элементов массива, вам не нужно использовать спецификатор `const`, когда вы объявляете `ar`.

### Объединение частей в единое целое

Теперь, когда мы получили описание типа данных, в котором мы собираемся их хранить (массив), и того, как мы собираемся их использовать (три функции), можно объединить эти описания в единую программу, в которой будут использованы эти заготовки. Поскольку все необходимые средства работы с массивами уже созданы, задача разработки программы функции `main()` существенно упростилась. Оставшаяся часть работы, связанной с программированием, касается обращений `main()` к функциям, которые мы только что описали в программных кодах. В листинге 7.7 представлен результат наших усилий.

### Листинг 7.7 Программа arrfun3.cpp.

```
// arrfun3.cpp -- константы и функций для
// работы с массивами
#include <iostream>
using namespace std;
const int Max = 5;

// прототипы функций
int fill_array(double ar[], int limit);
void show_array(const double ar[], int n);
 // изменение данных не допускается
void reassess(double r, double ar[], int n);

int main()
{
 double properties[Max];

 int size = fill_array(properties, Max);
 show_array(properties, size);
 cout << "Enter reassessment rate: ";
 double rate;
 cin >> rate;
 reassess(rate, properties, size);
 show_array(properties, size);
 return 0;
}

int fill_array(double ar[], int limit)
{
 double temp;
 int i;
 for (i = 0; i < limit; i++)
 ar[i] = i * 100000.0;
```

```

 cout << "Enter value #" << (i + 1)
 << ": ";
 cin >> temp;
 if (temp < 0)
 break;
 ar[i] = temp;
}
return i;
}

// следующая функция может использовать, но не
// изменять массив, адресом которого является ar
void show_array(const double ar[], int n)
{
 for (int i = 0; i < n; i++)
 {
 cout << "Property #"
 << (i + 1) << ": $";
 cout << ar[i] << "\n";
 }
}

// умножает каждый элемент ar[] на r
void reassess(double r, double ar[], int n)
{
 for (int i = 0; i < n; i++)
 ar[i] *= r;
}
```

Ниже представлены результаты двух выполнений программы. Напоминаем, что ввод прекращается после того, как пользователь введет сведения о пяти единицах недвижимости или отрицательное целое число, в зависимости от того, какое событие произойдет раньше. В первом примере раньше достигается предел из пяти единиц недвижимого имущества, а во втором примере вводится отрицательное целое число.

```
Enter value #1: 100000
Enter value #2: 80000
Enter value #3: 222000
Enter value #4: 240000
Enter value #5: 118000
Property #1: $100000
Property #2: $80000
Property #3: $222000
Property #4: $240000
Property #5: $118000
Enter reassessment rate: 1.10
Property #1: $110000
Property #2: $88000
Property #3: $244200
Property #4: $264000
Property #5: $129800
Enter value #1: 200000
Enter value #2: 84000
Enter value #3: 160000
Enter value #4: -2
Property #1: $200000
Property #2: $84000
Property #3: $160000
Enter reassessment rate: 1.20
```

```
Property #1: $240000
Property #2: $100800
Property #3: $192000
```

## Примечания к программе

Мы уже рассмотрели наиболее важные вопросы, касающиеся программирования, теперь настала очередь поразмышлять о процессе программирования. Мы начали с того, что задумались над выбором подходящего типа данных и разработали функции, выполняющие соответствующую обработку данных. Далее мы объединили эти функции в единую программу. Это иногда называют программированием "снизу вверх", поскольку процесс разработки направлен от составляющих частей к единому целому. Такой подход характерен для объектно-ориентированного программирования, которое, прежде всего, сосредоточивается на представлении данных и манипулировании данными. С другой стороны, традиционное процедурное программирование склоняется к программированию "сверху вниз", в рамках которого вы сначала разрабатываете большой проект в виде модулей, а затем обращаете внимание на детали. Оба метода полезны, ибо в результате их применения создаются модульные программы.

## Указатели и спецификатор const

Использование спецификатора **const** с указателями связано с некоторыми тонкостями. При работе с указателями вы можете воспользоваться ключевым словом **const** двумя различными способами. В условиях первого способа указатель ссылается на постоянный объект, и это обстоятельство не дает возможности использовать указатель для изменения величины, на которую он указывает. Второй способ состоит в том, чтобы сам указатель сделать константой, и таким образом воспрепятствовать попыткам вносить изменения там, куда указывает указатель. Теперь обратимся к подробностям.

Сначала объявим указатель **pt**, который указывает на константу:

```
int age = 39;
const int * pt = &age;
```

Это объявление утверждает, что **pt** указывает на **const int** (39 в рассматриваемом случае). По этой причине невозможно использовать **pt** с целью изменить это значение. Другими словами, значением **\*pt** является **const**, и его модификация невозможна:

```
*pt += 1; // НЕПРАВИЛЬНО, поскольку pt
 // указывает на const int
cin >> *pt; // НЕПРАВИЛЬНО по той же
 // причине
```

Теперь перейдем к тонкостям. Из данного описания **pt** можно сделать вывод, что величина, на которую он

указывает, не обязательно является константой, это значение является константой лишь постольку, поскольку это касается указателя **pt**. Например, **pt** указывает на **age**, а **age** не является **const**. Вы можете изменить значение **age** непосредственно, используя для этой цели переменную **age**, но вы не можете изменить это значение косвенно, в данном случае через указатель **pt**:

```
*pt = 20; // НЕПРАВИЛЬНО, поскольку pt
 // указывает на const int
age = 20; // ПРАВИЛЬНО, поскольку age
 // не объявлена как const
```

Раньше мы присваивали адрес обычной переменной обычному указателю. Сейчас мы присвоили адрес обычной переменной указателю-на-**const**. При этом остается еще два варианта: присваивание адреса переменной типа **const** указателю-на-**const** и присваивание адреса **const** обычному указателю. Можно ли использовать эти варианты? Первый — да, а второй — нет:

```
const float g_earth = 9.80;
const float * pe = &g_earth; // ПРАВИЛЬНО
const float g_moon = 1.63;
float * pm = &g_moon; // НЕПРАВИЛЬНО
```

В первом случае вы не можете воспользоваться ни константой **g\_earth**, ни константой **pe**, чтобы изменить значение 9.80. Язык C++ не допускает второго случая по одной простой причине — если вы можете присвоить **pm** адрес **g\_moon**, значит, вам нетрудно обмануть систему и использовать **pm** для изменения значения **g\_moon**. Таким образом статус **const** величины **g\_moon** превращается в фикцию, поэтому C++ запрещает присваивать адрес величины со статусом **const** указателю, не имеющему статуса **const**.

### ПОМНИТЕ

Вы можете присвоить указателю-на-**const** адрес значения **const** либо значение, не имеющее статуса **const**. Однако указателю на величину, не имеющей статуса **const**, можно присвоить только адрес данных, не имеющих статуса **const**.

Предположим, что имеется массив данных со статусом **const**:

```
const int months[12] =
{ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Запрет на присваивание адреса массива величин со статусом **const** означает, что вы не можете передать имя массива в качестве аргумента функции, использующей формальный аргумент, не являющийся константой:

```
int sum(int arr[], int n); // должен быть
 // const int arr[]
...
int j = sum(months, 12); // не допускается
```

При вызове функции предпринимается попытка присвоить указателю (`agg`), не имеющему статуса `const`, указатель (`months`) со статусом `const`, а компилятор препятствует выполнению вызова этой функции.

### ИСПОЛЬЗУЙТЕ СПЕЦИФИКАТОР CONST ТАМ, ГДЕ ЭТО ВОЗМОЖНО

Рекомендуется объявлять аргументы указателя как указатели на данные со статусом константы по двум веским причинам:

- Это позволяет избежать программных ошибок, которые неизбежно искажают данные.
- Использование спецификатора `const` позволяет функции выполнять обработку фактических аргументов со статусом и без статуса `const`, в то время как функция, в прототипе которой `const` отсутствует, может принимать только данные, не имеющие статус `const`.

Следует объявлять формальные аргументы указателя как указатели на `const` всякий раз, когда в этом возникает необходимость.

Еще одно узкое место: объявления

```
int age = 39;
const int * pt = &age;
```

лишают вас возможности изменять значение, равное 39, на которое ссылается указатель `pt`. Он отнюдь не препятствует модификации самого указателя `pt`. Иначе говоря, вы не можете присвоить `pt` новый адрес:

```
int sage = 80;
pt = &sage; // возможен указатель на
 // другую ячейку
```

Однако вы все еще не можете использовать `pt`, чтобы изменить значение, на которое он указывает (равное теперь 80).

Второй способ использования спецификатора `const` делает невозможным изменение значения самого указателя:

```
int sloth = 3;
const int * ps = &sloth; // указатель на
 // константу int
int * const finger = &sloth; // неизменный
 // указатель на int
```

Обратите внимание на то, что в последнем описании ключевое слово `const` появилось в другом месте. Такая форма описания приводит к тому, что указатель `finger` указывает только на `sloth`. Тем не менее, он позволяет воспользоваться указателем `finger`, чтобы изменить значение `sloth`. Описание, приведенное в середине (см. выше), не дает возможности использовать указатель `ps` с целью изменить значение `sloth`, однако оно позволяет `ps` указывать на другую ячейку. Короче говоря, и `finger`, и `*ps` имеют статус `const`, а `*finger` и `ps` не имеют статуса `const` (рис. 7.5).

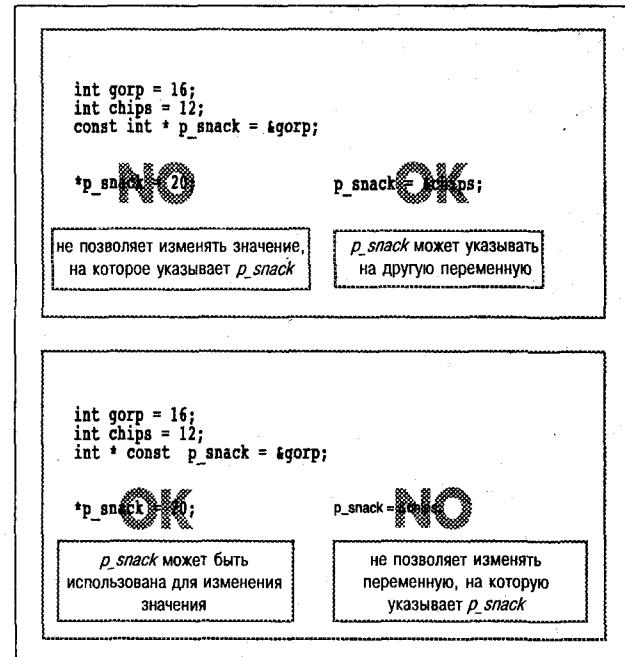


РИСУНОК 7.5 Указатели на величину, имеющую статус `const` и указатели, имеющие статус `const`.

При желании вы можете объявить указатель со статусом `const` на объект со статусом `const`:

```
double trouble = 2.0E30;
const double * const stick = &trouble;
```

Здесь `stick` может указывать только на `trouble`, при этом `stick` не может быть использован для того, чтобы изменить значение `trouble`. Короче говоря, как `stick`, так и `*stick` имеют статус `const`.

Многие из вас часто используют форму указателя-на-`const` для защиты данных в тех случаях, когда вы передаете указатели как аргументы функции. Примером может служить прототип функции `show_array()` из листинга 7.5:

```
void show_array(const double ar[], int n);
```

Использование статуса `const` в этом описании означает, что функция `show_array()` не может изменять значения элементов ни одного из передаваемых ей массивов.

## ФУНКЦИИ И СТРОКИ В СТИЛЕ С

Как вам уже известно, строка в стиле С состоит из последовательности символов, оканчивающейся пустым символом. Многое из того, что вы узнали в процессе проектирования функций, выполняющих обработку массивов, применимо и к функциям, выполняющим обработку строк. Однако при этом имеются несколько осо-

бенностей, обусловленных природой строк, к рассмотрению которых мы сейчас переходим.

Предположим, что вы хотите передать некоторой функции в качестве аргумента строку. Для представления строки в вашем распоряжении имеется три варианта:

- Массив типа `char`
- Строковая константа, заключенная в кавычки (она еще называется строковым литералом).
- Указатель на значение типа `char` (сокращенно — указатель-на-`char`) ссылается на начало строки

Однако все три варианта имеют тип указатель-на-`char` (или короче — тип `char *`), так что вы можете использовать все три в качестве аргументов функций, выполняющих обработку строк:

```
char ghost[15] = "galloping";
char * str = "galumphing";
int n1 = strlen(ghost); // ghost есть
 // &ghost[0]
int n2 = strlen(str); // указатель на
 // символ
int n3 = strlen("gamboling"); // адрес
 // строки
```

В принципе, вы можете говорить, что передаете строку в качестве аргумента, но на самом деле передается адрес первого символа строки. Отсюда следует, что прототип строковой функции должен использовать тип `char*` в качестве типа формального параметра, представляющего строку.

Существенное различие между строкой и обычным массивом заключается в том, что в строке имеется встроенный завершающий символ. (Напомним, что массив типа `char`, который содержит символы, среди которых нет пустых символов, — это всего лишь массив, но не строка.) Это означает, что не следует передавать длину строки как аргумент. Вместо этого функция может воспользоваться циклом, чтобы поочередно проанализировать каждый символ строки, пока цикл не достигнет завершающего пустого символа. Листинг 7.8 показывает, насколько эффективен такой подход, на примере функции, которая подсчитывает, сколько раз тот или иной символ появляется в заданной строке.

### Листинг 7.8 Программа strgfun.cpp.

```
// strgfun.cpp -- функция со строковым
// аргументом
#include <iostream>
using namespace std;
int c_in_str(const char * str, char ch);
int main()
{
 char mmm[15] = "minimum"; // строка в
 // массиве
```

```
// некоторые системы требуют, чтобы типу char
// предшествовала директива static, что
// позволит осуществить инициализацию массива
```

```
char *wail = "ululate"; // wail указывает
 // на строку
```

```
int ms = c_in_str(mmm, 'm');
int us = c_in_str(wail, 'u');
cout << ms << " m characters in " << mmm
 << "\n";
cout << us << " u characters in "
 << wail << "\n";
return 0;
```

```
}
```

---

```
// эта функция подсчитывает число символов ch
// в строке str
int c_in_str(const char * str, char ch)
{
 int count = 0;

 while (*str) // выйти из программы,
 // когда *str будет равен '\0'
 {
 if (*str == ch)
 count++;
 str++; // переместить указатель на
 // следующий символ — char
 }
 return count;
}
```

Результаты выполнения программы:

```
3 m characters in minimum
2 u characters in ululate
```

### Примечания к программе

Поскольку функция `c_in_str()` не должна вносить изменения в исходную строку, она использует спецификатор `const` при описании формального параметра `str`. Если вы ошибочно позволили функции изменить какую-то часть строки, компилятор отловит вашу ошибку. Естественно, можно использовать обозначение массива, вместо того чтобы давать описание `str` в заголовке функции:

```
int c_in_str(const char str[], char ch)
// теперь правильно
```

Однако использование обозначения указателя напоминает вам о том, что аргумент не обязательно должен быть именем массива, он может быть некоторой другой разновидностью указателя.

Функция сама демонстрирует стандартный прием обработки символов в строке:

```
while (*str)
{
 statements
 str++;
}
```

Первоначально `str` указывает на первый символ строки, так что `*str` представляет собой первый символ. Например, сразу после первого обращения к функции `*str` имеет значение `m`, первый символ в строке `minimum`. До тех пор пока таким символом не является пустой символ (`\0`), `*str` имеет ненулевое значение, следовательно, цикл продолжает выполняться. В конце каждого цикла выражение `str++` увеличивает значение указателя на единицу, так что он указывает на следующий символ в строке. В конце концов, `str` укажет на завершающий строку пустой символ, а `*str` получит значение 0, которое является числовым кодом пустого значения. В этом случае цикл прекращается. (Почему функции обработки строк столь "бесцеремонны"? Да потому, что они выполняются в любых ситуациях.)

## Функции, возвращающие строки

Теперь предположим, что вы хотите создать функцию, которая возвращает строку. Правильно, функция не способна сделать это. Но она может возвратить адрес строки, а это даже еще лучше. Например, программа, представленная в листинге 7.9, объявляет функцию с именем `buildstr()`, которая возвращает указатель. Эта функция принимает два аргумента: символ и число. Используя спецификатор `new`, функция создает строку, длина которой равна этому числу, после чего она инициализирует каждый символ. Затем она возвращает указатель на новую строку.

### Листинг 7.9 Программа strgback.cpp.

```
// strgback.cpp -- функция возвращает
// указатель на символ
#include <iostream>
using namespace std;
char * buildstr(char c, int n); // прототип
int main()
{
 int times;
 char ch;

 cout << "Enter a character: ";
 cin >> ch;
 cout << "Enter an integer: ";
 cin >> times;
 char *ps = buildstr(ch, times);
 cout << ps << "\n";
 delete [] ps; // свободная память
 ps = buildstr('+', 20); // повторное
 // использование указателя
 cout << ps << "-DONE-" << ps << "\n";
 delete [] ps; // свободная память
 return 0;
}

// создает строку, состоящую из n символов с
char * buildstr(char c, int n)
{
```

```
 char * pstr = new char[n + 1];
 pstr[n] = '\0'; // завершить строку
 while (n-- > 0)
 pstr[n] = c; // заполнить остальную
 // часть строки
 return pstr;
}
```

Результаты выполнения программы:

```
Enter a character: V
Enter an integer: 46
Vvvv
+++++-----DONE-----+++++
```

### Примечания к программе

Чтобы создать строку из `n` видимых символов, необходима область памяти для хранения `n + 1` символов, поскольку нужно место для пустого символа. Таким образом, функция требует `n + 1` байтов для хранения такой строки. После этого она отводит последний байт для пустого символа и заполняет остальную часть массива в обратном направлении. Цикл

```
while (n-- > 0)
 pstr[n] = c;
```

повторяется `n` раз, пока значение `n` не уменьшится до нуля, при этом заполняются все `n` элементов массива. В начале заключительного цикла `n` имеет значение 1. Поскольку `n--` означает использовать значение и уменьшить его на единицу, проверяемое условие цикла `while` позволяет сравнить 1 с 0 и убедиться, что результатом проверки является `true`, поэтому цикл продолжает выполняться. Но по завершении проверки функция уменьшает значение `n` на 1, так что `n` становится равным 0, поэтому `pstr[0]` — это последний элемент, которому присваивается значение `c`. Причиной того, что заполнение строки ведется в обратном порядке, объясняется стремление избежать использования дополнительной переменной. Использование какого-нибудь другого порядка приводит к необходимости выполнять такого рода операции:

```
int i = 0;
while (i < n)
 pstr[i++] = c;
```

Обратите внимание на то обстоятельство, что переменная `pstr` по отношению к функции `buildstr` является локальной, так что, когда выполнение этой функции заканчивается, память, отведенная под указатель `pstr` (но не под саму строку), освобождается. Но поскольку функция возвращает значение `pstr`, программа сохраняет возможность доступа к новой строке через указатель `ps` в теле функции `main()`.

Программа использует оператор `delete`, чтобы освободить память, которая была отведена для строки, после того, как строка станет ненужной. Затем она повторно использует указатель `ps`, чтобы указывать на новый блок памяти, выделенный для следующей строки, и освобождает эту память. Недостаток такого подхода (когда функция возвращает указатель на память, выделенную спецификатором `new`) состоит в том, что программист обязан помнить, что надо воспользоваться оператором `delete`. Шаблон `auto_ptr`, который будет рассмотрен в главе 15, позволит облегчить автоматизацию этого процесса.

## Функции и структуры

Теперь от массивов перейдем к структурам. Создавать функции для работы со структурами легче, чем функции для работы с массивами. Несмотря на то что переменные типа структуры напоминают массивы в том, что содержат некоторое множество элементов данных, структуры ведут себя по отношению к функциям как базовые переменные, принимающие только одно значение. Вы можете передать структуру по значению точно так же, как это делается с обычной переменной. В этом случае функция работает с копией исходной структуры. С другой стороны, функция может возвращать структуры. Если говорить о структуре, то в этом случае не бывает таких неудобств, когда имя массива, по сути, является адресом его первого элемента. Имя структуры — это просто имя структуры и ничего больше, и если вам нужен ее адрес, нужно воспользоваться операцией адресации `&`.

Наиболее простой путь для создания программы, использующей структуры, заключается в том, чтобы рассматривать эти структуры как базовые типы данных, т.е. передавать их как аргументы и при необходимости использовать как возвращаемые значения. Однако для передачи структур по значению характерен один недостаток. Если сама структура большая, то возникнет необходимость в большем объеме памяти, кроме того, требуется приложить больше усилий для создания копии структуры и при этом замедляется работа системы. По этим причинам (и прежде всего, в силу того, что язык C не позволяет передавать структуры по значению) многие программисты, работающие в среде C, предпочитают передавать адрес структуры и только после этого использовать указатель для доступа к содержимому структуры. C++ предоставляет третью альтернативу, именуемую передачей по ссылке, которая будет рассмотрена в главе 8. Сейчас мы рассмотрим два первых варианта, начиная с передачи и возврата всей структуры.

## Передача и возврат структур

Передача структур по значению имеет смысл, когда сама структура относительно компактна. Рассмотрим два соответствующих примера. В первом примере речь идет о времени путешествия (не путать с путешествием во времени). Согласно некоторым картам, чтобы добраться от водопада Тандер-Фолс (Thunder Falls) до города Бинго (Bingo), потребуется 3 часа 50 минут, а чтобы добраться от города Бинго до Гротеско (Grotesquo) — 1 час 25 минут. Чтобы представить эти временные величины, можно воспользоваться структурой, в которой один элемент представляет значение часов, а второй — значения минут. Сложение требует определенного навыка, так как можно напутать и переслать значения минут в ту часть структуры, в которой фиксируются часы. Например, сумма двух указанных ранее временных величин составляет 4 часа 75 минут, или 5 часов 15 минут. Разработаем структуру, представляющую временные величины, а затем и функцию, которая принимает две такие структуры в качестве аргументов и возвращает структуру, которая представляет их сумму.

Определение структуры не представляет собой трудностей:

```
struct travel_time
{
 int hours;
 int mins;
};
```

Далее рассмотрим прототип функции `sum()`, которая возвращает сумму двух таких структур. Возвращаемое значение должно иметь тип `travel_time`, этот же тип должны иметь оба ее аргумента. Таким образом, прототип должен иметь такой вид:

```
travel_time sum(travel_time t1, travel_time t2);
```

Чтобы сложить две временные величины, сложим сначала элементы, содержащие минуты. Целочисленное деление на 60 дает число часов, которое нужно перенести в раздел часов, а оператор деления по модулю (%) дает число минут в остатке. Программа, представленная в листинге 7.10, реализует этот подход через функцию `sum()`, к ней добавлена функция `show_time()`, в задачу которой входит отображение содержимого структуры `travel_time`.

### Листинг 7.10 Программа travel.cpp.

```
// travel.cpp -- использование структур и
// функций
#include <iostream>
using namespace std;
struct travel_time
{
 int hours;
```

```

int mins;
};

const int Mins_per_hr = 60;

travel_time sum(travel_time t1, travel_time t2);
void show_time(travel_time t);

int main()
{
 travel_time day1 = {5, 45}; // 5 часов
 // 45 минут
 travel_time day2 = {4, 55}; // 4 часа
 // 55 минут

 travel_time trip = sum(day1, day2);
 cout << "Two-day total: ";
 show_time(trip);

 travel_time day3 = {4, 32};
 cout << "Three-day total: ";
 show_time(sum(trip, day3));

 return 0;
}

travel_time sum(travel_time t1, travel_time t2)
{
 travel_time total;

 total.mins = (t1.mins + t2.mins) %
 Mins_per_hr;
 total.hours = t1.hours + t2.hours +
 (t1.mins + t2.mins) / Mins_per_hr;
 return total;
}

void show_time(travel_time t)
{
 cout << t.hours << " hours, "
 << t.mins << " minutes\n";
}

```

Здесь `travel_time` ведет себя так же, как и имя стандартного типа; вы можете воспользоваться им для объявления переменных, возвращаемых типов функции и типов аргументов функции. Поскольку такие переменные, как `total` и `t1`, — это структуры типа `travel_time`, вы можете применить к ним оператор принадлежности "точка". Обратите внимание на то обстоятельство, что функция `sum()` возвращает структуру `travel_time`. Вы можете использовать эту структуру в качестве аргумента функции `show_time()`. Поскольку по умолчанию функции в языке C++ передают аргументы по значению, функция `show_time(sum(trip, day3))` вычисляет значение функции `sum(trip, day3)`, чтобы найти возвращаемое ею значение. Затем выполняется вызов функции `show_time()` для передачи значения, возвращенного функцией `sum()`, самой функции `show_time()`. Ниже приведены результаты выполнения этой программы:

Two-day total: 10 hours, 40 minutes  
 Three-day total: 15 hours, 12 minutes

## Еще один пример

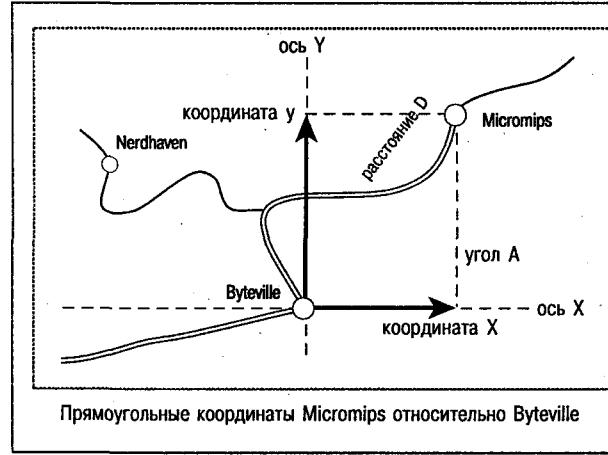
Многое из того, что мы узнаем о функциях и структурах, приведет нас к понятию классов, поэтому полезно рассмотреть второй пример. На этот раз мы будем иметь дело с пространством, а не со временем. В частности, в примере даются определения двух структур, представляющих собой различные способы задания местоположения точки, а затем разрабатываются функции, переводящие одну форму в другую и отображающие результат. Этот пример требует больших математических познаний, чем предыдущий, но вам не обязательно заниматься математикой, чтобы успешно изучать язык C++.

Предположим, что требуется описать положение точки на экране или конкретного объекта на карте относительно некоторой начальной точки. Один из способов состоит в задании смещений объекта по горизонтали и вертикали относительно этой начальной точки. Традиционно математики используют символ  $x$  для представления горизонтального смещения и  $y$  — для представления вертикального смещения (рис. 7.6). В совокупности  $x$  и  $y$  представляют декартовы (или прямоугольные) координаты. Можно определить структуру, состоящую из двух координат, определяющих позицию точки:

```

struct rect
{
 double x; // расстояние от начальной
 // точки по горизонтали
 double y; // расстояние от начальной
 // точки по вертикали
};

```



Прямоугольные координаты Micromips относительно Byteville

**РИСУНОК 7.6** Декартовы (или прямоугольные) координаты точки.

Второй способ описания местоположения точки заключается в том, что определяется ее расстояние от начальной точки и указывается, в каком направлении она находится (например, 40 градусов на северо-восток). По традиции математики измеряют угол в направлении, противоположном вращению часовой стрелки от горизонтальной оси (рис. 7.7). Это расстояние и угол в совокупности составляют полярные координаты точки. Можно объявить вторую структуру, представляющую собой такой метод определения местоположения точек:

```
struct polar
{
 double distance; // расстояние
 // от начальной точки
 double angle; // направление
 // относительно начальной точки
};
```

Построим функцию, которая отображает содержимое структуры типа **polar**. Математические функции в библиотеке C++ предполагают, что углы измеряются в радианах, так что и мы будем измерять углы в этих же единицах. Однако для целей отображения мы преобразуем радиан как единицу измерения углов в градусы. Это означает умножение на величину  $180/\pi$ , которая приблизительно равна 57.29577951. Вот эта функция:

```
// показать полярные координаты, представив
// значение угла в градусах
void show_polar (polar dapos)
{
 const double Rad_to_deg = 57.29577951;

 cout << "distance = " << dapos.distance;
 cout << ", angle = " << dapos.angle *
 Rad_to_deg;
 cout << " degrees\n";
}
```

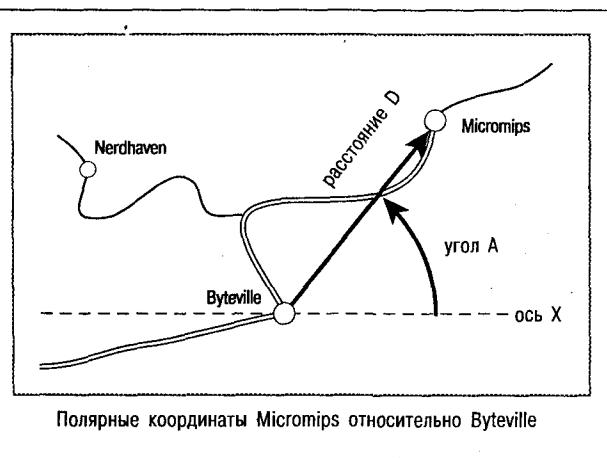


РИСУНОК 7.7 Полярные координаты точки.

Обратите внимание на тот факт, что типом формальной переменной является **polar**. Когда вы передаете структуру типа **polar** этой функции, содержимое структуры **polar** копируется в структуру **dapos**, а функция дальней использует эту копию в своей работе. Поскольку **dapos** является структурой, функция использует оператор принадлежности (точка) (см. главу 4) для идентификации элементов структуры.

Далее попытаемся сделать что-нибудь более полезное и создадим функцию, которая преобразует прямоугольные координаты в полярные. Напишем функцию таким образом, чтобы она передавала структуру **rect** другой функции и принимала от нее структуру **polar** для последующей ее передачи вызывающей программе. Для этого требуется воспользоваться функциями из математической библиотеки, поэтому в программу должен быть включен заголовочный файл **math.h**. Следовательно, в некоторых системах необходимо сообщить компилятору, чтобы он загрузил библиотеку математических функций (см. главу 1). Чтобы определить расстояния по горизонтальным и вертикальным координатам, можно воспользоваться теоремой Пифагора:

```
distance = sqrt(x * x + y * y)
```

Функция **atan2()** из библиотеки математических функций вычисляет значение угла по значениям **x** и **y**:

```
angle = atan2(y, x)
```

(В этой библиотеке имеется также функция **atan()**, но она может определять значение угла в пределах 180 градусов. Такая неопределенность так же нежелательна для математической функции, как и отсутствие проводника в пустыне.)

Имея в распоряжении эти формулы, можно представить искомую функцию в следующем виде:

```
// преобразование прямоугольных координат в
// полярные
polar rect_to_polar(rect xypos) // тип polar
{
 polar answer;

 answer.distance = sqrt(xypos.x *
 xypos.x + xypos.y * xypos.y);
 answer.angle = atan2(xypos.y, xypos.x);
 return answer; // возвращает структуру
 // polar
}
```

Теперь, когда эти функции уже готовы, процесс создания программы не вызовет трудностей. В листинге 7.11 представлен текст готовой программы.

#### Листинг 7.11 Программа strctfun.cpp.

```
// strctfun.cpp -- функции, аргументами
// которых являются структуры
#include <iostream>
```

```
#include <cmath>
using namespace std;

// шаблоны структур
struct polar
{
 double distance; // расстояние от точки
 // отсчета
 double angle; // расстояние от точки
 // отсчета
};

struct rect
{
 double x; // расстояние от точки
 // отсчета по горизонтали
 double y; // расстояние от точки
 // отсчета по вертикали
};

// прототипы
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);

int main()
{
 rect rplace;
 polar pplace;

 cout << "Enter the x and y values: ";
 while (cin >> rplace.x >> rplace.y)
 // удачное использование cin
 {
 pplace = rect_to_polar(rplace);
 show_polar(pplace);
 cout << "Next two numbers (q to
 quit): ";
 }
 return 0;
}

// преобразование прямоугольных координат в
// полярные
polar rect_to_polar(rect xypos)
{
 polar answer;

 answer.distance = sqrt(xypos.x *
 xypos.x + xypos.y * xypos.y);
 answer.angle = atan2(xypos.y, xypos.x);
 return answer; // возвращает структуру
 // polar
}

// отображение полярных координат с
// представлением угла в градусах
void show_polar (polar dapos)
{
 const double Rad_to_deg = 57.29577951;

 cout << "distance = " << dapos.distance;
 cout << ", angle = " << dapos.angle *
 Rad_to_deg;
 cout << " degrees\n";
}

```



### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторых версиях все еще используется заголовочный файл `math.h`, а не более новый вариант заголовочного файла `cmath`. Некоторые компиляторы требуют задания явных инструкций для поиска библиотеки математических функций. Например, в старых версиях `g++` используется такая командная строка:

```
g++ structfun.C -lm
```

Ниже приведены результаты выполнения программы:

```
Enter the x and y values: 30 40
distance = 50, angle = 53.1301 degrees
Next two numbers (q to quit): -100 100
distance = 141.421, angle = 135 degrees
Next two numbers (q to quit): q
```

### Примечания к программе

Мы уже анализировали работу двух функций, использованных в этой программе, теперь рассмотрим, как программа использует объект `cin` для управления циклом `while`:

```
while (cin >> rplace.x >> rplace.y)
```

Напомним, что `cin` — это объект класса `istream`. Оператор извлечения (`>>`) спроектирован таким образом, что результатом выполнения конструкции `cin >> rplace.x` будет также объект этого типа. Как вы сможете убедиться в главе 10, операторы класса реализуются вместе с функциями. Когда вы фактически используете конструкцию `cin >> rplace.x`, то реакция на нее программы состоит в том, что она вызывает функцию, которая возвращает значение типа `istream`. Если вы примените к конструкции `cin >> rplace.x` оператор извлечения (подобно тому, как это делается в выражении `cin >> rplace.x >> rplace.y`), то снова получите объект класса `istream`. Следовательно, оценка проверяемого выражение цикла `while` в конечном итоге сводится к оценке `cin`, в результате выполнения которой, как известно, будучи использованным в контексте проверяемого выражения, получает значение `true` или `false` типа `bool`. Например, в данном цикле `cin` ожидает, что пользователь введет два целых числа. Если вместо них ввести `q`, что мы в конце концов и сделали, `cin >>` убеждается в том, что `q` не является числом. Этот оператор помещает `q` во входную очередь и возвращает значение, преобразованное в `false`, которое завершает выполнение цикла.

Сравните этот подход считывания чисел с подходом, использованным в программе, представленной в листинге 7.7:

```
for (int i = 0; i < limit; i++)
{
 cout << "Enter value #" << (i + 1)
 << ": ";
 cin >> temp;
 if (temp < 0)
```

```
 break;
 ar[i] = temp;
}
```

Чтобы завершить выполнение цикла на ранней стадии, необходимо ввести отрицательное число. Следовательно, ввод ограничивается неотрицательными значениями. Такое ограничение не выходит за рамки требований этой программы, но в большинстве случаев потребуется средство прерывания выполнения программы, которое не исключает использования каких-либо конкретных числовых значений. Использование конструкции `cin >>` в качестве проверяемого условия позволяет избежать подобного рода ограничений, так как она правильно воспринимает ввод любого цифрового значения. Имейте это в виду в тех случаях, когда вам придется воспользоваться циклом для ввода чисел. Кроме того, следует иметь в виду, что ввод нечислового символа вызывает состояние ошибки, которое делает невозможным дальнейшее продолжение ввода. Если вашей программе понадобится ввести данные уже после того, как будет выполнен входной цикл, вам следует воспользоваться функцией `cin.clear()`, чтобы перенастроить выполнение ввода так, как описано в главах 6 и 16.

## Передача адресов структур

Предположим, что вы хотите сэкономить время и память, передавая адрес структуры, вместо того чтобы передавать саму структуру. Это требует переделок функций таким образом, чтобы они могли работать с указателями на структуры. Прежде всего подумаем, какие внести изменения в функцию `show_polar()`. Требуется внести следующие три изменения:

- При вызове функции передавать адрес структуры (`&place`), но не саму структуру (`place`).
  - Объявить формальный параметр таким образом, чтобы он был указателем-на-`polar`, т.е. имел тип `polar*`. Поскольку функция не должна подвергать структуру изменениям, нужно пользоваться спецификатором `const`.
  - Поскольку формальный параметр представляет собой указатель, а не структуру, воспользуйтесь оператором непрямой принадлежности (`->`) вместо операции принадлежности (точка).

После внесения указанных выше изменений функция принимает вид:

```
// отобразить полярные координаты,
// представить с этой целью угол в градусах
void show_polar (const polar * pda)
{
 const double Rad_to_deg = 57.29577951;
 cout << "distance = " << pda->distance;
```

```
cout << ", angle = "
 << pda->angle * Rad_to_deg;
cout << " degrees\n";
```

Далее внесем изменения в `rect_to_polar`. Это более сложная задача, так как исходная функция `rect_to_polar` возвращает структуру. Чтобы в полной мере воспользоваться всеми преимуществами, предоставляемыми указателями, следует отказаться от возвращаемых значений в пользу указателей. Чтобы решить эту задачу, функции нужно передать два указателя. Первый из них указывает на преобразуемую структуру, второй — на структуру, в которой сохраняется результат преобразования. Вместо того чтобы *возвращать* новую структуру, эта функция *модифицирует* существующую структуру в вызывающей функции. Отсюда следует, что если первый аргумент является указателем со статусом `const`, то второй таким не является. В противном случае примените те же принципы, которые были использованы для преобразования функции `show_polar()` в целях использования аргументов типа указатель. В листинге 7.12 представлена переработанная программа.

### Листинг 7.12 Программа strctptr.cpp.

```

// strctptr.cpp -- функции с аргументами типа
// указатели на структуры
#include <iostream>
#include <cmath>
using namespace std;

// шаблоны структур
struct polar
{
 double distance; // расстояние от точки
 // отсчета
 double angle; // направление от точки
 // отсчета
};

struct rect
{
 double x; // расстояние от точки
 // расчета по горизонтали
 double y; // расстояние от точки
 // отсчета по вертикали
};

// прототипы
void rect_to_polar(const rect * pxy, polar * pda);
void show_polar (const polar * pda);

int main()
{
 rect rplace;
 polar pplace;

 cout << "Enter the x and y values: ";
 while (cin >> rplace.x >> rplace.y)
 {
 rect_to_polar(&rplace, &pplace);
 // передать адреса
 }
}

```

```

show_polar(&place); // передать
 // адрес
cout << "Next two numbers (q to quit): ";
}

return 0;
}

// преобразование прямоугольных координат в
// полярные
void rect_to_polar(const rect * rxy, polar * pda)
{
 pda->distance =
 sqrt(rxy->x * rxy->x + rxy->y * rxy->y);
 pda->angle = atan2(rxy->y, rxy->x);
}

```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В некоторых реализациях системы все еще используется заголовочный файл `math.h` вместо более нового – `cmath`. Некоторые компиляторы требуют прямых инструкций для поиска в библиотеке математических функций.

С точки зрения пользователя поведение программы, представленной в листинге 7.12, не отличается от поведения программы, представленной в листинге 7.11. Скрытое различие заключается в том, что первая (листинг 7.11) работает с копиями структур, в то время как вторая (листинг 7.12) использует указатели на исходные структуры.

## Рекурсия

А теперь обратимся к совершенно другой теме. Функция в C++ обладает весьма интересным свойством — она может вызывать саму себя. (В отличие от языка C, однако, C++ не допускает обращения функции `main()` к самой себе.) Это свойство называется *рекурсией*. Рекурсия — это исключительно важное инструментальное средство в некоторых видах программ, таких как программа искусственного интеллекта, однако мы сейчас лишь в общих чертах ознакомимся с тем, как она работает.

Когда рекурсивная функция вызывает саму себя, только что вызванная функция также вызывает саму себя и т.д. до бесконечности, если, конечно, ее программный код не содержит ничего такого, что способно прервать эту последовательность вызовов. Обычно рекурсивный вызов стараются сделать частью условного оператора. Например, рекурсивная функция типа `void` с именем `recurs()` может иметь такой вид:

```

void recurs(список_аргументов)
{
 операторы1
 if (проверка)
 recurs(аргументы)
 операторы2
}

```

По счастливому стечению обстоятельств или как результат запланированных действий `test` в конечном итоге принимает значение `false` и последовательность вызовов прерывается.

Рекурсивные вызовы порождают следующую цепь событий. Пока условный оператор будет истинным, в результате каждого вызова функции `recurs()` выполняется блок *операторы1*, а затем выполняется очередной вызов функции `recurs()`, при этом блок *операторы2* остается вне досягаемости. Когда условный оператор примет значение `false`, начнется выполнение блок *операторов2*. Далее, когда выполнение текущего блока будет завершено, управление программой передается к предыдущей версии функции `recurs()`, которая его вызвала. Затем эта версия функции `recurs()` заканчивает выполнение своего блока *операторы2* и завершается сама, возвращая управление предыдущему вызову функции, и т.д. Таким образом, если функция `recurs()` подвергнется пяти рекурсивным вызовам, первый блок операторов *операторы1* будет выполнен пять раз в том порядке, в каком эти функции были вызваны, затем блок *операторы2* будет выполнен пять раз в порядке, обратном последовательности вызова функций. Погрузившись на пять уровней рекурсии, программа должна затем подняться на те же пять уровней. Программа, представленная в листинге 7.13, служит иллюстрацией подобного поведения.

### Листинг 7.13 Программа `recur.cpp`.

```

// recur.cpp -- выполняется рекурсия
#include <iostream>
using namespace std;
void countdown(int n);

int main()
{
 countdown(4); // вызов рекурсивной
 // функции
 return 0;
}

void countdown(int n)
{
 cout << "Counting down ... "
 << n << "\n";
 if (n > 0)
 countdown(n-1); // функция вызывает
 // саму себя
 cout << n << ": Kaboom!\n";
}

```

Вот результат выполнения программы:

```

Counting down ... 4 " level 1--beginning
 to add levels of recursion
Counting down ... 3 " level 2
Counting down ... 2 " level 3

```

```

Counting down ... 1 " level 4
Counting down ... 0 " level 5
0: Kaboom! " level 5--beginning to
 back out through the series of calls
1: Kaboom! " level 4
2: Kaboom! " level 3
3: Kaboom! " level 2
4: Kaboom! " level 1

```

Обратите внимание на тот факт, что в результате каждого рекурсивного вызова создается собственный набор переменных, так что к тому времени, как программа достигнет пятого вызова, у нее уже будет пять отдельных переменных с именем `i`, при этом каждая имеет собственное, отличное от других значение.

Рекурсия особенно полезна в ситуациях, когда требуется многократное разбиение задачи на две аналогичные подзадачи меньшего масштаба. В качестве примера рассмотрим, как работает этот подход при изготовлении чертежа линейки. Отметим оба конца линейки, найдем среднюю точку и пометим ее. Затем применим эту процедуру сначала к левой половине линейки, а затем к правой. Если вы намерены продолжить разметку, выполните процедуру этого типа по отношению к очередному участку. Такой рекурсивный подход иногда называют *стратегией "разделяй и властвуй"*. Программа, представленная в листинге 7.14, иллюстрирует этот подход на примере функции `subdivide()`. Она использует строку, которая в исходном положении заполнена пробелами, если не считать символа |, обозначающего оба конца строки. Чтобы обратиться к функции `subdivide()` шесть раз, главная программа прибегает к помощи цикла, каждый раз увеличивая глубину рекурсии и выводя получающуюся при этом строку. Таким образом, каждая выходная строка представляет собой дополнительный уровень рекурсии.

#### Листинг 7.14 Программа ruler.cpp.

```

// ruler.cpp - использование рекурсии для
// нанесения делений линейки
#include <iostream>
using namespace std;
const int Len = 66;
const int Dvls = 6;
void subdivide(char ar[], int low, int high,
 int level);
int main()
{
 char ruler[Len];
 int i;
 for (i = 1; i < Len - 2; i++)
 ruler[i] = ' ';
 ruler[Len - 1] = '\0';
 int max = Len - 2;
 int min = 0;
 ruler[min] = ruler[max] = '|';
 cout << ruler << "\n";
}

```

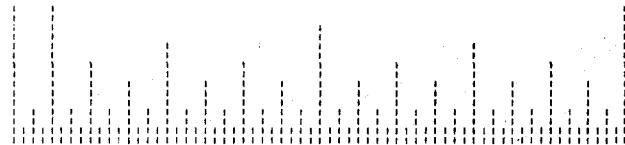
```

for (i = 1; i <= Dvls; i++)
{
 subdivide(ruler, min, max, i);
 cout << ruler << "\n";
 for (int j = 1; j < Len - 2; j++)
 ruler[j] = ' ';
 // возврат к
 // исходному разбиению линейки
}

return 0;
}
void subdivide(char ar[], int low, int high,
 int level)
{
 if (level == 0)
 return;
 int mid = (high + low) / 2;
 ar[mid] = '|';
 subdivide(ar, low, mid, level - 1);
 subdivide(ar, mid, high, level - 1);
}

```

Результаты выполнения программы:



#### Примечания к программе

Для управления уровнями ресурсов функция `subdivide()` использует переменную с именем `level`. Всякий раз, когда функция обращается к самой себе, она уменьшает значение переменной `level` на 1, и функция, значение переменной `level` которой достигает нуля, перестает выполняться. Обратите внимание на то обстоятельство, что функция `subdivide()` вызывает сама себя дважды: один раз для деления левой части и один раз для деления правой части. Исходная средняя точка становится правым концом для первого вызова и левым концом для второго вызова. Следует отметить, что число обращений возрастает в геометрической прогрессии. То есть, один вызов генерирует два последующих, которые, в свою очередь, генерируют уже четыре вызова, потом восемь и т.д. Вот почему вызов уровня 6 способен заполнить 64 элемента ( $2^6 = 64$ ).

#### Указатели на функции

Ни какое описание функций С или С++ не будет полным, если не будут рассмотрены указатели на функции. Мы коротко расскажем о них.

Функции, как и элементы данных, имеют адреса. Адресом функции является адрес, с которого начинается машинный код функции, хранящийся в памяти. Обычно пользователи не извлекают никаких преиму-

ществ из того, что знают этот адрес, но эти сведения могут оказаться полезными для программы. Например, можно создать функцию, которая принимает адрес другой функции в качестве аргумента. Это дает возможность первой функции найти вторую функцию и выполнить ее. Такой подход более громоздкий, чем прямое обращение ко второй функции со стороны первой, тем не менее, он позволяет осуществлять передачу адресов различных функций в различные моменты. Это означает, что первая функция может использовать различные функции в различные моменты.

## Назначение указателя на функцию

Внесем в этот процесс ясность, проиллюстрировав его примером. Предположим, вы намерены спроектировать функцию `estimate()`, которая вычисляет время, необходимое для написания заданного числа строк программного кода, а вы хотите, чтобы разные программисты пользовались этой функцией. Часть программного кода функции `estimate()` будет одинаковой для всех пользователей, однако данная функция позволит каждому программисту воспользоваться собственным алгоритмом вычисления времени. Механизм реализации этого подхода состоит в передаче функции `estimate()` адреса функции, реализующей конкретный алгоритм, который пользователь желает использовать. Чтобы осуществить этот план, нужно выполнить следующее:

- Принять адрес функции.
- Объявить указатель на функцию.
- Использовать указатель на функцию для вызова этой функции.

## Получение адреса функции

Получение адреса функции несложно: достаточно воспользоваться именем функции без замыкающих скобок, т.е. если `think()` — функция, то `think` — адрес этой функции. Чтобы передать какую-либо функцию как аргумент, достаточно передать имя этой функции. Проведите четкое различие между передачей адреса функции и передачей возвращаемого значения функции.

```
process(think); // передает функции
 // process() адрес функции think()
thought(think()); // передает функции
 // thought() возвращаемое значение
 // функции think()
```

Вызов `process()` позволяет функции `process()` вызывать функцию `think()` из функции `process()`. Обращение к функции `thought()` сначала приводит к вызову функции `think()`, а затем к передаче значения, возвращаемого функцией `think()`, функции `thought()`.

## Объявление указателя на функцию

Если вы объявили указатели на типы данных, то в этом объявлении должно быть четко отмечено, на какой тип указывает тот или иной указатель. Аналогично указатель на функцию должен указывать, на какой тип функции ссылается указатель. Это означает, что такое объявление должно идентифицировать тип данных, возвращаемых функцией, равно как и сигнатуру функции (список ее аргументов). Другими словами, такое объявление должно сообщать нам те же сведения о функции, которые уже содержатся в прототипе этой функции. Например, предположим, что Памела Лекодер (Pam LeCoder) написала функцию подсчета затраченного времени со следующим прототипом:

```
double pam(int); // прототип
```

Вот как выглядит объявление соответствующего типа указателя:

```
double (*pf)(int); // pf указывает на
 // функцию, которая принимает один аргумент
 // типа int и которая возвращает тип double
```

Обратите внимание на тот факт, что это объявление во многом похоже на объявление `pam()`, при этом `(*pf)` выступает в роли одной из частей `pam`. Поскольку `pam` является функцией, таковой является и `(*pf)`. А если `(*pf)` является функцией, то `pf` является указателем на функцию.



### СОВЕТ

В общем случае, чтобы объявить указатель на функцию конкретного типа, сначала можно создать прототип обычной функции нужного типа, а затем заменить имя этой функции на выражение вида `{*pf}`. Благодаря этому `pf` становится указателем на функцию этого типа.

В объявлении необходимо заключить `*pf` в круглые скобки, чтобы соблюсти соответствующий приоритет операторов. Круглые скобки имеют более высокий приоритет, чем оператор `*`, следовательно, `*pf(int)` означает, что `pf()` — это функция, которая возвращает указатель, в то время как `(*pf)(int)` означает, что `pf` является указателем на функцию:

```
double (*pf)(int); // pf указывает на
 // функцию, которая возвращает значение
 // типа double
double *pf(int); // pf() является
 // функцией, которая возвращает указатель
 // на значение типа double
```

После того как вы правильно объявили указатель `pf`, вы можете присвоить ему адрес подходящей функции:

```
double pam(int);
double (*pf)(int);
pf = pam; // теперь pf указывает на
 // функцию pam()
```

Обратите внимание на то, что функция `pam()` должна соответствовать `pf` как по сигнатуре, так и по типу. Компилятор отвергает противоречивые операции присваивания, содержащие подобного рода несоответствия:

```
double ned(double);
int ted(int);
double (*pf)(int);
pf = ned; // неправильно - несоответствие
 // сигнатур
pf = ted; // неправильно - несоответствие
 // возвращаемых типов
```

Возвратимся к функции `estimate()`, о которой шла речь выше. Предположим, что вы хотите передать ей количество строк программного кода, которые нужно написать, а также адрес алгоритма оценки (например, адрес функции `pam()`). В этом случае может иметь место следующий прототип:

```
void estimate(int lines, double (*pf)(int));
```

Из этого объявления следует, что второй аргумент является указателем на функцию, аргументами которой являются значение типа `int` и возвращаемое значение типа `double`. Чтобы заставить функцию `estimate()` использовать функцию `pam()`, передайте ей адрес функции `pam()`:

```
estimate(50, pam); // обращение к функции,
 // вынуждающее функцию
 // estimate() использовать pam()
```

Совершенно ясно, что основная сложность в использовании указателей на функции приходится на написание прототипов, в то же время передача адреса — совсем простая процедура.

### Использование указателя для вызова функции

Сейчас мы перейдем к разработке завершающей части метода, который использует указатель для вызова функции, на которую он указывает. Ключевое значение имеет объявление указателя. Напомним, что `(*pf)` в этом случае играл ту же роль, что и имя функции. Таким образом, все что нам нужно сделать, — это воспользоваться `(*pf)`, как если бы это было имя функции:

```
double pam(int);
double (*pf)(int);
pf = pam; // pf сейчас указывает на
 // функцию pam()
double x = pam(4); // обратиться к pam(),
 // используя для этой цели имя функции
double y = (*pf)(5); // обратиться к pam(),
 // используя для этой цели указатель pf
```

По существу, C++ также позволяет вам использовать указатель `pf`, как если бы он был именем функции:

```
double y = pf(5); // также обращается к
 // функции pam(), используя для
 // этой цели указатель pf.
```

Мы будем пользоваться первой формой. Она более громоздкая и неудобная, но в то же время она служит наглядным напоминанием о том, что в программном коде используется указатель на функцию.

### ИСТОРИЯ И ЛОГИКА

Удивительны возможности синтаксиса! Как могут `pf` и `(*pf)` оказаться эквивалентными? В историческом плане одна философская школа утверждает, что, поскольку `pf` — это указатель на функцию, `*pf` — это функция. Следовательно нужно использовать конструкцию `(*pf)()` в качестве обращения к функции. Вторая философская школа утверждает, что, поскольку имя функции является указателем на эту функцию, указатель на эту функцию должен вести себя как имя функции. Следовательно, нужно использовать `pf()` в качестве обращения к функции. C++ занимает компромиссную позицию, признавая корректность обеих форм или их право на существование, даже если они логически несовместимы друг с другом. Прежде чем решительно осудить этот компромисс, подумайте, что способность придерживаться взглядов, которые не согласуются между собой, является отличительным свойством умственной деятельности человека.

В программе из листинга 7.15 показано, как используются указатели на функции. Программа вызывает функцию `estimate()` дважды: один раз для передачи адреса функции `betsy()` и один раз для передачи адреса функции `pam()`. В первом случае функция `estimate()` для вычисления необходимого количества часов использует функцию `betsy()`, а во втором случае — функцию `pam()`. Такой подход существенно облегчает дальнейшую разработку программы. Когда Ральф (Ralph) разрабатывает свой собственный алгоритм оценки времени, ему уже не нужно переделывать функцию `estimate()`. Вместо этого ему вполне достаточно применить собственную функцию `ralph()`, предварительно убедившись в правильности сигнатуры и типа возвращаемой величины. Разумеется, переписывание функции `estimate()` не представляет собой особых трудностей, но эти же принципы применимы и к более сложным программным кодам. Кроме того, метод использования указателей на функции позволяет Ральфу внести корректиды в поведение функции `estimate()`, даже если он не имеет доступа к исходным кодам функции `estimate()`.

### Листинг 7.15 Программа fun\_ptr.cpp.

---

```
// fun_ptr.cpp -- указатели на функции
#include <iostream>
using namespace std;
double betsy(int);
double pam(int);

// второй аргумент является указателем на
// функцию типа double, которая принимает
// аргумент типа int
void estimate(int lines, double (*pf)(int));
```

```

int main()
{
 int code;

 cout << "How many lines of code do you
need? ";
 cin >> code;
 cout << "Here's Betsy's estimate:\n";
 estimate(code, betsy);
 cout << "Here's Pam's estimate:\n";
 estimate(code, pam);
 return 0;
}

double betsy(int lns)
{
 return 0.05 * lns;
}

double pam(int lns)
{
 return 0.03 * lns + 0.0004 * lns * lns;
}

void estimate(int lines, double (*pf)(int))
{
 cout << lines << " lines will take ";
 cout << (*pf)(lines) << " hour(s)\n";
}

```

Результаты двукратного выполнения программы:

```

How many lines of code do you need? 30
Here's Betsy's estimate:
30 lines will take 1.5 hour(s)
Here's Pam's estimate:
30 lines will take 1.26 hour(s)

How many lines of code do you need? 100
Here's Betsy's estimate:
100 lines will take 5 hour(s)
Here's Pam's estimate:
100 lines will take 7 hour(s)

```

## Резюме

Функции в C++ представляют собой программные модули. Чтобы воспользоваться функцией, необходимо дать ее определение и составить прототип, после чего можно ее вызывать. Определением функции служит программный код, который выполняет действия, предусмотренные алгоритмом функции. Прототип функции описывает ее интерфейс: сколько и какого типа значения должны быть переданы функции и какого типа возвращаемое значение, если таковое имеется, должна получить от нее вызывающая программа. Обращаясь к функции, программа должна передать ей аргументы и передать управление программному коду функции.

По умолчанию функции C++ передают аргументы по значению. Это означает, что формальные параметры в

определении функции — это новые переменные, которые инициализируются значениями, содержащимися в обращении к функции. Таким образом, функции C++ обеспечивают целостность исходных данных, поскольку используют их копии.

C++ трактует аргумент, представляющий собой имя массива, как адрес первого элемента массива. С формальной точки зрения такую операцию можно считать передачей по значению, поскольку указатель является копией исходного адреса, однако функция использует этот указатель для доступа к содержимому исходного массива. При объявлении формальных параметров функции (и только тогда) следующие два объявления эквивалентны:

```

имяТипа arr[];
имяТипа * arr;

```

В обоих объявлениях **arr** обозначает указатель на **имяТипа**. Однако при написании программного кода функции можно использовать **arr**, как если бы это было имя массива, для доступа к его элементам: **arr[i]**. Даже при передаче указателей вы можете сохранить целостность исходных данных, объявив формальный аргумент указателем на тип **const**. Поскольку передача адреса любого массива не приводит к передаче информации о размере массива, обычно вы должны передать сведения о размере массива в виде отдельного аргумента.

C++ реализовано три вида представления строк: символьный массив, строковая константа и указатель на строку. Все они имеют тип **char\*** (указатель на величину типа **char**), поэтому передаются функции как аргумент типа **char\***. C++ использует пустой символ (\0) для обозначения конца строки, а функции, работающие с символами, производят проверку на наличие пустого символа в целях обнаружения конца любых обрабатываемых ими строк.

C++ применяет к структурам такой же подход, как и к основным типам. Это значит, что вы имеете возможность передавать их по значению и использовать как типы возвращаемых значений. Однако если структура достаточно крупная, то более оптимальной может оказаться передача указателя на структуру, позволяющая функции работать с исходными данными.

Функция в C++ может быть рекурсивной, т.е. программный код такой функции может вызывать сам себя.

Имя функции C++ ведет себя как адрес этой функции. Путем использования аргумента типа функции, являющегося указателем на другую функцию, вы можете передать функции имя второй функции, если вам нужно, чтобы к ней обратилась первая функция.

## Вопросы для повторения

1. Назовите три условия, необходимых для использования функции.
2. Постройте прототипы функций, соответствующие следующим описаниям:
  - a. **igor()** не принимает ни одного аргумента и не возвращает никакого значения.
  - b. **tofu()** принимает аргумент типа **int** и возвращает значение типа **float**.
  - c. **mpg()** принимает два аргумента типа **double** и возвращает значение типа **double**.
  - d. **summation()** принимает имя массива с элементами типа **long** и размер массива по значению и возвращает значение типа **long**.
  - e. **doctor()** принимает строковый аргумент (эта строка модификации не подлежит) и возвращает значение типа **double**.
  - f. **ofcourse()** принимает структуру **boss** в качестве аргумента, но ничего не возвращает.
  - g. **plot()** принимает в качестве аргумента указатель на структуру **map** и возвращает строку.
3. Создайте функцию, которая принимает три аргумента: имя массива элементов типа **int**, размер массива и значение типа **int**. Функция должна установить для каждого элемента массива значение типа **int**.
4. Создайте функцию, которая принимает в качестве аргументов имя массива элементов типа **double** и размер массива и возвращает наибольшее значение из элементов этого массива. Обратите внимание на то, что данная функция не должна менять содержимое массива.
5. Почему мы не используем спецификатор **const** для аргументов функции, которые имеют базовые типы данных?
6. Программа, представленной в листинге 7.7, использует отрицательное значение стоимости имущества для прерывания выполнения цикла. Теперь представьте себе, что она использует нечисловые данные для прерывания цикла. Переделайте функцию **fill\_array()** таким образом, чтобы она решала эту задачу.
7. Какими являются три формы, которые строка в стиле C может принимать в программе, написанной на C++?
8. Создайте функцию, имеющую следующий прототип:
 

```
int replace(char * str, char c1, char c2);
```

Пусть функция заменит каждый символ **c1**, встретившийся в строке **str**, на символ **c2**, и возвращает число сделанных ею замен.

9. Что означает выражение `""pizza"`? Что можно сказать о `"taco"[2]`?
10. C++ предоставляет вам возможность передавать структуры по значению и позволяет передавать адрес структуры. Если **glitz** является переменной типа структура, как вы передадите ее по значению? Как вы передадите ее адрес? Каким может быть компромисс между двумя подходами?
11. Функция **judge()** возвращает значение типа **int**. В качестве аргументов она запрашивает адрес функции, которая принимает в качестве аргумента указатель на **const char** и возвращает величину типа **int**. Создайте прототип этой функции.

## Упражнения по программированию

1. Создайте программу, которая периодически обращается к вам с просьбой ввести пару чисел и делать это до тех пор, пока одно из чисел этой пары не окажется равным нулю. Для каждой пары программа должна вычислить среднее гармоническое этих чисел. Функция должна возвратить ответ в программу **main()**, которая отображает результат на экране. Гармоническое среднее двух чисел — это обратная величина среднего значения обратных величин этих чисел, которая может быть вычислена следующим образом:  

$$\text{гармоническое среднее} = 2.0 * x * y / (x + y)$$
2. Создайте программу, которая обращается к вам с запросом ввести результаты десяти игр в гольф, которые должны храниться в массиве. Необходимо снабдить пользователя средством прерывания ввода до того, как он введет все десять результатов. Программа должна отображать на экране все результаты в одной строке и сообщать, каким является средний результат. Управление вводом данных, отображение данных и вычисление средних значений выполняют три отдельные функции, ориентированные на работу с массивами.
3. Имеется следующий образец структуры:

```
struct box
{
 char maker[40];
 float height;
 float width;
 float length;
 float volume;
};
```

- a. Создайте функцию, которая передает структуру **box** по значению и отображает значение каждого элемента.
- b. Создайте функцию, которая передает адреса структуры **box** и присваивает элементу **volume** произведение остальных трех измерений.
- c. Создайте простую программу, которая использует все эти три функции.
4. Дать определение рекурсивной функции, которая принимает в качестве аргумента целое число и возвращает факториал этого аргумента. Напоминаем, что 3 факториал, записывается как  $3!$ , что равно  $3 \times 2!$ , при этом  $0! = 1$ , по определению, равен 1. В общем случае  $n! = n * (n - 1)!$ . Проверьте это в программе, которая использует цикл для того, чтобы позволить пользователю вводить различные значения, для которых программа вычисляет факториал.

5. Создайте программу, которая использует следующие функции:

**Fill\_array()** принимает в качестве аргументов имя массива элементов данных типа **double** и размер этого массива. Она требует от пользователя ввести значения типа **double**, чтобы затем поместить эти значения в массив. Она прекращает процесс ввода, когда массив заполнен или когда пользователь вводит нечисловое значение, при этом она возвращает фактическое число введенных элементов данных.

**Show\_array()** принимает в качестве аргумента имя массива элементов типа **double** и размер массива и отображает содержимое массива.

**Reverse\_array()** принимает в качестве аргументов имя массива элементов типа **double** и размер массива и изменяет порядок следования значений, хранимых в массиве, на обратный.

Программа должна заполнить массив, отобразить его на экране и поменять порядок следования всех элементов массива на обратный, за исключением первого и последнего элементов, а затем отобразить полученный массив на экране.

6. Это упражнение служит для приобретения практики написания функций, выполняющий обработку массивов и структур. Ниже представлен каркас программы. Дополните его, снабдив программу описанными ниже функциями.

```
#include <iostream>
using namespace std;

const int SLEN = 30;
struct student {
 char fullname[SLEN];
 // ...
};
```

```
char hobby[SLEN];
int ooplevel;
};

// у функции getinfo() имеется два
// аргумента: указатель на первый элемент
// структуры student и значение типа int,
// представляющее собой число элементов
// массива. Эта функция требует ввода
// данных о студентах и сохраняет эти
// данные в памяти. Она прекращает ввод
// сразу после заполнения массива
// или при получении символа пустой строки
// вместо имени студента. Функция
// возвращает фактическое число заполненных
// элементов массива.
int getinfo(student pa[], int n);

// функция display1() принимает структуру
// student в качестве аргумента и
// отображает ее содержимое
void display1(student st);

// функция display2() принимает адрес
// структуры student в качестве аргумента и
// отображает ее содержимое
void display2(const student * ps);

// функция display3() принимает адрес
// первого элемента массива структур
// student и число элементов массива в
// качестве аргументов и отображает
// содержимое структур
void display3(const student pa[], int n);

int main()
{
 cout << "Enter class size: ";
 int class_size;
 cin >> class_size;
 while (cin.get() != '\n')
 continue;

 student * ptr_stu = new
 student[class_size];
 int entered = getinfo(ptr_stu,
 class_size);
 for (int i = 0; i < entered; i++)
 {
 display1(ptr_stu[i]);
 display2(&ptr_stu[i]);
 }
 display3(ptr_stu, entered);
 delete [] ptr_stu;
 cout << "Done\n";
 return 0;
}
```

7. Разработайте функцию **calculate()**, которая принимает два значения типа **double** и указатель на функцию, который принимает в качестве аргументов два значения типа **double** и возвращает значение типа **double**. Функция **calculate()** также должна иметь тип **double**. Она должна возвратить значение, вычисляе-

мое функцией, на которую указывает указатель. При этом используются значения типа **double**, являющиеся аргументами функции **calculate()**. Например, предположим, что у нас имеется следующее определение функции **add()**:

```
double add(double x, double y)
{
 return x + y;
}
```

Затем вызов функции

```
double q = calculate(2.5, 10.4, add);
```

заставит функцию **calculate()** передать значения 2.5 и 10.4 функции **add()**, а затем передать функции **add()** возвращаемое значение (12.9).

Причтите эти функции, а также, по меньшей мере, еще с дну дополнительную функцию, определенную

в теле функции **add()**, для работы программы. Программа должна использовать цикл, в котором выполняется ввод пар чисел. Для ввода каждой пары значений используйте функцию **calculate()**, чтобы обратиться к функции **add()** или, по меньшей мере, еще к одной функции. Если вас не отпугивают возможные трудности, попытайтесь создать массив указателей на функции типа **add()** и воспользуйтесь циклом для успешного применения **calculate()** к последовательности функций. При этом используйте указатели. Указание: вот как следует объявлять массив такого рода для трех указателей:

```
double (*pf[3])(double, double);
```

Вы можете инициировать подобного рода массив, воспользовавшись обычным синтаксисом инициализации массива и именами функций в качестве адресов.

# Работа с функциями

**В этой главе рассматривается следующее:**

- Встроенные функции
- Ссылочные переменные
- Передача аргументов функции по ссылке
- Аргументы, заданные по умолчанию
- Перегрузка функций
- Шаблоны функций
- Специализация шаблонов функций
- Раздельная компиляция
- Классы памяти, диапазон доступа и связывание
- Пространство имен

Вы уже имеете представление о функциях языка C++, эта глава поможет вам расширить свои знания. C++ вводит в обиход множество новых функциональных средств, благодаря которым он существенно отличается от C. Из числа новых средств следует отметить встроенные функции, средство передачи переменных по ссылкам, значения аргументов, принимаемые по умолчанию, перегрузку функций (полиморфизм) и шаблонные функции. В этой главе подробно рассматриваются все эти усовершенствования. Кроме того, в ней описываются многофайловые программы и многообразные классы памяти языка C++, включая пространства имен.

## Встроенные функции

Начнем с рассмотрения встроенных функций, нового инструментального средства C++, предназначенного для повышения быстродействия программ. Основное различие между обычными и встроенными функциями заключается не в том, как для них разрабатываются программные коды, а в том, как компилятор встраивает их в программу. Чтобы понять различие между обычными и встроенными функциями, нужно изучить особенности разрабатываемой программы гораздо глубже, чем мы это делали до сих пор.

Конечный продукт процесса компиляции — это исполняемая программа, которая состоит из набора инструкций на языке машинных кодов. Когда вы запускаете

программу, операционная система загружает эти инструкции в оперативную память компьютера, таким образом, каждая инструкция имеет свой собственный адрес в памяти. Компьютер последовательно выполняет эти инструкции. Иногда в программе встречается цикл или условный оператор, тогда в ходе выполнения программы пропускаются некоторые инструкции, при этом происходит переход вперед или назад, к конкретному адресу. Обычный вызов функции также влечет за собой переход выполняемой программы на другой адрес (адрес функции) и возврат в прежнее место, когда выполнение функции будет завершено. Рассмотрим подробнее, как обычно протекает этот процесс. Когда программа переходит к инструкции, вызывающей функцию, она запоминает адрес инструкции, которая непосредственно следует за командой вызова функции, копирует аргументы в стек (блок памяти, зарезервированный специально для этой цели), переходит в ячейку памяти, которая представляет собой начало функции, выполняет код функции (при этом она, возможно, помещает возвращаемое значение в регистр), а затем возвращается к выполнению инструкции, адрес которой она запомнила. (Это в какой-то степени напоминает процесс чтения некоторого текста, когда приходится отвлекаться на ознакомление с содержанием сноски, а затем возвращаться в то место текста, где чтение было прервано.) Переход туда и обратно и запоминание со-

ответствующих адресов означает, что имеют место не-производительные затраты времени и ресурсов, обусловленные использованием функций.

Встраиваемая функция C++ предлагает новые возможности. Это такая функция, скомпилированный программный код которой "согласуется" с остальным кодом программы. Иначе говоря, компилятор подставляет вместо вызова функции соответствующий код функции. При наличии подставляемого кода программе уже не надо выполнять переходы в другое место, а затем возвращаться назад. Таким образом, встраиваемые функции выполняются немного быстрее, чем обычные функции, однако за это нужно платить дополнительным расходом памяти. Если в 10 разных местах программа выполняет 10 отдельных вызовов одной и той же функции, то она создает 10 копий этой функции в соответствующих местах программного кода (рис. 8.1).

Нужно подходить к использованию встраиваемых функций выборочно. Выигрыш в быстродействии обычно минимален, если не считать случаев, когда функция сама по себе настолько компактна, что время, необходимое для ее выполнения, сопоставимо со временем, затрачиваемым на переход и возврат из функции. В этом случае функция сама по себе обладает высоким быстродействием, так что максимальный выигрыш по времени вы получите тогда, когда эта функция выполняется в некотором критическом цикле и является основным потребителем машинного времени.

Чтобы воспользоваться этим средством, нужно выполнить следующее:

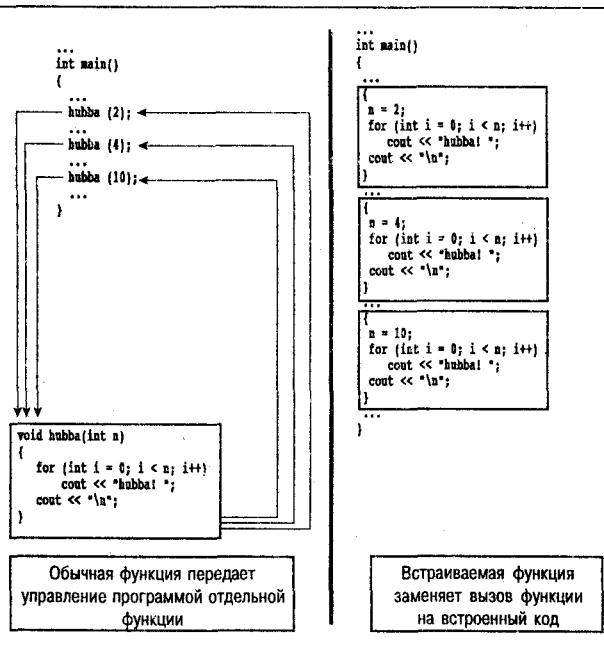


РИСУНОК 8.1 Встроенные и обычные функции.

- Предпослать определению функции ключевое слово `inline`.
- Поместить определение этой функции перед определениями всех функций, которые к ней обращаются.

Обратите внимание на то обстоятельство, что требуется поместить перед определениями других функций полностью все определение (имеется в виду заголовок функции и весь программный код функции), а не только прототип.

Компилятор не обязательно должен исполнять вашу просьбу о том, чтобы придать той или иной функции статус встраиваемой. Он может прийти к заключению, что ваша функция слишком большая, или вдруг обнаружит, что она обращается сама к себе (рекурсия для встраиваемых функций не допускается), к тому же может случиться так, что компилятор, имеющийся в вашем распоряжении, не способен реализовать требуемое свойство.

Программа, представленная в листинге 8.1, иллюстрирует метод встраивания на примере функции `square()`, которая возводит в квадрат свой аргумент. Обратите внимание на то, что все определение функции поместились в одной строке. Это совсем не обязательно, тем не менее, если определение не помещается в одной строке, значит эта функция, скорее всего, не относится к числу тех, которые, получив статус встраиваемых, могут принести какую-то пользу.

#### Листинг 8.1 Программа inline.cpp.

```

// inline.cpp -- использование встраиваемой
// функции
#include <iostream>
using namespace std;

// определение встраиваемой функции должно
// быть дано прежде, чем она будет использована
inline double square(double x) { return x * x; }

int main()
{
 double a, b;
 double c = 13.0;

 a = square(5.0);
 b = square(4.5 + 7.5); // может
 // передавать выражения
 cout << "a = " << a << ", b = "
 << b << "\n";
 cout << "c = " << c;
 cout << ", c squared = " << square(c++)
 << "\n";
 cout << "Now c = " << c << "\n";
 return 0;
}

```

Результаты выполнения этой программы:

```
a = 25, b = 144
c = 13, c squared = 169
Now c = 14
```

Полученные результаты показывают, что встраиваемая функция передает аргументы по значению, как это характерно для обычных функций. Если аргумент представляет собой выражение, такое как, например, 4.5 + 7.5, функция передает значение этого выражения, в рассматриваемом случае оно равно 12. Из этого следует, что средство **inline** языка C++ существенно лучше макроопределений языка С. Ниже приведено примечание, сравнивающее возможности встраивания и макросов.

Даже если в программе нет отдельного прототипа, тем не менее, возможности прототипирования C++ проявляются и в ней. Это объясняется тем, что полное определение функции, которое дается перед тем, как она будет выполнена первый раз, служит в качестве прототипа. Это означает, что вы можете использовать функцию **square()** с аргументом типа **int** или **long** и программа автоматически выполнит приведение полученного значения к типу **double**, прежде чем передавать его функции.

### СРАВНЕНИЕ ВОЗМОЖНОСТЕЙ СРЕДСТВА **INLINE** И МАКРОСОВ

Средство **inline** реализовано только в C++. В С используется оператор препроцессора **#define**, обеспечивающий реализацию макросов, представляющих собой грубое приближение встраиваемого кода. Например, макрос, возводящий целое число в квадрат, имеет вид:

```
#define SQUARE(x) x*x
```

Этот макрос работает не по принципу передачи аргументов, а по принципу подстановки текста, при этом X играет роль символической метки "аргумента":

```
a = SQUARE(5.0); заменяется на a = 5.0*5.0;
b = SQUARE(4.5 + 7.5); заменяется на b = 4.5 +
 7.5 * 4.5 + 7.5;
d = SQUARE(c++); заменяется на d = c++*c++;
```

Только первый пример выполняется так, как надо. Вы можете улучшить ситуацию, снабдив выражение скобками:

```
#define SQUARE(X) ((X)*(X))
```

Но все еще остается проблема, поскольку макрос не передает аргументы по значению. Даже если дать это новое определение, функция **SQUARE(c++)** увеличивает значение **c** на 1 дважды, в то время как встраиваемая функция **square()**, представленная в листинге 8.1, вычисляет **c**, передает полученное значение для возвведения в квадрат, а затем увеличивает значение **c** на 1.

Мы продемонстрировали все это вовсе не для того, чтобы научить вас, как правильно создавать макросы в С. Мы просто хотели, чтобы при использовании макросов С, вы помнили о том, что с пользой для дела их можно преобразовать в C++ во встраиваемые функции.

## Ссылочные переменные

Язык C++ вводит в практику новый производный тип данных — тип ссылочной переменной. Ссылка представляет собой имя, которое является альтернативным или псевдонимом для ранее объявленной переменной. Например, если вы делаете **twain** ссылкой на переменную **clemens**, вы можете поочередно использовать либо **twain**, либо **clemens** для представления этой переменной. В чем смысл использования альтернативного имени? Уж не в том ли, чтобы помочь тем программистам, которые не удовлетворены сделанным ими выбором имен переменных? Вполне возможно, однако основное назначение ссылок — их использование в качестве формальных аргументов функций. Используя ссылку в качестве аргумента, функция работает с исходными данными, а не с их копиями. Ссылки представляют собой удобную альтернативу указателям при обработке крупных структур посредством функций, они широко используются при разработке классов. Однако, прежде чем учиться пользоваться ссылками при работе с функциями, рассмотрим, как правильно давать определения ссылок и использовать их. Следует иметь в виду, что цель предстоящего обсуждения заключается не столько в том, чтобы научить читателя пользоваться ссылками, сколько в том, чтобы показать, как они работают.

### Создание ссылочных переменных

Вы, возможно, еще не забыли, что в языках С и C++ используется символ **&** для обозначения адреса переменной. Язык C++ придает символу **&** дополнительный смысл и внедряет его для объявления ссылок. Например, чтобы **rodents** стало альтернативным именем для переменной **rats**, необходимо сделать следующее:

```
int rats;
int & rodents = rats; // rodents становится
// псевдонимом имени rats
```

В этом контексте **&** не является адресным оператором. В этом случае, он воспринимается как часть идентификатора типа. Подобно тому как **char \*** в объявлении является указателем на значение типа **char**, **int &** является ссылкой на тип **int**. Объявление ссылки позволяет поочередно использовать либо **rats**, либо **rodents**; оба имени ссылаются на одну и ту же ячейку памяти. Программа, представленная в листинге 8.2, — яркое тому подтверждение.

### Листинг 8.2 Программа **firstref.cpp**

```
// firstref.cpp -- определение и использование
// ссылок
#include <iostream>
using namespace std;
int main()
```

```

{
 int rats = 101;
 int & rodents = rats; // rodents - это
 // ссылка

 cout << "rats = " << rats;
 cout << ", rodents = " << rodents
 << "\n";
 rodents++;
 cout << "rats = " << rats;
 cout << ", rodents = " << rodents << "\n";

// некоторые реализации системы требуют
// выполнения преобразования типа следующих
// адресов в тип unsigned
 cout << "rats address = " << &rats;
 cout << ", rodents address = "
 << &rodents << "\n";
 return 0;
}

```

Обратите внимание на то, что оператор **&** в выражении

```
int & rodents = rats;
```

не является адресным, он всего лишь объявляет, что переменная **rodents** имеет тип **int &**, т.е. является ссылкой на переменную типа **int**. Однако оператор **&** в выражении

```
cout << ", rodents address =" << &rodents << "\n";
```

является адресным, при этом **&rodents** представляет собой адрес переменной, на которую ссылается **rodents**. Получаем следующие результаты выполнения программы:

```

rats = 101, rodents = 101
rats = 102, rodents = 102
rats address = 0068FDF4, rodents address =
0068FDF4

```

Нетрудно заметить, что и **rats**, и **rodents** имеют одно и то же значение и один и тот же адрес. Увеличение значения **rodents** на 1 затрагивает обе переменные. Точнее, в результате выполнения операции **rodents++** увеличивается значение одной переменной, у которой имеется два имени. (Имейте в виду, что, несмотря на то что этот пример показывает, как действует ссылка, он не может служить образцом типичного использования ссылки в качестве параметра функции, в частности, в качестве аргумента, представляющего структуру или объект. Мы вскоре рассмотрим эти виды применения ссылок.)

На первых порах освоение ссылок программистами, которые работали в среде С и перешли в среду С++, не проходит гладко, поскольку они сразу никак не могут забыть указатели, хотя между ними имеются различия. Например, вы можете создать как ссылку, так и указатель, чтобы ссылаться на **rats**:

```

int rats = 101;
int & rodents = rats; // rodents - это
 // ссылка

```

```

int * prats = &rats; // prats - это
 // указатель

```

Затем вы можете использовать выражения **rodents** и **\*prats** наряду с **rats** и поочередно пользоваться выражениями **&rodents** и **prats** наряду с **&rats**. С этой точки зрения ссылка мало чем отличается от указателя в замаскированной записи, в рамках которой операция разыменования **\*** предполагается неявно. И фактически это в какой-то степени именно то, чем является ссылка. Но между ними существуют различия (помимо различий в обозначениях). Одним из них является то, что ссылку необходимо инициализировать в момент, когда вы ее объявляете; вы не можете сначала объявить ссылку, а затем присвоить ей значение, как это делается в случае с указателем:

```

int rat;
int & rodent;
rodent = rat; // Вы не можете сделать это.

```

### ПОМНИТЕ

Необходимо инициализировать ссылку в момент ее объявления.

Ссылка во многом аналогична указателю **const**; нужно инициализировать ее в момент создания, и, как только ссылка засвидетельствует "свою верность" по отношению к конкретной переменной, она сохраняет ее до конца выполнения программы. Таким образом, конструкция,

```
int & rodents = rats;
```

по существу, является замаскированной записью

```
int * const pr = &rats;
```

или подобных выражений. В данном случае **rodents** играет ту же роль, что и **\*pr**.

Листинг 8.3 показывает, что может случиться, если вы попытаетесь заставить ссылку переключиться с переменной **rats** на переменную **bunnies**.

### Листинг 8.3 Программа secref.cpp.

```

// secref.cpp -- объявление и использование
// ссылок
#include <iostream>
using namespace std;
int main()
{
 int rats = 101;
 int & rodents = rats; // rodents - это
 // ссылка

 cout << "rats = " << rats;
 cout << ", rodents = " << rodents << "\n";

 cout << "rats address = " << &rats;
 cout << ", rodents address = "
 << &rodents << "\n";
}

```

```

int bunnies = 50;
rodents = bunnies; // можем ли мы
// поменять ссылку?
cout << "bunnies = " << bunnies;
cout << ", rats = " << rats;
cout << ", rodents = " << rodents << "\n";
cout << "bunnies address = " << &bunnies;
cout << ", rodents address = "
 << &rodents << "\n";
return 0;
}

```

Результаты выполнения этой программы:

```

rats = 101, rodents = 101
rats address = 0068FDF4,
rodents address = 0068FDF4
bunnies = 50, rats = 50, rodents = 50
bunnies address = 0068FDF0,
rodents address = 0068FDF4

```

Сначала `rodents` ссылается на `rats`, но затем программа предпринимает попытку сделать `rodents` ссылкой на переменную `bunnies`:

```
rodents = bunnies;
```

В какой-то момент кажется, что эта попытка закончилась удачно, поскольку значение переменной `rodents` изменяется со 101 на 50. Однако при ближайшем рассмотрении оказывается, что и значение `rats` также изменилось и стало равным 50 и что `rats` и `rodents` все еще имеют один и тот же адрес, который отличается от адреса переменной `bunnies`. Поскольку `rodents` является альтернативным именем переменной `rats`, оператор присваивания в действительности эквивалентен следующему выражению:

```
rats = bunnies;
```

Это выражение означает следующее: "присвоить переменной `rats` значение переменной `bunnies`". Одним словом, вы можете установить ссылку путем инициализации объявления, но не путем присваивания значения.

Предположим, что вы попытались выполнить следующее:

```

int rats = 101;
int * pi = &rats;
int & rodents = *pi;
int bunnies = 50;
pi = &bunnies;

```

Инициализация `rodents` значением `*pi` приводит к тому, что `rodents` ссылается на переменную `rats`. Последующая попытка заставить `pi` указывать на `bunnies` не изменяет того факта, что `rodents` ссылается на `rats`.

## Ссылки в роли параметров функции

Чаще всего ссылки используются в качестве параметров функций, при этом имя переменной в функции станов-

ится псевдонимом переменной из вызывающей программы. Такой метод передачи аргументов называется *передачей по ссылке*. Передача параметров по ссылке позволяет вызываемой функции получить доступ к переменным в вызывающей функции. Реализация этого средства в C++ представляет собой дальнейшее развитие основных принципов С, где происходила только передача по значению. Передача по значению, как известно, приводит к тому, что вызываемая функция работает только с копиями значений из вызывающей программы (рис. 8.2). Разумеется, С позволяет обойти ограничения, накладываемые передачей аргументов по значению, предлагая воспользоваться указателями.

Теперь сравним, как используются ссылки и указатели при решении простой компьютерной задачи: обмен значениями двух переменных. Функция обмена должна иметь возможность изменять значения переменных в вызывающей программе. Это означает, что обычный подход, применяемый в таких случаях, здесь неприемлем, поскольку функция закончит работу после того, как выполнит обмен содержимым копий исходных переменных вместо обмена значениями самих переменных. Однако, если передавать ссылки, функция получит возможность работать с исходными данными. С другой стороны, можно передать указатели, чтобы получить доступ к исходным данным. Программа, представленная в листинге 8.4, реализует три метода, включая и тот, который не работает, так что вы сами можете сравнить их.

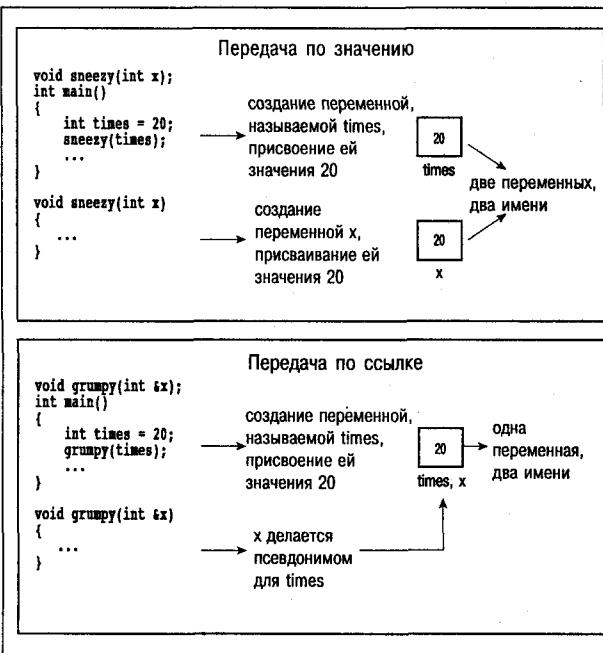


РИСУНОК 8.2 Передача по значению и передача по ссылке.

**Листинг 8.4 Программа swaps.cpp**

```

// swaps.cpp -- обмен со ссылками и
// указателями
#include <iostream>
using namespace std;
void swapr(int & a, int & b); // a, b - псевдонимы переменных типа int
void swapp(int * p, int * q); // p, q - адреса переменных типа int
void swapv(int a, int b); // a, b - новые переменные
int main()
{
 int wallet1 = 300;
 int wallet2 = 350;

 cout << "wallet1 = $" << wallet1;
 cout << " wallet2 = $" << wallet2 << "\n";
 cout << "Using references to swap contents:\n";
 swapr(wallet1, wallet2); // передать переменные
 cout << "wallet1 = $" << wallet1;
 cout << " wallet2 = $" << wallet2 << "\n";
 cout << "Using pointers to swap contents:\n";
 swapp(&wallet1, &wallet2); // передать адреса переменных
 cout << "wallet1 = $" << wallet1;
 cout << " wallet2 = $" << wallet2 << "\n";
 cout << "Trying to use passing by value:\n";
 swapv(wallet1, wallet2); // передать значения переменных
 cout << "wallet1 = $" << wallet1;
 cout << " wallet2 = $" << wallet2 << "\n";
 return 0;
}
void swapr(int & a, int & b) // используются ссылки
{
 int temp;
 temp = a; // использовать a, b для хранения значений переменных
 a = b;
 b = temp;
}
void swapp(int * p, int * q) // используются указатели
{
 int temp;
 temp = *p; // используются *p, *q для хранения значений переменных
 *p = *q;
 *q = temp;
}
void swapv(int a, int b) // попытка использовать значения
{
 int temp;
 temp = a; // использовать a, b для хранения значений переменных
 a = b;
 b = temp;
}

```

Результаты выполнения программы из листинга 8.4:

|                                    |                                      |
|------------------------------------|--------------------------------------|
| wallet1 = \$300 wallet2 = \$350    | " исходные значения"                 |
| Using references to swap contents: |                                      |
| wallet1 = \$350 wallet2 = \$300    | " обмен значениями выполнен"         |
| Using pointers to swap contents:   |                                      |
| wallet1 = \$300 wallet2 = \$350    | " обмен значениями выполнен еще раз" |
| Trying to use passing by value:    |                                      |
| wallet1 = \$300 wallet2 = \$350    | " обмен закончился неудачей"         |

Как мы и ожидали, оба метода — метод указателей и метод ссылок — успешно реализовали обмен содержимым двух бумажников (*wallets*), в то время как метод передачи по значению закончился неудачей.

### Примечания к программе

Прежде всего обратите внимание на то, как вызывается каждая функция:

```
swapr(wallet1, wallet2); // передача
 // переменных
swapp(&wallet1, &wallet2); // передача
 // адресов переменных
swav(wallet1, wallet2); // передача
 // значений переменных
```

Передача аргументов по ссылке (*swapr(wallet1, wallet2)*) и передача по значению (*swav(wallet1, wallet2)*) выглядят идентичными. Единственный способ, позволяющий убедиться в том, что функция *swapr()* передает аргументы по значению, — это обратиться к прототипу или к описанию функции. В то же время благодаря наличию адресного оператора (&) легко распознать, когда функция передает значение по адресу (*swapp(&wallet1, &wallet2)*). (Напоминаем, что объявление типа *int \*p* означает, что *p* — это указатель значения типа *int*, и поэтому аргумент, соответствующий *p*, должен быть адресом, таким как, например, *&wallet1*.)

Далее сравним программные коды функций *swapr()* (передача по ссылке) и *swav()* (передача по значению). Единственное видимое различие между ними состоит в том, как объявлены параметры функции:

```
void swapr(int & a, int & b)
void swav(int a, int b)
```

Внутреннее различие между ними, естественно, состоит в том, что в функции *swapr()* переменные *a* и *b* служат в качестве псевдонимов имен *wallet1* и *wallet2*, так что обмен значениями между *a* и *b* вызывает обмен значениями между *wallet1* и *wallet2*. В то же время в *swav()* переменные *a* и *b* — это новые переменные, которые копируют значения переменных *wallet1* и *wallet2*, в этом случае обмен значениями между *a* и *b* никак не влияет на *wallet1* и *wallet2*.

И наконец, сравним функцию *swapr()* (передает ссылку) и *swapp()* (передает указатель). Первое различие, которое при этом обнаруживается, заключается в том, как объявлены параметры этих функций:

```
void swapr(int & a, int & b)
void swapp(int * p, int * q)
```

Второе различие состоит в том, что вариант с указателем требует применения операции разыменования \* во всех случаях, когда функция использует *p* и *q*.

Раньше мы говорили о том, что требуется инициализировать ссылочную переменную при ее определении.

Вы можете считать, что аргументы ссылочной функции инициализированы значениями, переданными в обращении к функции. А именно, обращение к функции

```
swapr(wallet1, wallet2);
```

инициализирует формальный параметр *a* значением *wallet1* и формальный параметр *b* значением *wallet2*.

### Свойства и особенности ссылок

Для использования ссылочных аргументов характерен ряд особенностей, о которых вам следует знать. Сначала обратимся к листингу 8.5. Программа, представленная в этом листинге, использует две функции для возведения значения аргумента в куб. Одна из них получает аргумент типа *double*, в то время как другая получает ссылку на значение типа *double*. Фактический программный код процедуры возведения в куб выглядит несколько эксцентрично, чтобы более наглядно проиллюстрировать работу с ссылками.

#### Листинг 8.5 Программа cubes.cpp.

```
// cubes.cpp -- регулярные и ссылочные
// аргументы
#include <iostream>
using namespace std;
double cube(double a);
double refcube(double &a);
int main ()
{
 double x = 3.0;
 cout << cube(x);
 cout << " = cube of " << x << "\n";
 cout << refcube(x);
 cout << " = cube of " << x << "\n";
 return 0;
}
double cube(double a)
{
 a *= a * a;
 return a;
}
double refcube(double &a)
{
 ra *= ra * ra;
 return ra;
}
```

Получаем следующий результат:

```
27 = cube of 3
27 = cube of 27
```

Обратите внимание на то, что функция *refcube()* модифицирует значение *x* в *main()*, в то время как функция *cube()* этого не делает, что напоминает нам о том, что передача по значению является нормой. Переменная *a* является локальной для функции *cube()*. Она инициализируется значением *x*, однако изменения, которым подвергается *a*, не отражаются на *x*. Тем не менее, поскольку функция *refcube()* использует в качестве аргу-

мента ссылку, те изменения, которым она подвергает переменную `ra`, фактически отражаются и на `x`. Если вы хотите, чтобы функция использовала информацию, которая ей передается, и не изменяла ее, и если при этом вы желаете пользоваться ссылками, то вам потребуется прибегнуть к помощи постоянной ссылки. В рассматриваемом случае, например, может потребоваться воспользоваться спецификатором `const` в прототипе функции и в заголовке функции:

```
double refcube(const double &ra);
```

Если бы мы это сделали, компилятор отображал бы сообщение об ошибке всякий раз, когда обнаруживал код, изменяющий значение `ra`.

Между прочим, если вам нужно создать функцию с использованием строк рассматриваемого примера, выполните передачу аргумента по значению, а не более экстравагантную передачу по ссылке. Ссылочные аргументы становятся полезными по мере увеличения размеров элементов данных, таких как структуры и классы, в чем вы вскоре сами убедитесь.

Функции, которые осуществляют передачу данных по значению, такие как, например, функция `cube()`, представленная в листинге 8.5, могут использовать фактические аргументы многих видов. Например, все приводимые ниже вызовы функции допустимы:

```
double z = cube(x + 2.0); // вычислить
 // x + 2.0, передать значение
z = cube(8.0); // передать значение 8.0
int k = 10;
z = cube(k); // привести значение k к
 // типу double, передать значение
double yo[3] = { 2.2, 3.3, 4.4 };
z = cube(yo[2]); // передать значение 4.4
```

Предположим, что вы делаете попытку использовать аналогичные аргументы для функции со ссылочными параметрами. Создается впечатление, что передача ссылки сопряжена с большими ограничениями. В конце концов, если `ra` является альтернативным именем переменной, то фактическим аргументом должна быть именно эта переменная. Подобное выражение

```
double z = refcube(x + 3.0); // может
 // привести к ошибке при компиляции
```

по-видимому, не имеет смысла, поскольку выражение `x + 3.0` не является переменной. Например, вы не можете присвоить значение такому выражению:

```
x + 3.0 = 5.0; // не имеет смысла
```

Что произойдет, если вы попробуете выполнить обращение к функции, подобное следующему: `refcube(x + 3.0)`? В современном C++ это ошибка, и некоторые компиляторы не преминут сообщить вам об этом. Другие отобразят предупреждение примерно такого содержания:

```
Warning: Temporary used for parameter 'ra'
in call to refcube(double &)
```

Причина того, что это сообщение звучит не совсем категорично, заключается в том, что C++ в первые годы своего становления допускал передачу выражений типа ссылочной переменной. В некоторых случаях это допускается и сейчас. А происходит следующее: поскольку `x + 3.0` не является переменной типа `double`, программа создает временную переменную, не имеющую имени, инициализируя ее значением выражения `x + 3.0`. Затем `ra` становится ссылкой на эту временную переменную. Рассмотрим временные переменные более подробно и выясним, в каких случаях они создаются и в каких — нет.

### **Временные переменные, ссылочные аргументы и модификатор `const`**

C++ может создавать временную переменную, если фактический аргумент не соответствует ссылочному. В настоящее время C++ допускает это, только когда аргументом является ссылка `const`, но это опять-таки новое ограничение. Рассмотрим те случаи, когда C++ генерирует временную переменную, и выясним, почему целесообразно вводить ограничение, требующее, чтобы ссылка использовалась со спецификатором `const`.

Прежде всего, в каких случаях создается временная переменная? При условии, что ссылочным параметром является `const`, компилятор генерирует временную переменную в двух случаях:

- Тип фактического аргумента выбран правильно, но сам параметр не является LValue (L-значение)
- Тип фактического параметра выбран неправильно, но этот тип может быть преобразован в корректный тип

Аргумент, другими словами, *L-значение*, представляет собой объект данных, который можно снабжать ссылками. Например, переменная, элемент массива, элемент структуры, ссылка и разыменованный указатель — все они являются L-значениями. К L-значениям не относятся буквенные константы и выражения, состоящие из нескольких термов. Например, предположим, что мы переопределели функцию `refcube()`, так что у нее имеется аргумент в виде ссылочной константы:

```
double refcube(const double &ra)
{
 return ra * ra * ra;
}
```

Теперь рассмотрим следующий программный код:

```
double side = 3.0;
double * pd = &side;
double & rd = side;
long edge = 5L;
double lens[4] = { 2.0, 5.0, 10.0, 12.0 };
```

```

double c1 = refcube(side); // ra - это
 // side
double c2 = refcube(lens[2]); // ra -
 // lens[2]
double c3 = refcube(rd); // ra - rd есть
 // side
double c4 = refcube(*pd) // ra - *pd есть
 // side
double c4 = refcube(edge); // ra -
 // временная переменная
double c5 = refcube(7.0); // ra - временная
 // переменная
double c6 = refcube(side + 10.0); // ra -
 // временная переменная

```

Аргументы `side`, `lens[2]`, `rd` и `*pd` — именованные объекты данных типа `double`, и это обстоятельство позволяет генерировать ссылки на них, при этом не нужны временные переменные. (Напомним, что элемент массива ведет себя так же, как и переменная, имеющая тот же тип, что и элемент массива.) Однако объект `edge`, будучи переменной, имеет другой тип. Ссылка на объект типа `double` не может именовать объект типа `long`. С другой стороны, аргументы `7.0` и `side + 10.0` имеют нужный тип, но не являются именованными объектами данных. В каждом из этих случаев компилятор генерирует временную анонимную переменную и превращает `ra` в ссылку на эту переменную. Такие временные переменные существуют не дольше, чем само обращение к функции, после чего компилятор волен распоряжаться ими по собственному усмотрению.

Почему такое поведение вполне оправдано для ссылочных констант и недопустимо в других случаях? Возвратимся к функции `swapr()` из программы, представленной в листинге 8.4:

```

void swapr(int & a, int & b)
 // используются ссылки
{
 int temp;
 temp = a; // a, b используются для
 // хранения значений переменных
 a = b;
 b = temp;
}

```

Что произойдет, если мы выполним представленный ниже программный код в духе менее жестких правил ранних версий C++?

```

long a = 3, b = 5;
swapr(a, b);

```

Здесь имеет место несоответствие типов, поэтому компилятор создает две временные переменные типа `int`, инициализирует их значениями 3 и 5, а затем производит обмен содержимым временных переменных, оставляя при этом значения `a` и `b` неизменными.

Одним словом, если назначение функции со ссылочными аргументами состоит в том, чтобы модифицировать переменные, передаваемые в качестве аргументов, ситуация, которую порождают временные переменные,

препятствуют достижению этой цели. В этом случае решение заключается в том, чтобы запретить создание временных переменных, и именно это делает сейчас C++.

Теперь рассмотрим функцию `refcube()`. В ее задачу входит просто использование переданных ей значений без их модификации. В этом случае временные переменные не приносят никакого вреда, они придают функции более универсальный характер в смысле разнообразия аргументов, с которыми она способна работать. Следовательно, объявление функции устанавливает, что ссылка имеет тип `const`, C++ генерирует временные переменные там, где это необходимо. По существу, функция C++ с формальным аргументом в виде ссылки `const` и с несоответствующим фактическим аргументом имитирует традиционные действия, выполняемые при стандартной передаче значений. При этом она гарантирует то, что исходные данные не подвергнутся изменениям, при которых применяется временная переменная для хранения соответствующего значения.

#### ПОМНИТЕ

Если аргумент в обращении к функции не является L-значением или несовместим по типу с соответствующим параметром ссылки `const`, C++ создает анонимную переменную нужного типа, присваивает значение аргумента обращения к функции анонимной переменной и получает параметр, ссылающийся на эту переменную.

#### ИСПОЛЬЗУЙТЕ МОДИФИКАТОР CONST ТАК, ГДЕ ЭТО ВОЗМОЖНО

Имеются три "железных" довода в пользу объявления ссылочных аргументов в качестве ссылки на константы:

- Применение спецификатора `const` предотвращает появление программных ошибок, которые могут изменить данные.
- Применение спецификатора `const` позволяет функции выполнять обработку формальных аргументов как со спецификатором `const`, так и без него, в то время как функция, в прототипе которой спецификатор `const` опущен, может принимать только данные, не имеющие статуса `const`.
- Использование ссылки с модификатором `const` позволяет функции генерировать и использовать временные переменные по своему усмотрению.

Рекомендуется объявлять формальные ссылочные аргументы со статусом `const` во всех случаях, когда для этого есть возможность.

#### Использование ссылок при работе со структурами

Ссылки очень хорошо сочетаются со структурами и классами, т.е. с типами данных, которые в C++ объявляет пользователь. Собственно говоря, ссылки были введены прежде всего для использования именно с этими типами, а не с основными встроенными типами данных.

Метод использования ссылок на структуры ничем не отличается от метода использования ссылок на перемен-

ные базовых типов; для этого просто достаточно воспользоваться оператором ссылки & при объявлении параметра структуры. Программа, представленная в листинге 8.6, именно это и делает. При этом выявляется интересная особенность, которая заключается в том, что функция возвращает ссылку. Это дает возможность использовать обращение к функции в качестве ее аргумента. Подобное утверждение справедливо для любой функции, возвращающей значение. Но при этом обеспечивается возможность присвоения значения при вызове функции, а это возможно только при использовании типа данных, возвращающего ссылку. Мы дадим пояснения по всем этим вопросам после анализа результатов выполнения программы. В рассматриваемой программе используется функция `use()`, которая отображает два элемента структуры и увеличивает на единицу значение третьего элемента. Таким образом, третий элемент может фиксировать, сколько раз конкретная структура подвергалась обработке со стороны функции `use()`.

### Листинг 8.6 Программа strref.cpp.

```
// strref.cpp -- использование ссылок на
// структуру
#include <iostream>
using namespace std;
struct sysop
{
 char name[26];
 char quote[64];
 int used;
};
sysop & use(sysop & sysopref); // функция с
 // типом возврата ссылки
int main()
{
 // Примечание: некоторые версии системы
 // требуют использования ключевого слова
 // static в двух объявлениях структуры для
 // того, чтобы стала возможной инициализация
 sysop looper = {
 "Rick\\"Fortran\" Looper",
 "I'm a goto kind of guy.", 0
 };
 use(looper); // looper имеет тип sysop
 cout << looper.used << " use(s)\n";
 use (use(looper)); // use(looper) имеет
 // тип sysop
 cout << looper.used << " use(s)\n";
 sysop morf = {
 "Polly Morf",
 "Polly's not a hacker.", 0
 };
 use(looper) = morf; // может присвоить
 // функции!
 cout << looper.name << " says:\n"
 << looper.quote << '\n';
 return 0;
}
```

```
// use() возвращает ссылку, переданную ей
sysop & use(sysop & sysopref)
{
 cout << sysopref.name << " says:\n";
 cout << sysopref.quote << "\n";
 sysopref.used++;
 return sysopref;
}
```

Результаты выполнения программы:

```
Rick "Fortran" Looper says:
I'm a goto kind of guy.
1 use(s)
Rick "Fortran" Looper says:
I'm a goto kind of guy.
Rick "Fortran" Looper says:
I'm a goto kind of guy.
3 use(s)
Rick "Fortran" Looper says:
I'm a goto kind of guy.
Polly Morf says:
Polly's not a hacker.
```

### Примечание к программе

Программа охватывает три новые области. В первой области используется ссылка на структуру, что наглядно показывает первый вызов функции.

```
use(looper);
```

При этом передается структура `looper` по ссылке функции `use()`, благодаря чему `sysopref` становится синонимом структуры `looper`. Когда функция `use()` отображает элементы `name` и `quote` структуры `sysopref`, она на самом деле отображает элементы структуры `looper`. Кроме того, когда функция увеличивает значение `sysopref.used` на 1, она фактически увеличивает значение `looper.used`, как это видно из выходных данных программы:

```
Rick "Fortran" Looper says:
I'm a goto kind of guy.
1 use(s)
```

Во второй новой области используется ссылка в качестве возвращаемого значения. Обычно механизм возврата копирует возвращаемое значение во временную область памяти, к которой вызывающая программа затем осуществляет доступ. Возвращение ссылки, однако, означает, что вызывающая программа осуществляет прямой доступ к возвращаемому значению без использования приготовленной заранее копии. В обычных случаях такая ссылка означает ссылку, передаваемую функции на первом месте, так что вызывающая функция фактически сворачивается, направляя обращаясь к одной из своих собственных переменных. Здесь, например, `sysopref` ссылается на `looper`, в результате чего возвращаемым значением становится исходное значение переменной `looper` в функции `main()`.

Поскольку функция `use()` возвращает ссылку типа `sysop`, она может быть использована в качестве аргумента

та для любой функции, ожидающей поступления либо аргумента `sysop`, либо аргумента, представляющего собой ссылку на аргумент `sysop`, такого, как, например, сама функция `use()`. Таким образом, следующий вызов функции в программе из листинга 8.6 фактически представляет собой два вызова функции, когда значение, возвращаемое первой функцией, служит аргументом для второй функции:

```
use(use(looper));
```

Внутренняя функция выполняет вывод на печать элементов `name` и `quote` и увеличивает величину элемента `used` до значения, равного 2. Функция возвращает значение `sysopref`, уменьшая то, что осталось, до такого значения:

```
use(sysopref);
```

Поскольку `sysopref` является ссылкой на `looper`, то вызов этой функции эквивалентен следующему вызову

```
use(looper);
```

Таким образом, функция `use()` снова отображает два строковых элемента и увеличивает значение элемента `used` до 3.

### ПОМНИТЕ

Функция, которая возвращает ссылку, фактически является псевдонимом переменной, на которую производится ссылка.

Третья область, которая исследуется в программе, касается того, что вы можете присваивать значение функции, если эта функция имеет возвращаемое значение типа ссылки:

```
use(looper) = morf;
```

Для возвращаемых значений, которые не являются ссылками, такое присваивание представляет собой синтаксическую ошибку, но для функции `use()` оно таковой не является. Таков порядок событий. Во-первых, функция `use()` вычисляется. Это значит, что `looper` передается функции `use()` по значению. Как обычно, эта функция отображает два элемента и увеличивает значение элемента `used` до 4. Затем эта функция возвращает ссылку. Поскольку возвращаемое значение ссылается на `looper`, завершающее действие становится эквивалентным следующему оператору:

```
looper = morf;
```

C++ позволяет присваивать одну структуру другой. Таким образом, в результате выполнения этой операции осуществляется копирование содержимого структуры `morf` в `looper`. Это становится ясно после того, как при отображении `looper.name` на экран выводится имя `Morf`, а не имя `Looper`. Одним словом, оператор

```
use(looper) = morf; // возвращаемое
// значение — это ссылка на looper
```

эквивалентен следующим действиям:

```
use(looper);
looper = morf;
```

### ПОМНИТЕ

Вы можете присвоить значение (включая структуры и классы или объекты типа класс) некоторой функции C++, только если эта функция возвращает ссылку на переменную или (в более общем случае) на объект данных. В этом случае значение присваивается переменной или объекту данных, на которые уже имеется ссылка.

Это еще одно свойство, которое обеспечивает возможность использования некоторой формы переопределения оператора. Вы можете воспользоваться ею, например, для переопределения оператора `[]` индексов массива для класса, который определяет более крупный массив.

### **Некоторые соображения по вопросу о том, когда возвращать ссылку или указатель**

Если функция возвращает ссылку или указатель на объект данных, то лучше, чтобы этот объект существовал и после того, когда выполнение функции закончится. Наиболее простой способ, позволяющий добиться этого, состоит в том, чтобы заставить функцию возвращать ссылку или указатель, который был ей передан ранее в качестве аргумента. Таким образом, эта ссылка или указатель уже ссылается на какой-либо объект в вызывающей программе. Функция `use()` из листинга 8.6 использует именно такой прием.

Второй метод состоит в использовании спецификатора `new` для создания новой области памяти. Вы уже сталкивались с примерами, в которых `new` отводит пространство памяти под строку, а функция возвращает указатель на эту область. Вот как вы можете создать нечто подобное следующей ссылке:

```
sysop & clone(sysop & sysopref)
{
 sysop * psysop = new sysop;
 *psysop = sysopref; // скопировать
 // info
 return *psysop; // возвратить ссылку
 // копии
}
```

Первый оператор создает неименованную структуру `sysop`. Указатель `psysop` указывает на эту структуру, следовательно, `*psysop` — это структура. Из программного кода как будто следует, что возвращаемым значением является структура, однако из объявления функции можно сделать вывод, что в действительности функция возвращает ссылку на эту структуру. Теперь вы можете использовать функцию следующим образом:

```
sysop & jolly = clone(looper);
```

Благодаря этому `jolly` становится ссылкой на новую структуру. При использовании такого подхода возникает проблема, которая заключается в том, что требуется использовать оператор `delete`, чтобы высвободить память, выделенную спецификатором `new`, когда в ней отпадает необходимость. Обращение к функции `clone()` скрывает в себе обращение к `new`, из-за чего возрастает вероятность того, что вы забудете воспользоваться оператором `delete` позднее. Шаблон `auto_ptr`, который рассматривается в главе 15 поможет автоматизировать процесс высвобождения памяти.

Желательно избегать использования кода, заключенного в следующих строках:

```
sysop & clone2(sysop & sysopref)
{
 sysop newguy; // первый шаг к
 // большой ошибке
 newguy = sysopref; // копировать info
 return newguy; // возвратить
 // ссылку копии
}
```

Все это производит неблагоприятный эффект возвращения ссылки на временную переменную (`newguy`), которая прекращает свое существование, как только выполнение функции завершится. (В данной главе продолжительность существования различных видов переменных рассматривается в главе, посвященной классам памяти.) Подобным же образом следует избегать возвращения указателей на такие временные переменные.

### **Когда имеет смысл пользоваться ссылочными аргументами**

Использование ссылочных аргументов имеет два больших преимущества:

- Предоставляет возможность вносить изменения в объекты данных в вызывающей функции.
- Позволяет повысить быстродействие программы, передавая ей ссылку вместо всего объекта данных.

Второе преимущество бывает особенно ценно при работе с большими объектами данных, такими как структуры и объекты классов. Эти два преимущества являются также причиной использования указателей в качестве аргументов. Такое использование целесообразно, поскольку ссылочные аргументы, по сути, можно рассматривать как еще один интерфейс программного кода, использующего указатели. Возникают следующие вопросы. Когда следует пользоваться ссылкой? Когда применять указатель? Передавать аргумент по значению? Ниже приводим некоторые рекомендации по всем этим вопросам.

Функция использует передаваемые ей данные, не подвергая их изменениям, в следующих случаях:

- Если размеры объекта данных невелики, например, встроенный тип данных или структура небольших размеров, происходит их передача по значению.
- Если объектом данных является массив, используйте указатель, поскольку это для вас единственный приемлемый вариант. Сделайте этот указатель указателем на `const`.
- Если объект данных является большой структурой, то, чтобы повысить эффективность программы, используйте указатель `const` или ссылку `const`. Вы сэкономите время и пространство памяти, необходимое для создания копии структуры или для построения класса. Назначьте этому указателю или ссылке статус `const`.
- Если объект данных представляет собой объект класса, воспользуйтесь ссылкой `const`. Семантика проекта класса часто требует использования ссылки, что и послужило главной причиной появления в C++ этого инструментального средства. Таким образом, стандартный способ передачи объекта класса в качестве аргумента — это передача по значению.

Функция модифицирует данные для вызывающей функции:

- Если объект данных является встроенным типом данных, воспользуйтесь указателем. Если вам встретится программный код, подобный `fixit(&x)`, где `x` — это величина типа `int`, то есть все основания полагать, что данная функция предназначена для модификации `x`.
- Если объектом данных является массив, используйте единственно возможный для вас выбор, т.е. указатель.
- Если объектом данных является структура, используйте ссылку или указатель.
- Если объектом данных является объект класса, используйте ссылку.

Разумеется, это всего лишь общие рекомендации, но могут быть и особые причины для использования того или иного варианта. Например, `cin` использует ссылки на базовые типы, в этом случае вы можете воспользоваться выражением `cin >> n` вместо `cin >> &n`.

### **Аргументы, заданные по умолчанию**

Теперь рассмотрим еще одно средство из набора новых инструментальных средств C++ — *аргументы, заданные по умолчанию*. Такие аргументы представляют собой значения, которые используются автоматически, если вы пропускаете соответствующий фактический параметр в обращении к функции. Например, если вы устанавливаете `void wow(int n)`, так что `n` по умолчанию имеет значение 1, то в этом случае обращение к функции `wow()` — это то же самое, что и вызов `wow(1)`. Это свойство

придает вашим действиям больше гибкости при работе с функциями. Предположим, вы используете функцию с именем **left()**, которая возвращает первые **n** символов строки, при этом сама строка и число **n** являются аргументами. Точнее, функция возвращает указатель на новую строку, представляющую собой избранный фрагмент исходной строки. Например, в результате вызова функции **left("theory", 3)** создается новая строка "the" и возвращается указатель на нее. Теперь предположим, что вы установили по умолчанию значение второго аргумента равным 1. Обращение **left("theory", 3)** сработает как и раньше, но только выбранное вами значение 3 перекроет значение, принятое по умолчанию. Однако вызов **left("theory")** теперь уже не будет ошибочным, при этом предполагается, что значение второго аргумента равно 1 и возвращается указатель на строку "t". Этот вид выбора значений по умолчанию может оказаться полезным для вашей программы, если она часто извлекает из строки один символ и только время от времени извлекает строки большей длины.

Как установить значение по умолчанию? Для этой цели следует воспользоваться прототипом функции. Поскольку компилятор использует прототип, чтобы узнать, сколько аргументов имеет функция, прототип функции должен сообщить программе о возможности использования аргументов, заданных по умолчанию. Используемый метод состоит в том, что значения аргументам назначаются в прототипе. Например, пусть имеется прототип, соответствующий такому описанию функции **left()**:

```
char * left(const char * str, int n = 1);
```

Мы хотим, чтобы эта функция возвращала новую строку, поэтому ее типом будет **char\***, т.е. указатель на значение типа **char**. Если нужно сохранить исходную строку неизменной, следует использовать спецификатор **const** для первого аргумента. А чтобы аргумент **n** имел значение 1, заданное по умолчанию, присвоим это значение аргументу **n**. Значение, принимаемое аргументом по умолчанию, — это значение, задаваемое при инициализации. Таким образом, прототип, представленный выше, инициализирует **n** значением 1. Если вы игнорируете аргумент **n**, он сохраняет значение 1, но если вы передадите соответствующий аргумент, то новое значение перекроет значение 1.

Если вы используете функцию со списком аргументов, нужно присваивать значения, заданные по умолчанию, в направлении справа налево. Иначе говоря, вы не можете задать значение по умолчанию некоторому конкретному аргументу, пока не зададите значения, определенные по умолчанию, для всех остальных аргументов, размещенных справа:

```
int harpo(int n, int m = 4, int j = 5);
 // ПРАВИЛЬНО
int chico(int n, int m = 6, int j);
 // НЕПРАВИЛЬНО
int groucho(int k = 1, int m = 2, int n = 3);
 // ПРАВИЛЬНО
```

Например, прототип функции **harpo()** допускает реализацию вызова функции с одним, двумя или тремя аргументами:

```
beeps = harpo(2); // то же, что и
 // harpo(2,4,5)
beeps = harpo(1,8); // то же, что и (1,8,5)
beeps = harpo (8,7,6); // ни одно из
 // значений аргумента, заданных по
 // умолчанию, не используется
```

Фактические аргументы присваиваются соответствующим формальным аргументам в направлении "слева направо"; вы не можете пропустить ни одного аргумента. Таким образом, следующее выражение недопустимо:

```
beeps = harpo(3, ,8); // неправильно,
 // аргументу m не присвоено значение 4
```

Аргументы, заданные по умолчанию, не являются каким-то особо выдающимся достижением в программировании; они, скорее, обеспечивают удобство программирования. Когда вы перейдете к проектированию классов, вы убедитесь в том, что они могут сократить число методов конструктора, которые требуется определить.

Программа, представленная в листинге 8.7, находит применение аргументам, заданным по умолчанию. Обратите внимание на то обстоятельство, что только в прототипе указаны значения по умолчанию. Определение функции то же, каким оно было бы и без аргументов, заданных по умолчанию.

### Листинг 8.7 Программа left.cpp.

```
// left.cpp - строковая функция с аргументами,
// заданными по умолчанию
#include <iostream>
using namespace std;
const int ArSize = 80;
char * left(const char * str, int n = 1);
int main()
{
 char sample[ArSize];
 cout << "Enter a string:\n";
 cin.get(sample,ArSize);
 char *ps = left(sample, 4);
 cout << ps << "\n";
 delete [] ps; // освободить старую
 // строку
 ps = left(sample);
 cout << ps << "\n";
 delete [] ps; // освободить новую
 // строку
 return 0;
}
// Эта функция возвращает указатель на новую
// строку, содержащую первые n символов строки
// str.
```

```

char * left(const char * str, int n)
{
 if(n < 0)
 n = 0;
 char * p = new char[n+1];
 int i;
 for (i = 0; i < n && str[i]; i++)
 p[i] = str[i]; // скопировать
 // символы
 while (i <= n)
 p[i++] = '\0'; // заполнить
 // оставшуюся часть строки символами '\0'
 return p;
}

```

Результаты выполнения программы:

```

Enter a string:
forthcoming
fort
f

```

### Примечания к программе

Программа использует оператор `new` для создания новой строки, в которой будут храниться выбранные символы. Одно из нежелательных последствий заключается в том, что какой-нибудь пользователь может затребовать отрицательное число символов. В этом случае функция сбрасывает счетчик символов к 0 и в конечном итоге возвращает пустую строку. Еще одна нежелательная возможность состоит в том, что пользователь может затребовать число символов, превышающее то число, которое содержит исходная строка. Функция защищена от такого исхода тем, что выполняет комбинированную проверку:

```
i < n && str[i]
```

В результате осуществления проверки `i < n` выполнение цикла прекратится сразу после того, как будут скопированы `n` символов. Вторая часть условия проверки, выражение `str[i]`, представляет собой код для символа, который вот-вот будет скопирован. Если цикл достигнет пустого символа, кодом которого является нуль, выполнение цикла прекратится. Заключительный цикл `while` завершает обработку строки тем, что помещает в ее конец символ пробела и заполняет остаток выделенного под строку пространства памяти, если такое имеется, пустыми символами.

Другой способ определения размера новой строки заключается в том, чтобы заданная величина `n` не превышала переданное значение и длину строки:

```

int len = strlen(str);
n = (n < len) ? n : len; // наименьшее из
 // n и len
char * p = new char[n+1];

```

Это гарантирует, что оператор `new` не выделит объем памяти, превышающий необходимый для хранения строки. Это может оказаться полезным в тех случаях, когда вы обращаетесь к функции подобным образом:

`left("Hi!", 32767)`. В рамках первого подхода строка "Hi!" копируется в массив размером 32767 символов, при этом всем символам, за исключением первых трех, присваиваются пустые символы. При реализации второго подхода строка "Hi!" копируется в массив, состоящий из четырех символов. Но в то же время добавление в программу еще одного вызова функции (`strlen()`) приводит к увеличению размеров программы, замедляет процесс ее выполнения и требует, чтобы вы не забыли включить заголовочный файл `cstring` (или `string.h`). Программисты, работающие в среде C, предпочитают иметь более компактный программный код, обеспечивающий высокое быстродействие программы, вследствие чего на них ложится ответственность за правильное использование функций. Однако в C++ основное значение традиционно придается надежности программного продукта. В конце концов, несколько более медленная, зато правильно выполняющаяся программа лучше, чем быстродействующая программа, которая работает неправильно. Если время, затраченное на вызов функции `strlen()`, неприемлемо, вы можете заставить саму функцию `left()` непосредственно выбрать наименьшее из значений `n` и длины строки. Например, представленный ниже цикл прекращает свое выполнение, когда величина `m` станет равной значению `n` или длине строки, независимо от того, что из них меньше:

```

int m = 0;
while (m <= n && str[m] != '\0')
 m++;
char * p = new char[m+1];
// использовать m вместо n до конца
// программного кода

```

### Полиморфизм функций (перегрузка функций)

Полиморфизм функций — это расширение возможностей языка C, реализованное в C++. Аргументы, заданные по умолчанию, позволяют вам обратиться к одной и той же функции, воспользовавшись с этой целью различным числом аргументов. А *полиморфизм функций*, который еще называют *перегрузкой функций*, предоставляет вам возможность воспользоваться несколькими функциями, имеющими одно и то же имя. Слово "полиморфизм" означает "иметь множество форм", следовательно, полиморфизм функций позволяет функции иметь множество форм. Аналогично выражение "перегрузка функций" означает, что вы можете привязать более чем одну функцию к одному и тому же имени. Оба выражения означают одно и то же, но мы будем пользоваться выражением "перегрузка функций" — оно звучит более четко. Вы можете воспользоваться перегрузкой функции для проектирования семейства функций, ко-

торые фактически делают одно и то же, но с различными списками аргументов.

Выражение "перегруженные функции" имеют несколько значений. Например, мисс Питти может искать площадку для игры в гольф своей команде и рыться в земле в поисках трюфелей (два значения слова root). Контекст вам подскажет (можно надеяться), какое из значений подходит для того или иного случая. Аналогично C++ использует контекст, чтобы решить, какой версией перегруженной функции следует воспользоваться.

Главную роль в использовании перегрузки функций играет список аргументов функции, который еще называют *сигнатурой функции*. Если две функции используют одно и то же число и типы аргументов в одном и том же порядке, то они имеют одну и ту же сигнатуру, при этом имя функции не имеет значения. C++ предоставляет вам возможность определить две функции под одним и тем же именем при условии, что эти функции обладают разными сигнатурами. Сигнатуры могут различаться по числу аргументов или по типам аргументов либо по тому и другому. Например, вы можете определить набор функций print() со следующими прототипами:

```
void print(const char * str, int width);
// #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(const char *str); // #5
```

Если затем вы воспользуетесь функцией print(), компилятор сопоставит используемый вами вариант с прототипом, имеющим ту же сигнатуру:

```
print("Pancakes", 15); // используется #1
print("Syrup"); // используется #5
print(1999.0, 10); // используется #2
print(1999, 12); // используется #4
print(1999L, 15); // используется #3
```

Например, функция print("Pancakes", 15) использует строку и целое число в качестве аргументов, а это соответствует прототипу #1.

Когда вы прибегаете к помощи перегруженных функций, убедитесь в том, что при вызове функции используются допустимые типы аргументов. Например, рассмотрим следующие операторы:

```
unsigned int year = 3210;
print(year, 6); // неопределенный вызов
```

Какому прототипу соответствует в рассматриваемом случае вызов функции print()? Ни одному из них! Отсутствие подходящего прототипа автоматически не исключает какой-либо из функций, так как C++ предпримет попытку выполнить стандартное приведение типов,

чтобы достичь соответствия. Если бы, скажем, единственным прототипом функции print() был прототип #2, вызов функции print(year, 6) повлек бы за собой преобразование значения year к типу double. Однако в программном коде, приведенном выше, имеется три прототипа, которые используют число в качестве первого аргумента, при этом возникают три варианта преобразования аргумента year. Очнувшись в такой неоднозначной ситуации, C++ отвергает подобный вызов функции, истолковывая его как ошибку.

Некоторые сигнатуры, различаясь между собой, не могут сосуществовать. Например, рассмотрим два таких прототипа:

```
double cube(double x);
double cube(double & x);
```

Можно предположить, что это именно тот случай, когда вы можете воспользоваться перегрузкой функций, так как создается впечатление, что сигнатуры обеих функций различны. Однако подойдем к этой ситуации с точки зрения компилятора. Предположим, мы имеем такой программный код:

```
cout << cube(x);
```

Аргумент x соответствует как прототипу double x, так и прототипу double &x. Следовательно, компилятор не может определить, какую функцию использовать. В связи с этим, чтобы избежать такой путаницы, при проверке сигнатуры функции компилятор считает, что ссылка на тип и сам тип имеют одну и ту же сигнатуру.

В ходе сопоставления функций учитываются различия между переменными со статусом const и со статусом, отличным от const. Рассмотрим следующие прототипы:

```
void dribble(char * bits); // перегружена
void dribble (const char *cbits); // перегружена
void dabble(char * bits); // не перегружена
void drivel(const char * bits); // не перегружена
```

Вот чему соответствуют вызовы различных функций:

```
const char p1[20] = "How's the weather?";
char p2[20] = "How's business?";
dribble(p1); // dribble(const char *);
dribble(p2); // dribble(char *);
dabble(p1); // нет соответствия
dabble(p2); // dabble(char *);
drivelp1); // drivel(const char *);
drivelp2); // drivel(const char *);
```

У функции dribble() имеется два прототипа: один — для указателей со статусом const и один — для обычных указателей, а компилятор выбирает тот или другой в зависимости от того, имеет ли фактический аргумент статус const или нет. Функция dabble() сопоставляется

только вызовы с аргументом, имеющим статус, отличный от `const`, однако функция `drivel()` сопоставляет вызовы как с аргументами `const`, так и с аргументами, не являющимися `const`. Причина такого различия в поведении функций `drivel()` и `dabble()` заключается в том, что правильной будет операция присвоения статуса, отличного от `const`, переменной `const`, но не наоборот.

Все время имейте в виду, что именно сигнатура, а не тип функции делает возможным перегрузку функции. Например, два следующих объявления не являются совместимыми:

```
long gronk(int n, float m); // одинаковые
 // сигнатурь,
double gronk(int n, float m); // поэтому
 // не допускаются
```

В связи с этим C++ не позволяет перегрузить функцию `gronk()` таким манером. У вас могут быть различные типы возвращаемых значений, но только в тех случаях, когда сигнатурь различны:

```
long gronk(int n, float m); // различные
 // сигнатурь,
double gronk(float n, float m); // поэтому
 // допустимы
```

После того как далее в этой главе мы обсудим шаблоны, рассмотрим, как функции приводятся в соответствие.

## Пример перегрузки

Выше мы уже создали функцию `left()`, которая возвращает указатель на первые `n` символов в строке. Теперь построим еще одну функцию `left()`, но на этот раз эта функция возвращает первые `n` цифр в записи целого числа. Вы можете воспользоваться ею, например, для анализа первых трех цифр почтового кода США (zip), который хранится в виде целого числа, — весьма полезное занятие, если вы намерены сортировать почту в городских районах.

Функцию, выполняющую обработку целых чисел, несколько труднее запрограммировать, чем их строковые версии, поскольку во втором случае каждый символ содержится в собственном элементе массива, это преимущество не проявляется, когда речь идет о числах. Один из походов к решению этой задачи состоит в подсчете числа цифр в представлении числа. Деление числа на 10 позволяет определить одну цифру, таким образом, вы можете воспользоваться делением, чтобы подсчитать количество цифр в числе. Точнее, можно выполнить эту процедуру в цикле, подобном приведенному ниже:

```
unsigned digits = 1;
while (n /= 10)
 digits++;
```

В данном цикле подсчитывается, сколько раз вы можете вычесть цифру из `n`, пока ничего не останется. Напомним, что запись `n /= 10` короче, чем `n = n / 10`. Если, например, `n` равно 8, то в ходе выполнения проверки циклу `n` присваивается значение  $8 / 10$  или 0, поскольку деление целочисленное. Выполнение цикла при этом прекращается, и значение переменной `digits` остается равным 1. Но если значение `n` равно 238, то на первом этапе проверки условия цикла устанавливается значение `n` равным  $238 / 10$  или 23. Это значение не равно нулю, поэтому в ходе выполнения цикла увеличивается значение `digits` до 2. На следующем этапе цикла значение `n` устанавливается равным  $23 / 10$  или 2. И опять эта величина не равна нулю, так что значение `digits` возрастает до 3. На следующем этапе цикла значение `n` устанавливается равным  $2 / 10$  или 0, и выполнение цикла прекращается, а значение переменной `digits` остается равным 3, т.е. правильному значению.

Теперь предположим, что вам известно, что исходное число содержит пять цифр, и вы хотите возвратить первые три цифры. При каждом делении числа на 10 в записи числа удаляется одна цифра справа. Вы можете вычислить это значение, поделив число на 10, а потом поделив ответ на 10. Чтобы подсчитать, какое количество цифр нужно удалить, следует просто вычислить число цифр, которые требуется отобразить, из общего количества цифр в представлении исходного числа. Например, чтобы отобразить четыре цифры числа, представленного девятью цифрами, удалите последние пять цифр. Этот подход можно представить в виде следующего программного кода:

```
ct = digits - ct;
while (ct--)
 num /= 10;
return num;
```

Программа, представленная в листинге 8.8, включает этот код в новую функцию `left()`. Эта функция содержит и другие коды, предназначенные для специальных случаев, таких как запрос на ввод нулевого числа цифр или запрос на ввод большего количества цифр, чем содержится в исходном числе. Поскольку сигнатурь новой функции `left()` отличается от сигнатурь старой функции `left()`, мы получаем возможность использовать обе функции в одной и той же программе, и мы воспользуемся ею.

### Listing 8.8 Программа leftover.cpp.

```
// leftover.cpp — перегрузка функции left()
#include <iostream>
using namespace std;
unsigned long left(unsigned long num,
 unsigned ct);
char * left(const char * str, int n = 1);
```

```

int main()
{
 char * trip = "Hawaii!!"; // тестовое
 // значение
 unsigned long n = 12345678; // тестовое
 // значение

 int i;
 char * temp;
 for (i = 1; i < 10; i++)
 {
 cout << left(n, i) << "\n";
 temp = left(trip, i);
 cout << temp << "\n";
 delete [] temp; // указывает на
 // временную память
 }
 return 0;
}

// Эта функция возвращает первые ct цифр
// числа num.
unsigned long left(unsigned long num, unsigned ct)
{
 unsigned digits = 1;
 unsigned long n = num;
 if (ct == 0 || num == 0)
 return 0; // возвращает 0, если
 // нет цифр
 while (n /= 10)
 digits++;
 if (digits > ct)
 {
 ct = digits - ct;
 while (ct--)
 num /= 10;
 return num; // возвращает ct цифр слева
 }
 else // если ct >= число цифр
 return num; // возвращает все число
}

// Эта функция возвращает указатель на новую
// строку, состоящую из n первых символов
// строки str.
char * left(const char * str, int n)
{
 if (n < 0)
 n = 0;
 char * p = new char[n+1];
 int i;
 for (i = 0; i < n && str[i]; i++)
 p[i] = str[i]; // копировать
 // символы
 while (i <= n)
 p[i++] = '\0'; // заполнить
 // оставшуюся часть строки символами '\0'
 return p;
}

```

Результаты выполнения программы:

```

1
H
12
Ha
123
Haw

```

```

1234
Hawa
12345
Hawai
123456
Hawaii
1234567
Hawaii!
12345678
Hawaii!!
12345678
Hawaii!!

```

## Когда целесообразно использовать перегрузку функции

Перегрузка функций вам может показаться исключительно удобным средством, тем не менее, не злоупотребляйте им. Следует зарезервировать перегрузку функций для функций, которые выполняют почти одни и те же действия, но с различными типами данных. Наряду с этим вы, вероятно, захотите проверить, сможете ли добиться тех же целей с помощью использования аргументов, заданных по умолчанию. Например, если вы захотите заменить одну функцию `left()`, ориентированную на обработку строк, на две перегруженные функции, используйте следующее выражение:

```

char * left(const char * str, unsigned n); // два аргумента
char * left(const char * str); // один аргумент

```

Тем не менее, использование одной функции с аргументами, заданными по умолчанию, проще. Прежде всего, нужно создавать только одну функцию, а не две, к тому же программа запрашивает пространство под одну функцию, а не под две. Если вы захотите внести в функцию изменения, вам придется изменять только одну функцию. Тем не менее, если вам понадобятся различные типы аргументов, то аргументы, заданные по умолчанию, здесь недоступны, и вот тут-то вам следует воспользоваться перегрузкой функций.

## Шаблоны функций

Современные компиляторы языка C++ реализуют одно из его новейших средств — *шаблоны функций*. Шаблоны функций представляют собой обобщенное описание функций. Другими словами, они определяют функцию, используя термины обобщенных типов, вместо которых могут быть подставлены конкретные типы, такие как, например, `int` или `double`. Передавая тип в качестве параметра, вы можете заставить компилятор генерировать функцию этого конкретного типа. Поскольку шаблоны позволяют программировать на основе обобщенных типов без использования специализации, этот процесс

иногда называют *обобщенным программированием*. Поскольку типы представлены параметрами, шаблоны функций иногда называют *параметризованными типами*. Давайте посмотрим, чем полезно это средство и как оно работает.

Мы уже определили функцию, которая производила обмен значений двух переменных типа `int`. Предположим, что вместо этого можно произвести обмен значений двух переменных типа `double`. Один из подходов к решению этой задачи состоит в сохранении исходного программного кода, однако при этом вам придется заменить каждый тип `int` на `double`. Если нужно произвести обмен значениями двух переменных типа `char`, вы снова можете воспользоваться этим методом. Тем не менее, чтобы выполнить такие незначительные изменения, вам придется затратить время, которого всегда не хватает, при этом отнюдь не исключена вероятность ошибки. Если вы будете вносить эти изменения вручную, то можете пропустить ту или иную величину типа `int`. Если же производится глобальный поиск и замену, то в итоге вы можете получить такое преобразование, когда конструкция

```
int integer;
```

превращается в

```
double doubleeger;
```

Средство шаблона функции в языке C++ автоматизирует этот процесс, обеспечивая высокую надежность и экономию времени.

Шаблоны функций предоставляют возможность давать определения функций на основе некоторого произвольного типа данных. Например, вы можете построить шаблон обмена, подобный приведенному ниже:

```
template <class Any>
void Swap(Any &a, Any &b)
{
 Any temp;
 temp = a;
 a = b;
 b = temp;
}
```

Первая строка показывает, что устанавливается шаблон и что вы присваиваете произвольному типу данных имя `Any`. Обязательным условием является употребление ключевых слов `template` и `class` (или `typename`), равно как и угловых скобок. Имя типа вы выбираете по собственному усмотрению, при этом необходимо соблюдать все правила использования имен, действующие в C++. Многие программисты применяют простые имена, например, такие как `T`. Остальная часть программного кода описывает алгоритм обмена значениями типа `Any`. Шаблон не создает никаких функций. Вместо этого он предостав-

ляет компилятору указания, касающиеся того, как определить функцию. Если вы хотите, чтобы функция осуществляла обмен значениями типа `int`, то компилятор создаст функцию, соответствующую образцу шаблона, подставляя `int` вместо `Any`. Аналогично, если вам нужна функция, которая производит обмен значениями типа `double`, компилятор будет руководствоваться требованиями шаблона, подставляя тип `double` вместо `Any`.

Недавно в C++ появилось новое ключевое слово, `typename`, которое может быть использовано вместо ключевого слова `class` в рассматриваемом контексте. Это значит, что вы можете записать определение шаблона в такой форме:

```
template <typename Any>
void Swap(Any &a, Any &b)
{
 Any temp;
 temp = a;
 a = b;
 b = temp;
}
```

С использованием ключевого слова `typename` становится более очевидным, что параметр `Any` представляет тип; в то же время уже созданы большие библиотеки программных кодов, которые были разработаны с применением старого ключевого слова `class`. Стандартный C++ рассматривает в данном контексте оба эти ключевых слова как идентичные.



#### СОВЕТ

Используйте шаблоны в тех случаях, когда вам нужны функции, применяющие один и тот же алгоритм к различным типам данных. Если перед вами не стоит проблема обратной совместимости и если вас не пугает перспектива вывода на печать более длинного слова, используйте при объявлении типов параметров ключевое слово `typename` вместо ключевого слова `class`.

Чтобы сообщить компилятору о том, что вам нужна некоторая конкретная форма функции обмена, воспользуйтесь в программе функцией с именем `Swap()`. Компилятор сначала проверит типы аргументов, которые вы используете, а затем создаст соответствующую функцию. В листинге 8.9 показано, как все это делается. Формат программы выбран по образцу для обычных функций, в котором шаблон прототипа функции располагается в верхней части файла, а шаблонное определение функции следует за `main()`.



#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Более ранние версии C++ могут не поддерживать шаблоны. Новые версии рассматривают ключевое слово `typename` как альтернативу ключевому слову `class`. Версия 2.71 g++ требует, чтобы и прототип шаблона, и определение шаблона появились в программе раньше, чем шаблон будет использован.

### Листинг 8.9 Программа funtemp.cpp.

```
// funtemp.cpp - использование шаблона функции
#include <iostream>
using namespace std;
// прототип шаблона функции
template <class Any> // или typename Any
void Swap(Any &a, Any &b);

int main()
{
 int i = 10;
 int j = 20;
 cout << "i, j = " << i << ", "
 << j << ".\n";
 cout << "Using compiler-generated int
 \swapper\n";
 Swap(i,j); // генерирует void Swap(int &,
 // int &)
 cout << "Now i, j = " << i << ", "
 << j << ".\n";
 double x = 24.5;
 double y = 81.7;
 cout << "x, y = " << x << ", "
 << y << ".\n";
 cout << "Using compiler-generated double,
 \swapper:\n";
 Swap(x,y); // генерирует void
 // Swap(double &, double &)
 cout << "Now x, y = " << x << ", "
 << y << ".\n";
 return 0;
}

// определение шаблона функции
template <class Any> // или typename Any
void Swap(Any &a, Any &b)
{
 Any temp; // переменная temp типа Any
 temp = a;
 a = b;
 b = temp;
}
```

Первая функция **Swap()** имеет два аргумента **int**, следовательно, компилятор генерирует версию **int** функции обмена. Другими словами, она заменяет каждый тип **Any** на **int**, в результате чего появляется определение, которое принимает следующий вид:

```
void Swap(int &a, int &b)
{
 int temp;
 temp = a;
 a = b;
 b = temp;
}
```

Вы не видите результата действия этого программного кода, однако компилятор определяет этот результат. Вторая функция **Swap()** имеет два аргумента типа **double**, следовательно, компилятор генерирует версию **double** этой функции. Иначе говоря, он заменяет **Any** на **double**, благодаря чему появляется такой программный код:

```
void Swap(double &a, double &b)
{
 double temp;
 temp = a;
 a = b;
 b = temp;
}
```

Ниже представлены результаты выполнения программы; вы можете проследить, как выполнялся этот процесс:

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
x, y = 24.5, 81.7.
Using compiler-generated double swapper:
Now x, y = 81.7, 24.5.
```

Обратите внимание на то обстоятельство, что шаблоны функций не делают ваши выполняемые программы короче. В программе, представленной в листинге 8.9, вы все еще довольствуетесь двумя отдельными определениями функций, как если бы вы определили каждую функцию вручную. Однако окончательный программный код не содержит никаких шаблонов; он всего лишь содержит фактические функции, созданные для вашей программы. Преимущество использования шаблонов заключается в том, что они генерируют наборы определений более простых и более надежных функций.

### Перегруженные шаблоны

Вы прибегаете к помощи шаблонов, когда требуются функции, которые применяют один и тот же алгоритм к различным типам данных. Примером этого может служить программа, представленная в листинге 8.8. Однако вполне может случиться, что не для всех типов данных можно использовать один и тот же алгоритм. Чтобы не исключать подобного рода возможность, вы можете воспользоваться перегрузкой определений шаблонов, точно также, как вы перегружали определения обычных функций. Как и в случае обычной перегрузки, для перегруженных шаблонов требуются четко очерченные сигнатуры. Например, в листинг 8.10 добавлен новый шаблон обмена, обеспечивающий обмен элементами между двумя массивами. Исходный шаблон обладает сигнатурой (**Any &**, **Any &**), в то время как новый шаблон имеет сигнатуру (**Any []**, **Any []**, **int**). Обратите внимание на тот факт, что последним аргументом является специальный тип (**int**), а не обобщенный. Не все аргументы шаблонов должны быть типами параметров шаблонов.

Когда в файле **twotemps.cpp** компилятор встречает первый пример использования функции **Swap()**, он обнаруживает, что в ней имеется два аргумента типа **int**, и сопоставляет ее с исходным шаблоном. Однако во втором случае использования этой функции в качестве аргументов выступают два массива элементов типа **int** и значение типа **int**, и это соответствует новому шаблону.

### Листинг 8.10 Программа twotemps.cpp.

```
// twotemps.cpp - использование перегруженных
// шаблонов функций
#include <iostream>
using namespace std;
template <class Any> // исходный шаблон
void Swap(Any &a, Any &b);
template <class Any> // новый шаблон
void Swap(Any *a, Any *b, int n);
void Show(int a[]);
const int Lim = 8;
int main()
{
 int i = 10, j = 20;
 cout << "i, j = " << i << ", "
 << j << ".\n";
 cout << "Using compiler-generated int "
 << "swapper:\n";
 Swap(i,j); // соответствует исходному
 // шаблону
 cout << "Now i, j = " << i << ", "
 << j << ".\n";
 int d1[Lim] = { 0,7,0,4,1,7,7,6} ;
 int d2[Lim] = { 0,6,2,0,1,9,6,9} ;
 cout << "Original arrays:\n";
 Show(d1);
 Show(d2);
 Swap(d1,d2,Lim); // соответствует новому
 // шаблону
 cout << "Swapped arrays:\n";
 Show(d1);
 Show(d2);

 return 0;
}

template <class Any>
void Swap(Any &a, Any &b)
{
 Any temp;
 temp = a;
 a = b;
 b = temp;
}

template <class Any>
void Swap(Any a[], Any b[], int n)
{
 Any temp;
 for (int i = 0; i < n; i++)
 {
 temp = a[i];
 a[i] = b[i];
 b[i] = temp;
 }
}
void Show(int a[])
{
 cout << a[0] << a[1] << "/";
 cout << a[2] << a[3] << "/";
 for (int i = 4; i < Lim; i++)
 cout << a[i];
 cout << "\n";
}
```



### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Более ранние версии C++ могут не поддерживать шаблоны. Новые версии рассматривают ключевое слово **typename** как альтернативу ключевому слову **class**. Ранние версии C++ более привередливы в отношении соответствия типов, и, чтобы выражение **const int Lim** соответствовало требованиям шаблона, предъявляемых к обычному типу **int**, они требуют наличия в программе такого программного кода:

```
Swap(d1,d2, int (Lim)); // приведение типа
 // Lim к int, не являющемуся const
```

Версия g++ 2.7.1 требует, чтобы все определения шаблона были размещены в программе перед **main()**.

Результаты выполнения программы:

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Original arrays:
07/04/1776
07/20/1969
Swapped arrays:
07/20/1969
07/04/1776
```

### Явная специализация

Предположим, что вы даете определение структуры, подобное следующему:

```
struct job
{
 char name[40];
 double salary;
 int floor;
};
```

Предположим также, что вы хотите выполнить обмен данными для этих структур. Исходный шаблон использует следующий код для осуществления такого обмена:

```
temp = a;
a = b;
b = temp;
```

Поскольку C++ позволяет присваивать одну структуру другой, этот код работает безупречно даже в том случае, когда тип **Any** является структурой **job**. Предположим, что мы хотим совершить только обмен данными между элементами **salary** и **floor**. Для этого требуется другой программный код, но аргументы функции **Swap()** будут такими же, как и в первом случае (ссылки на две структуры **job**), так что вы не можете воспользоваться перегрузкой шаблона, чтобы составить альтернативный программный код.

Однако вы можете снабдить программу специализированным определением функции, именуемым **явной специализацией** и содержащим требуемый программный код. Если ваш компилятор находит специализированное определение, которое соответствует вызову функции, он

использует это определение, не просматривая использование шаблонов.

Механизм специализации претерпевал изменения по мере эволюции языка. Мы рассмотрим исходную форму, поддерживаемую старыми компиляторами, промежуточную форму, а затем форму, действующую на текущий момент, которая регламентируется стандартом C++.

### Метод первой генерации

Поначалу объявление обычной функции, которое в точности соответствует обращению к функции, приводило к переопределению шаблона. Например, рассмотрим следующие фрагменты программного кода:

```
template <class Any>
void Swap(Any &a, Any &b); // прототип
 // шаблона
void Swap(int &n, int &m); // обычный
 // прототип

int main()
{
 double u, v;
 ...
 Swap(u,v); // использовать шаблон
 int a, b;
 ...
 Swap(a,b); // использовать void
 // Swap(int &, int &)
```

Когда компилятор достигает вызова функции `Swap(a,b)`, он становится перед выбором создания определения функции путем использования для этой цели шаблона или функции `Swap(int &, int &)`, не имеющей шаблона. Исходное средство построения шаблонов обращалось в подобных ситуациях к компилятору, чтобы тот использовал версию без шаблонов, трактуя ее как специализацию шаблона.

### Вторая генерация

До некоторого времени C++ (неофициальная начальная версия стандарта) использовался для выполнения различных видов обработки данных. В случае с представленным выше программным кодом компилятор использовал шаблон и игнорировал обычный прототип и определения функции. Однако такое изменение не устраняет необходимости в выполнении явной специализации. Поэтому в C++ разработан новый синтаксис объявления и описания явных специализаций. Идея заключается в том, чтобы за именем функции указывались угловые скобки, в которых содержался бы специализированный тип. Например, специализированный под `int` прототип функции `Swap()` принимает вид:

```
void Swap<int>(int &a, int &b);
 // специализация
```

Этот прототип должен появиться в программе перед первым обращением к функции с соответствующими аргументами:

```
template <class Any>
void Swap(Any &a, Any &b); // прототип
 // шаблона
void Swap<int>(int &n, int &m); // прототип специализации
int main()
{
 double u, v;
 ...
 Swap(u,v); // использовать шаблон
 int a, b;
 ...
 Swap(a,b); // используется void
 Swap<int>(int &, int &)
}
void Swap<int>(int &n, int &m) // определение специализации
{...}
```

Обратите внимание на тот факт, что `<int>` появляется в прототипе и в определении функции. Он может, но не обязан появляться при вызове функции:

```
Swap<int>(a, b); // возможная альтернатива
```

### Третья генерация

Стандарт C++ предусматривает еще один подход:

- Функция без шаблона перекрывает шаблон, но не рассматривается как специализация
- Прототипу и определению явной специализации должна предшествовать конструкция `template <>`:

Специализация перекрывает обычный шаблон, а функция без шаблона перекрывает и шаблоны, и специализацию.

Следовательно, версии без шаблонов выбираются на основе явных специализаций и версий с шаблонами, а явные специализации выбираются на основе версий, генерируемых на базе шаблонов.

```
template <class Any>
void Swap(Any &a, Any &b); // прототип
 // протокола
template <> void Swap<int>(int &n,
 int &m); // прототип специализации
int main()
{
 double u, v;
 ...
 Swap(u,v); // использование шаблона
 int a, b;
 ...
 Swap(a,b); // использование
 // void Swap<int>(int &, int &)
}
template <> void Swap<int>(int &n,
 int &m) // определение специализации
{...}
```

Использование конструкции `<int>` в конструкции `Swap<int>` вовсе не обязательно, поскольку типы аргументов функции показывают, что она является специализацией типа `int`. Следовательно, такой прототип может быть записан следующим образом:

```
template <> void Swap(int & n, int & m);
// более простая форма
```

Заголовок функции с шаблоном также может быть упрощен благодаря исключению из него части `<int>`.

Немного погодя мы выясним, что послужило причиной подобного усложнения синтаксиса, но сначала рассмотрим пример.

### Листинг 8.11 Программа `twoswap.cpp`.

```
// twoswap.cpp - специализация перекрывает шаблон
#include <iostream>
using namespace std;
template <class Any>
void Swap(Any &a, Any &b);
struct job
{
 char name[40];
 double salary;
 int floor;
};
//void Swap(job &j1, job &j2); // первая генерация
//void Swap<job>(job &j1, job &j2); // вторая генерация
template <> void Swap(job &j1, job &j2); // третья генерация
void Show(job &j);
int main()
{
 cout.precision(2);
 cout.setf(ios::fixed, ios::floatfield);
 int i = 10, j = 20;
 cout << "i, j = " << i << ", " << j << ".\n";
 cout << "Using compiler-generated int swapper:\n";
 Swap(i,j); // генерируется void Swap(int &, int &)
 cout << "Теперь i, j = " << i << ", " << j << ".\n";
 job sue = { "Susan Yaffee", 63000.60, 7} ;
 job sidney = { "Sidney Taffee", 66060.72, 9} ;
 cout << "Before job swapping:\n";
 Show(sue);
 Show(sidney);
 Swap(sue, sidney); // используется void Swap(job &, job &)
 cout << "After job swapping:\n";
 Show(sue);
 Show(sidney);
 return 0;
}
template <class Any>
void Swap(Any &a, Any &b) // обобщенная версия
{
 Any temp;
 temp = a;
 a = b;
 b = temp
}
```

### Пример

Программа, представленная в листинге 8.11, показывает, как выполняется явная специализация. Она разработана с соблюдением требований Стандарта C++, однако вы можете удалять и вставлять знаки комментариев, что позволяет выбрать какую-либо из возможных версий.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Одни компиляторы допускают форму `template <> void Swap()`, другие допускают форму `void Swap<job>()`, третий — форму `void Swap()`.

```

// производит только обмен полей salary (жалование) и floor (этаж) структуры job (задание)
//void Swap(job &j1, job &j2) // первая генерация
//void Swap<job>(job &j1, job &j2) // вторая генерация
template <> void Swap(job &j1, job &j2) // третья генерация
{
 double t1;
 int t2;
 t1 = j1.salary;
 j1.salary = j2.salary;
 j2.salary = t1;
 t2 = j1.floor;
 j1.floor = j2.floor;
 j2.floor = t2;
}
void Show(job &j)
{
 cout << j.name << ": $" << j.salary
 << "on floor" << j.floor << "\n";
}

```

Результаты выполнения программы:

```

i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Before job swapping:
Susan Yaffee: $63000.60 on floor 7
Sidney Taffee: $66060.72 on floor 9
After job swapping:
Susan Yaffee: $66060.72 on floor 9
Sidney Taffee: $63000.60 on floor 7

```

## Образование шаблонов и специализация

Имейте в виду, что включение шаблона функции в программный код само по себе не приводит к генерации определения функции. Это просто план построения определения функции. Когда компилятор использует шаблон при создании определения функции для конкретного типа данных, результат соответствующих действий компилятора называется *образованием шаблона*. Например, в программе, представленной в листинге 8.11, вызов функции `Swap(i,j)` заставляет компилятор создать `Swap()`, используя тип данных `int`. Шаблон не является определением функции, определением функции является некоторый конкретный экземпляр шаблона, использующий `int`. Такой способ создания экземпляров шаблонов получил название *неявного образования шаблонов*, поскольку компилятор приходит к выводу о необходимости построения определения, обнаружив, что в программе используется функция `Swap()` с параметрами типа `int`.

Первоначально неявное образование было единственным способом, посредством которого компилятор генерировал определения функций по шаблонам, однако сейчас C++ позволяет выполнять явное образование шаблонов. Это означает, что вы можете дать компилятору

прямую команду, например, `Swap<int>()`, создать конкретный экземпляр шаблона. Синтаксис позволяет вам объявить все, что необходимо, используя оператор `<>` для задания типа и предварив объявление ключевым словом `template`:

```
template Swap<int>(int, int); // явное
 // образование шаблонов
```

Компилятор, в котором реализовано это свойство, обнаружив такое объявление, использует шаблон функции `Swap()` в целях генерации экземпляра шаблона, применяющего тип `int`. Другими словами, такое объявление означает "Используйте шаблон функции `Swap()` для генерации определения функции для типа `int`".

Сопоставим явное образование шаблонов с явной специализацией, которая использует одно из следующих эквивалентных определений:

```
template <> Swap<int>(int, int); // явная
 // специализация
template <> Swap(int, int); // явная
 // специализация
```

Различие состоит в том, что эти объявления означают следующее: "Не смейте пользоваться шаблоном функции `Swap()` для генерации определения функции. Вместо этого воспользуйтесь отдельным, специализированным определением функции, явно сформулированным для типа `int`". Эти прототипы должны быть ассоциированы с их собственными определениями функции.



### ПРЕДОСТЕРЖЕНИЕ

Попытка одновременного использования в одном программном модуле как метода явного образования, так и явной специализации для одного и того же типа (типов) приводит к ошибке.

Неявное и явное образование шаблонов, а также явная специализация получили общее название — *специализация*. Общим для них является то, что они представляют определение функции, в основу которого положены конкретные типы, а не определение функции, являющееся обобщенным описанием.

Введение явного образования шаблонов привело к появлению нового синтаксиса использования в объявлениях префиксов **template** и **template <>**, что дает возможность провести различие между явным образованием шаблонов и явной специализацией. Чаще всего бывает так, что более широкие возможности обуславливаются увеличением количества синтаксических правил.

## Выбор функций

Что касается перегрузки функций, шаблонов функций и перегрузки шаблонов функций, то требуется четко определенная стратегия. Такая стратегия поможет принимать решение относительно того, какое определение функции следует использовать для вызова функции, особенно в тех случаях, когда рассматривается несколько функциональных аргументов. Подобный процесс называется *определением перегрузки*. Детальная проработка стратегии во всей ее полноте требует, по меньшей мере, отдельной небольшой главы, поэтому здесь мы рассмотрим в общих чертах только то, как этот процесс реализован:

- Фаза 1. Составьте список функций-кандидатов. Таковыми являются функции и шаблоны функций с таким же именем, как у вызванной функции.
- Фаза 2. Беря за основу список функций-кандидатов, составим список подходящих функций. Таковыми являются функции с правильно установленным числом аргументов, для которых существует неявная последовательность преобразований типов, в число которых входит случай точного соответствия каждого типа фактического аргумента типу формального аргумента. Например, при вызове функции с аргументом типа **float** эта величина может быть приведена к типу **double**, что позволит достигнуть соответствия типу **double** формального параметра, а шаблон может породить образование типа **float**.
- Фаза 3. Проверим, имеется ли среди них наиболее подходящая функция. Если имеется, то воспользуйтесь этой функцией. В противном случае подобный вызов функции является ошибочным.

Рассмотрим случай функции с единственным аргументом, например, следующее обращение:

```
may('B'); // фактический аргумент имеет
 // тип char
```

Прежде всего компилятор отмечает все "подозрительные" объекты, таковыми являются функции и шаблоны функций, имеющие имя **may()**. Затем он находит среди них те, которые вызываются с одним аргументом. Например, в этом случае проверку пройдут следующие кандидаты:

```
void may(int); // #1
float may(float, float = 3); // #2
void may(char); // #3
char * may(const char *); // #4
template<class T> void may(const T &); // #5
template<class T> void may(T *); // #6
```

Обратите внимание на тот факт, что при этом учитываются только сигнатуры, а не типы возвращаемых значений. Оба из этих кандидатов (#4 и #6) не подходят, поскольку интегрированный тип не может быть преобразован неявно (т.е. без явного приведения типов) к типу указателя. Теперь в списке остается четыре функции, каждая из которых могла бы быть использована так, если бы она была единственной объявлена функцией.

Далее компилятор должен определить, какая из функций-кандидатов в наибольшей степени соответствует критерию отбора. Он анализирует преобразования, необходимые для того, чтобы аргумент обращения к функции соответствовал аргументу наиболее подходящего кандидата. В общем случае градация от лучшего к наихудшему такова:

1. Точное совпадение
2. Приведение с повышением типа (например, автоматическое приведение типов **char** и **short** к **int** и типа **float** к **double**)
3. Приведение типов посредством стандартных преобразований (например, приведение **int** к **char** или **long** к **double**)
4. Приведения типов, заданные пользователем, подобные тем, что определяются при объявлении классов.

Например, функция #1 подходит лучше, чем функция #2, поскольку преобразование **char**-к-**int** является повышением типа (см. главу 3), в то время как **char**-к-**float** — это стандартное преобразование. Функции #3 и #4 подходят лучше, чем любая из функций #1 или #2, так как они точно соответствуют друг другу. В этом случае возникает два вопроса. Что такое точное соответствие и что произойдет, если таких соответствий будет два?

## Точное соответствие и наилучшее соответствие

Для достижения точного соответствия C++ допускает некоторые "тривиальные преобразования данных". В табл. 8.1 приводится список таких преобразований, при этом слово **Type** обозначает некоторый произвольный тип. Например, формальный аргумент типа **int** представляет собой точное соответствие формальному параметру **int &**. Обратите внимание на то, что **Type** может быть чем-то подобным **char &**, так что эти правила предусматривают приведение типа **char &** к **const char &**. Запись **Type (список-аргументов)** означает, что имя функции как фактический аргумент соответствует указателю функции, выступающему в качестве формального аргумента до тех пор, пока оба имеют один и тот же возвращаемый тип и один и тот же список аргументов. (Вспомните указатели функций, обсуждавшиеся в главе 7, и то, как можно передать имя функции в качестве аргумента функции, ожидающей указателя на функцию.) Далее в этой главе мы рассмотрим ключевое слово **volatile**.

**Таблица 8.1 Тривиальные преобразования, допустимые для случая точного соответствия.**

| <b>Фактический аргумент преобразуется</b> | <b>в формальный аргумент</b> |
|-------------------------------------------|------------------------------|
| Type                                      | Type &                       |
| Type &                                    | Type                         |
| Type []*                                  | Type                         |
| Type (список аргументов)                  | Type (*) (список аргументов) |
| Type                                      | const Type                   |
| Type                                      | volatile Type                |
| Type *                                    | const Type *                 |
| Type *                                    | volatile Type *              |

Помимо всего прочего, предположим, у вас есть следующий программный код функции:

```
struct blot { int a; char b[10]; } ;
blot ink = { 25, "spots" } ;
...
recycle(ink);
```

Таким образом, все приводимые ниже прототипы будут точными соответствиями:

```
void recycle(blot); // #1 blot-к-blot
void recycle(const blot); // #2
 // blot-к-(const blot)
void recycle(blot &); // #3
 // blot-к-(blot &)
void recycle(const blot &); // #4
 // blot-к-(const blot &)
```

Можно предположить, что в результате наличия нескольких соответствующих прототипов компилятор не

сможет завершить процесс определения перегрузки. В этом случае не существует наилучшей жизнеспособной функции, а компилятор генерирует сообщение об ошибке; скорее всего, в нем будут использованы такие слова, как "неоднозначный".

Тем не менее, определение перегрузки может иметь место, даже если две функции представляют собой точное соответствие. Прежде всего, указатели и ссылки на данные, не имеющие статуса **const**, сопоставляются преимущественно с указателями и со ссылочными параметрами, тоже не имеющими статуса **const**. Таким образом, если бы функции #3 и #4 были использованы в примере с **recycle()**, то была бы выбрана функция #3, поскольку переменная **ink** не была объявлена как **const**. Тем не менее, такое неодинаковое отношение к типам со статусом и без статуса **const** проявляется только в отношении данных, на которые имеются ссылки и указатели. Другими словами, если бы использовались функции #1 и #2, то сразу же появилось бы сообщение об ошибке, вызванной неопределенностью.

Вот другой пример того, когда одно точное совпадение лучше другого: когда одна из функций является функцией без шаблона, а другая таковой не является. В этом случае функция без шаблона считается более подходящей, чем функция, имеющая шаблон, в том числе и явные специализации.

Если вы столкнетесь с ситуацией, когда имеется два точных соответствия и обе функции имеют шаблоны, то шаблонная функция с более высокой специализацией (при наличии таковой) будет рассматриваться как более предпочтительная. Это означает, например, что явная специализация получает преимущество при выборе перед функцией, неявно построенной по шаблону:

```
struct blot { int a; char b[10]; } ;
template <class Type>
void recycle (Type t); // шаблон
template <> void recycle<blot> (blot & t);
 // специализация для blot
...
blot ink = { 25, "spots" } ;
...
recycle(ink); // используется специализация
```

Термин "наиболее специализированная" не всегда означает явную специализацию. В принципе, он показывает, что если компилятор принимает решение относительно того, какой тип использовать, то производится меньшее число преобразований. В качестве примера рассмотрим два следующих шаблона:

```
template <class Type>
void recycle (Type t); #1
template <class Type>
void recycle (Type * t); ... #2
```

Предположим, что программа, которая содержит эти шаблоны, включает также и следующий код:

```
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
...
recycle(&ink); // адрес структуры
```

Вызов `recycle(&ink)` соответствует шаблону #1, в котором `Type` интерпретируется как `blot *`. Обращение к функции `recycle(&ink)` соответствует шаблону #2, но на этот раз `Type` интерпретируется как `ink`. Это сочетание передает две неявные конкретизации, `recycle<blot *>(blot *)` и `recycle<blot>(blot *)`, в пул подходящих функций.

Считается, что из этих двух определенных по шаблону функций `recycle<blot *>(blot *)` специализирована в большей степени, поскольку она претерпевает меньшее число преобразований в процессе генерации. Иначе говоря, шаблон #2 уже явно заявил, что аргументом функции был указатель-на-`Type`, так что `Type` мог быть непосредственно отождествлен с `blot`. Однако, шаблон #1 использует `Type` в качестве аргумента функции, так что `Type` должен быть интерпретирован как указатель-на-`blot`. Другими словами, в шаблоне #2 `Type` уже был специализирован как указатель, отсюда происходит выражение "более специализированный".

Правила поиска наиболее специализированного шаблона называются *правилами частичного упорядочивания шаблонов функций*. Как и явные образования, они представляют собой новые языковые средства C++.

Одним словом, в процессе определения перегрузки осуществляется поиск функции, представляющей собой наилучшее соответствие. Если есть такое соответствие, то задача выбора такой функции решена. Если существует сразу несколько функций с другими связями, но только одна из них определена без шаблона, то выбирается именно она. Если имеется более одной функции-кандидата с другими связями и все они являются шаблонными, но уровень специализации одного из шаблонов выше, чем у остальных, то выбор однозначен. Если имеется две или большее число пригодных функций с шаблонами и ни одна из них не превосходит остальные по уровню специализации, то обращение к функции считается неоднозначным и ошибочным. Если соответствующих обращений не удалось обнаружить, то, естественно, такое обращение к функции также считается ошибочным.

### **Функции со многими аргументами**

Сопоставление обращения к функции, имеющей несколько аргументов, с прототипами, тоже имеющими несколько аргументов требует значительных усилий. Компилятор должен проверить соответствия по каждо-

му элементу. Если он сможет найти функцию, которая лучше всех других подходящих функций, то задача выбора будет решена. Следует заметить: чтобы одна функция превосходила другую, она не должна уступать ей по уровню специализации по всем аргументам и превосходить конкурента, по крайней мере, по одному из аргументов.

В задачу настоящей книги не входит иллюстрация процесса сопоставления сложными примерами. Основное правило в этом деле — наличие четко регламентированного результата для любого возможного набора шаблонов и прототипов функций.

## **Раздельная компиляция**

Язык C++, как и C, позволяет и даже поощряет размещение составляющих функций программы в отдельных файлах. Как указано в главе 1, вы можете компилировать файлы по отдельности, а затем связывать их в окончательную выполняемую программу. (Компилятор C++ обычно компилирует программы, а также управляет работой редактора связей.) Если вы вносите изменения всего лишь в один файл, можете повторно компилировать только один этот файл и затем компоновать его с ранее откомпилированными версиями других файлов. Этот механизм существенно облегчает работу с большими программами. Более того, большая часть версий C++ обладает дополнительными средствами, облегчающими управление этим процессом. Например, системы UNIX оснащены программой `make`; она ведет учет всех файлов, от которых зависит программа, и фиксирует время их последней модификации. Если вы пользуетесь услугами программы `make`, она обнаружит, что вы внесли изменения в один или большее число исходных файлов с момента последней компиляции, и предложит вам выполнить соответствующие действия, необходимые для воссоздания программы. Среды Symantec C++, Turbo C++, Borland C++, Watcom C++, Microsoft Visual C++ и Metrowerks CodeWarrior предоставляют в распоряжение пользователя аналогичные средства, доступ к которым осуществляется с помощью меню `Project` каждого рассматриваемого компилятора.

Рассмотрим простой пример. Вместо того чтобы изучать особенности компиляции, которые зависят от реализации, сосредоточимся на более общих аспектах, таких как, например, процесс проектирования.

Предположим, например, что вы решили разбить программу, представленную в листинге 7.11, на части, поместив используемые ею функции в отдельные файлы. Напомним, что эта программа преобразует прямоугольные координаты в полярные, после чего отображает результат на экране. Вы не можете просто "обрезать"

исходный файл по окончании выполнения функции `main()`. Проблема заключается в том, что `main()` и две другие функции используют одно и то же объявление структуры, так что необходимо поместить такие объявления в оба файла. Если просто ввести их с клавиатуры, то резко возрастет вероятность ошибки. Даже если вы скопируете объявление структуры без ошибок, не забудьте внести соответствующие изменения в оба набора объявлений, если вам придется модифицировать их позже. Одним словом, разбивка программы на несколько файлов создает новые проблемы.

Кому нужны лишние проблемы? Разработчикам С и С++ они не нужны, это точно, поэтому они предусмотрели такое средство, как `#include`, которое позволяет выходить из положения в ситуациях подобного рода. Вместо того чтобы помещать объявления структур в каждый файл, вы можете поместить их в заголовочный файл, а затем включить его в каждый файл исходного кода. Таким образом, если вы модифицируете объявление структуры, то можете сделать это только один раз, а именно в заголовочном файле. Итак, можно поместить прототип функции в заголовочный файл. Следовательно, можно разбить исходную программу на три части:

- Заголовочный файл, который содержит объявления структур и прототипов функций, использующих эти структуры
- Файл исходного программного кода, который содержит программный код функций, работающих со структурами.
- Файл исходного программного кода, который содержит коды, осуществляющие вызов этих функций.

Такая стратегия может быть успешно использована для организации программ. Если, например, вы создаете еще одну программу, которая использует те же самые функции, достаточно включить в нее заголовочный файл и добавить файл функции в проект или составить список. Кроме того, такая организация соответствует принципам ООП. Один файл, а именно заголовочный файл, содержит определения, сделанные пользователем. Второй файл содержит программный код функции, манипулирующей типами, определенными пользователем. В совокупности оба файла образуют пакет, которым можно пользоваться в различных программах.

Не помещайте определения функций или объявления переменных в заголовочный файл. И хотя это может привести к формированию конкретных объектов данных, но очень часто служит источником проблем. Например, если у вас в заголовочном файле хранилось определение функции, а вы затем включили этот заголовочный файл в два других файла, которые были частями одной и той же программы, то вы попадете в ситу-

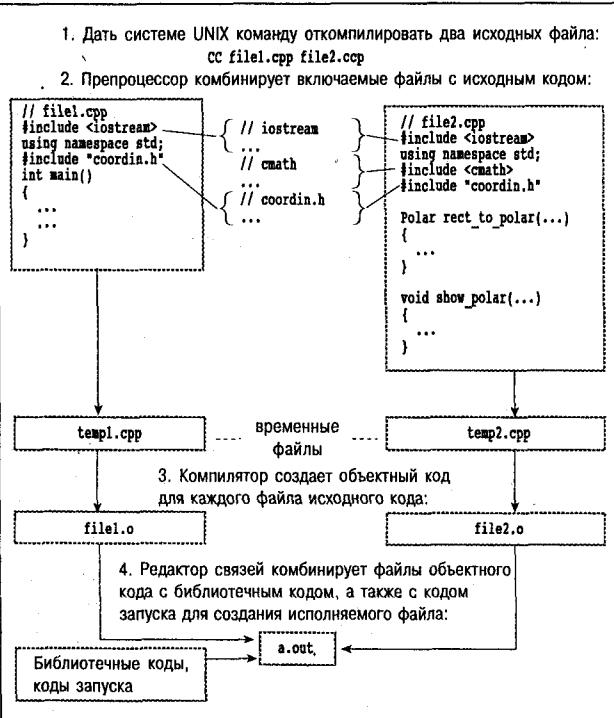
ацию, когда в одной программе имеется два определения одной и той же функции, что является ошибкой. Вот элементы данных, которые обычно содержатся в заголовочных файлах:

- Прототипы функций
- Символьные константы, при определении которых была использована директива `#define` или `const`
- Объявления структур
- Объявления классов
- Объявления шаблонов
- Встроенные функции

Размещение объявлений структур в заголовочном файле весьма удобно, так как они не создают переменных, они просто сообщают компилятору, как создавать переменные типа структура, когда вы объявляете такую переменную в файле исходного программного кода. Аналогично объявления шаблонов не являются программными кодами, которые требуется компилировать. Они представляют собой инструкции, которые предназначены для компилятора и в которых указывается, как генерировать определения функций, соответствующие вызовам функций, встречающимся в исходном программном коде. Данные, объявленные как `const`, и встроенные функции обладают специальными возможностями установления связей (об этом речь пойдет далее), которые позволяют разместить их в заголовочных файлах без особых проблем.

В листингах 8.12-8.14 продемонстрированы результаты разбиения программы, представленной в листинге 7.11, на отдельные части. Обратите внимание на то обстоятельство, что при включении в программу заголовочного файла мы используем `"coordin.h"` вместо `<coordin.h>`. Если имя файла заключено в скобки, то компилятор С++ исследует ту часть системы файлов базисной системы, в которой располагаются стандартные заголовочные файлы. Но если имя файла заключено в двойные кавычки, компилятор сначала просматривает текущий рабочий каталог или каталог исходных программных кодов (или делает другой аналогичный выбор, в зависимости от версии компилятора). Если он там не находит заголовочный файл, он просматривает стандартную область памяти. Включая в программу свои собственные заголовочные файлы, используйте кавычки, а не угловые скобки

На рис. 8.3 представлена схема компиляции программы в системе UNIX. Обратите внимание на тот факт, что вы только даете команду осуществить компиляцию СС (CC — compile command), а все остальные действия выполняются автоматически. Компиляторы Symantec C++, Borland C++, Turbo C++, Metrowerks



**РИСУНОК 8.3** Компиляция многофайловой программы в C++ в операционной системе UNIX.

CodeWarrior, Watcom C++ и Microsoft Visual C++, по существу, выполняют те же самые действия, но, как показано в главе 1, инициализация этого процесса проходит по-разному, с использованием меню, которые позволяют вам создавать проект и ставить в соответствие с ним файлы программных кодов. Обратите внимание на тот факт, что вы включаете в проект только файлы с исходными программными кодами, а не заголовочные файлы. Это объясняется тем, что заголовочными файлами манипулирует директива `#include`. Другими словами, нельзя пользоваться директивой `#include` для включения в программу файлов исходных программных кодов, так как это может повлечь за собой появление нескольких объявлений.



#### ПРЕДОСТЕРЕЖЕНИЕ

В интегрированной среде разработки программ не следует включать заголовочные файлы в список проекта, а также использовать директиву `#include` для включения в программу файлов исходного программного кода в другие файлы исходного программного кода.

**Листинг 8.12** Программа coordin.h.

```
// Шаблоны структур и прототипы функции
// Шаблоны структур
struct polar
{
 double distance; // расстояние от начала
 // координат
```

```
double angle; // направление от
 // начала координат
};

struct rect
{
 double x; // расстояние от начала
 // координат по горизонтали
 double y; // расстояние от начала
 // координат по вертикали
};

// прототипы
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);
```

**Листинг 8.13** Программа file1.cpp.

```
// file1.cpp – пример программы, составленной
// из двух файлов
#include <iostream>
#include "coordin.h" // шаблоны структур,
 // прототипы функций
using namespace std;
int main()
{
 rect rplace;
 polar pplace;
 cout << "Enter the x и y values: ";
 while (cin >> rplace.x >> rplace.y)
 // удачное использование cin
 {
 pplace = rect_to_polar(rplace);
 show_polar(pplace);
 cout << "Next two numbers (q to
 quit): ";
 }
 return 0;
}
```

**Листинг 8.14** Программа file2.cpp.

```
// file2.cpp – содержит функции, вызываемые из
// файла file1.cpp
#include <iostream>
#include <cmath>
#include "coordin.h" // шаблоны структур,
 // прототипы функций
using namespace std;
// преобразование прямоугольных координат в
// полярные
polar rect_to_polar(rect xypos)
{
 polar answer;
 answer.distance = sqrt(xypos.x * xypos.x +
 xypos.y * xypos.y);
 answer.angle = atan2(xypos.y, xypos.x);
 return answer; // возвращение
 // структуры polar
}
// отображение полярных координат,
// преобразование радиан в градусы
void show_polar (polar dapos)
{
 const double Rad_to_deg = 57.29577951;
 cout << "distance = " << dapos.distance;
 cout << ", angle = "
 << dapos.angle * Rad_to_deg;
 cout << " degrec\n";
}
```

Между прочим, несмотря на то что мы рассматривали раздельную компиляцию в понятиях файлов, в языке описания вместо термина "файл" используется термин *единица трансляции* в целях сохранения большей универсальности. Термин "файл" не является единственным при описании процессов, протекающих в компьютере.

## Классы памяти, диапазоны доступа и связывание

Теперь, когда вы ознакомились с многофайловой программой, настало время продолжить рассмотрение классов памяти, начатое в главе 4, поскольку классы памяти влияют на то, как информация будет совместно использоваться различными файлами. С тех пор как вы прочитали главу 4, возможно, прошло достаточно много времени, поэтому напомним кое-что. C++ использует три различные схемы для хранения данных, эти схемы отличаются между собой тем, как долго хранятся данные в памяти.

- Автоматическими являются те переменные, которые были объявлены в определении функции; в их число входят параметры функции. Они создаются в том момент, когда выполнение программы передается функции или блоку, в котором они определены, а отведенная под них память освобождается, когда управление передается за пределы функции или блока.
- Статическими называются переменные, определение которым дано вне определения функции, или переменные, определенные с использованием ключевого слова **static**. Они существует в течение всего времени выполнения программы.
- Динамическая память, выделенная в результате выполнения оператора **new**, существует до тех пор, пока не будет освобождена оператором **delete** или пока не закончится выполнение программы, в зависимости от того, какое из событий наступит раньше.

Заканчивая изучение этих вопросов, рассмотрим такие интересные случаи, когда переменные различных типов находятся в области действия или в диапазоне доступа (могут быть использованы программой), а также связывание, определяющее, какая информация совместно используется различными файлами.

## Диапазон доступа и связывание

*Диапазон доступа* определяет, насколько далеко "видно" конкретное имя в файле (в единице трансляции). Например, переменная, определенная в функции, может быть использована в этой функции, но отнюдь не в какой-либо другой. А переменная, определенная в файле, пред-

шествующем всем определениям функций, может быть использована во всех функциях. При выполнении *связывания* описывается, каким образом имя совместно используется в различных единицах трансляции. Имя с *внешним связыванием* может быть совместно использовано различными файлами, в то время как имя с *внутренним связыванием* может быть использовано функциями в пределах одного файла. Имена автоматических переменных лишены связывания, в связи с чем они не допускают совместного использования в программе.

Переменная в C++ обладает одним из нескольких возможных диапазонов доступа. Переменная, имеющая *локальный диапазон доступа* (его еще называют *областью видимости блока*), известна в пределах того блока, в котором она определена. Напомним, что блок — это последовательность операторов, заключенных в скобки. Тело функции, например, является блоком, однако в теле функции могут быть вложены другие блоки. Переменная с *глобальным диапазоном доступа* (его еще называют *областью видимости файла*) известна во всем файле, начиная с точки, где она была определена. Автоматические переменные обладают локальным диапазоном доступа, статическая переменная может обладать любым диапазоном доступа, в зависимости от того, как она была определена. Имена, используемые в *диапазоне доступа прототипа функции*, известны только в круглых скобках, охватывающих список аргументов. (Вот почему не имеет значения, что они собой представляют, неважно, есть они или их нет.) Элементы, объявленные в классе, обладают диапазоном доступа класса (см. главу 9). Переменные, объявленные в пространстве имен, обладают *диапазоном доступа пространства имен*. (После того как понятие пространства имен было введено в язык, глобальный диапазон доступа стал специальным случаем для диапазона доступа пространства имен.)

Функции C++ могут обладать диапазоном доступа класса или диапазоном доступа пространства имен, в том числе и глобальным диапазоном доступа, но они не могут иметь локального диапазона доступа. (Поскольку функция не может быть определена внутри блока, если предполагается, что у функции должен быть локальный диапазон доступа, она может быть известна только самой себе, и в связи с этим она не может быть вызвана другой функцией. Такая функция не может корректно выполняться.)

Рассмотрим более подробно свойства диапазона доступа и связывания различных классов памяти в C++. Начнем с того, что исследуем ситуацию, имевшую место перед тем, как пространства имен были введены в обиход, и посмотрим, как они повлияли на общую картину.

## Автоматическая память

Параметры функции и переменные, объявленные в теле функции, принадлежат по умолчанию к классу автоматической памяти. Эти переменные характеризуются локальной видимостью, иначе говоря, локальным диапазоном доступа. Другими словами, если вы объявляете переменную с именем `texas` в функции `main()` и объявляете еще одну переменную с тем же именем в функции `oil()`, вы, по существу, создаете две независимые переменные, каждая из которых известна только в функциях, в которых она была объявлена. Действия с переменной `texas` в функции `oil()` никак не отражаются на переменной `texas` в `main()` и наоборот. Каждая переменная размещается в памяти после того, как ее функция начинает выполняться, и каждая из них прекращает свое существование после того, как выполнение ее соответствующей функции закончено.

Если вы определили функцию внутри блока, время существования этой переменной и ее область видимости будут ограничены этим блоком. Предположим, например, что вы определили переменную с именем `teledeli` в начале функции `main()`. Теперь предположим, что вы открываете новый блок в `main()` и определяете в блоке новую переменную с именем `websight`. В таком случае переменная `teledeli` видима как во внешнем, так и во внутреннем блоке, в то время как `websight` видима только во внутреннем блоке и существует с момента ее определения только до тех пор, пока выполнение программы не дойдет до конца блока:

```
int main()
{
 int teledeli = 5;
 {
 int websight = -2;
 cout << websight << endl;
 << teledeli << endl;
 } // срок действия переменной
 // websight истекает
 cout << teledeli << endl;
 ...
}
```

Что, однако, произойдет, если во внутреннем блоке вы присвоите переменной имя `teledeli` вместо `websight`, в результате чего получите две переменные с одним и тем же именем, одна из которых будет находиться во внешнем блоке, а другая — во внутреннем. В этом случае программа интерпретирует имя `teledeli` как переменную, локальную по отношению к блоку, в то время как программа выполняет операторы в этом блоке. Мы в таких случаях говорим, что новое определение скрывает старое определение. Новое определение попадает в диапазон доступа, в то время как старое из нее временно удаляется. Когда выполнение программы передает-

ся за пределы блока, в диапазон доступа возвращается исходное определение (рис. 8.4).

Листинг 8.15 служит иллюстрацией того, как автоматические переменные локализуются в функции или в блоке, который их содержит.

Результаты выполнения программы из листинга 8.15:

```
In main(), texas = 31, &texas = 0068FDF0
In main(), year = 1999, &year = 0068FDF4
In oil(), texas = 5, &texas = 0068FDE0
In oil(), x = 31, &x = 0068FDEC
In block, texas = 113, &texas = 0068FDCC
In block, x = 31, &x = 0068FDEC
Post-block texas = 5, &texas = 0068FDE0
In main(), texas = 31, &texas = 0068FDF0
In main(), year = 1999, &year = 0068FDF4
```

Обратите внимание на то, что каждая из трех переменных `texas` имеет свой собственный отличный от других адрес и программа использует только ту переменную, которая на текущий момент находится в диапазоне доступа, присваивая, таким образом, переменной `texas` значение 113 во внутреннем блоке функции `oil()`, что никак не отражается на других переменных с тем же именем.

Подведем итоги последовательности событий. В момент начала работы `main()` программа выделяет пространство памяти для размещения переменных `texas` и `year` и обе эти переменные попадают в диапазон доступа. В момент, когда программа обращается к функции `oil()`, эти переменные остаются в памяти, но исчезают из диапазона доступа. Две новые переменные, `x` и `texas`, также размещаются в памяти и попадают в диапазон доступа. Когда выполнение программы передается внутреннему блоку в функции `oil()`, новая переменная `texas` выходит из зоны видимости и вытесняется более новым определением. Переменная `x` остается в диапазоне доступа, поскольку блок не определяет новую переменную `x`.

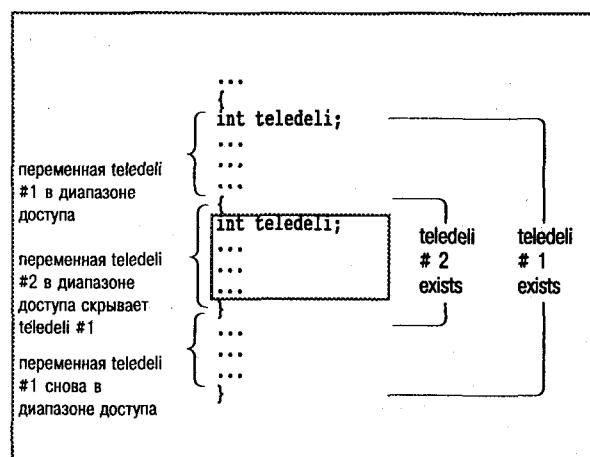


РИСУНОК 8.4 Блоки и диапазоны доступа.

**Листинг 8.15 Программа auto.cpp.**

```
// auto.cpp - иллюстрирует диапазон доступа автоматических переменных
#include <iostream>
using namespace std;
void oil(int x);
int main()
{
// ПРИМЕЧАНИЕ: некоторые версии требуют, чтобы вы привели в этой программе
// адреса к типу unsigned
 int texas = 31;
 int year = 1999;
 cout << "In main(), texas = " << texas << ", &texas =" ;
 cout << &texas << "\n";
 cout << "In main(), year = " << year << ", &year =" ;
 cout << &year << "\n";
 oil(texas);
 cout << "In main(), texas = " << texas << ", &texas =" ;
 cout << &texas << "\n";
 cout << "In main(), year = " << year << ", &year =" ;
 cout << &year << "\n";
 return 0;
}
void oil(int x)
{
 int texas = 5;
 cout << "In oil(), texas = " << texas << ", &texas =" ;
 cout << &texas << "\n";
 cout << "In oil(), x = " << x << ", &x =" ;
 cout << &x << "\n";
 { // начало блока
 int texas = 113;
 cout << "In block, texas = " << texas;
 cout << ", &texas = " << &texas << "\n";
 cout << "In block, x = " << x << ", &x =" ;
 cout << &x << "\n";
 } // конец блока
 cout << "Post-block texas = " << texas;
 cout << ", &texas = " << &texas << "\n";
}
```

Когда выполнение передается за пределы этого блока, память для самой новой переменной **texas** освобождается и **texas** номер 2 снова возвращается в диапазон доступа. Как только выполнение функции **oil()** прекращается, эта переменная **texas** и **x** перестают существовать, а в диапазон доступа возвращаются исходные переменные **texas** и **year**.

Между прочим, вы можете воспользоваться ключевым словом **auto** языка C++ (и C) для того, чтобы явно указать класс памяти:

```
int froob(int n)
{
 auto float ford;
 ...
}
```

Поскольку ключевое слово **auto** можно использовать только при работе с переменными, которые уже являются автоматическими по умолчанию, программисты ред-

ко прибегают к его использованию. Время от времени оно используется для того, чтобы программный код был более понятным пользователю. Например, вы можете намеренно воспользоваться этим словом, чтобы показать, что создается автоматическая переменная, которая перекрывает глобальное определение, подобное тому, которое мы вскоре рассмотрим.

**Автоматические переменные и работа со стеком**

Вы можете получить более полное представление об автоматических переменных, если будете знать, как обычный компилятор языка C++ формирует их. Поскольку число автоматических переменных растет или сокращается по мере того, как функции начинают или прекращают выполняться, программа должна управлять автоматическими переменными в процессе своего выполнения. Используемый с этой целью подход обычно состоит в том, что выделяется специальная область

памяти, которая используется в качестве стека по мере появления и исчезновения переменных. Эта область называется стеком, поскольку новые данные помещаются, figurально выражаясь, поверх старых данных (т.е. в смежных, а не в тех же самых ячейках памяти), а затем удаляются из стека, после того как программа завершит работу с ними. По умолчанию размер стека зависит от реализации, однако в общем случае у компилятора имеются средства, позволяющие изменять размер стека. Программа отслеживает местоположение стека, используя для этой цели два указателя. Один из них указывает на базу стека, с которой начинается область памяти, выделенная под стек, а другой указывает на вершину стека, которая представляет собой следующую ячейку свободной памяти. В момент, когда производится обращение к функции, ее автоматические переменные добавляются в стек, а указатель вершины устанавливается на свободную ячейку памяти, следующую за только что размещенными переменными. Когда выполнение функции прекращается, указатель вершины снова получает значение, которое он имел перед тем, как произошло обращение к функции, в результате чего освобождается память, которая была использована для хранения новых переменных.

Стек реализует архитектуру LIFO (last in-first out — последним пришел, первым обслужен), что означает, что переменная, которая попала в стек последней, удаляется из него первой. Такой механизм упрощает передачу аргументов. Вызов функции помещает значения своих аргументов в вершину стека и производит перестановку указателя вершины. Вызванная функция использует описание своих формальных аргументов для поиска адреса каждого аргумента. Например, на рис. 8.5 показана функция `fib()`, которая в момент вызова передает 2-байтовое значение `int` и 4-байтовое значение `long`. Эти значения попадают в стек. Когда функция `fib()` начинает выполняться, она связывает имена `real` и `tell` с этими двумя значениями. Как только прекратится выполнение функции `fib()`, указатель вершины стека восстанавливает ссылку на прежнюю позицию. Новые значения не удаляются, но теперь они лишаются меток, и пространство памяти, которое они занимают, будет использовано следующим процессом, в ходе выполнения которого будут размещаться значения в стеке. (Рисунок представляет немного упрощенную картину, так как при вызове функции может передаваться дополнительная информация, например, адреса возврата.)

Вы, должно быть, уже заметили, что адреса в программе, представленной в листинге 8.15, уменьшаются, а не увеличиваются по мере того, как добавляются новые переменные. Это объясняется тем, что конкретный компилятор языка C++ организует стек в направлении

сверху вниз. При этом меняется направление возрастания стека, однако это не нарушает принцип, заложенный в его основу.

### Переменные типа `register`

Язык C++, как и C, поддерживает ключевое слово `register`, предназначенное для объявления автоматических переменных. Это ключевое слово информирует компилятор, что вы хотите использовать регистр центрального процессора вместо стека для манипулирования конкретной переменной. Идея заключается в том, что центральный процессор получает доступ к некоторому значению гораздо быстрее, чем осуществляется доступ к памяти в стеке. Чтобы объявить регистровую переменную, следует предварить объявление типа переменной ключевым словом `register`:

```
register int count_fast; // запрос на
// создание регистровой переменной
```

Вы, по-видимому, заметили, как употребляются квалифицирующие слова "указание" и "запрос". Компилятор вовсе не обязательно должен выполнять запрос. Например, регистры уже могут быть заняты либо запрашивается создание типа, который не подходит для регистра. В настоящее время широко распространено убеждение, что современные компиляторы достаточно "разумны", чтобы обходиться без каких-либо указаний.

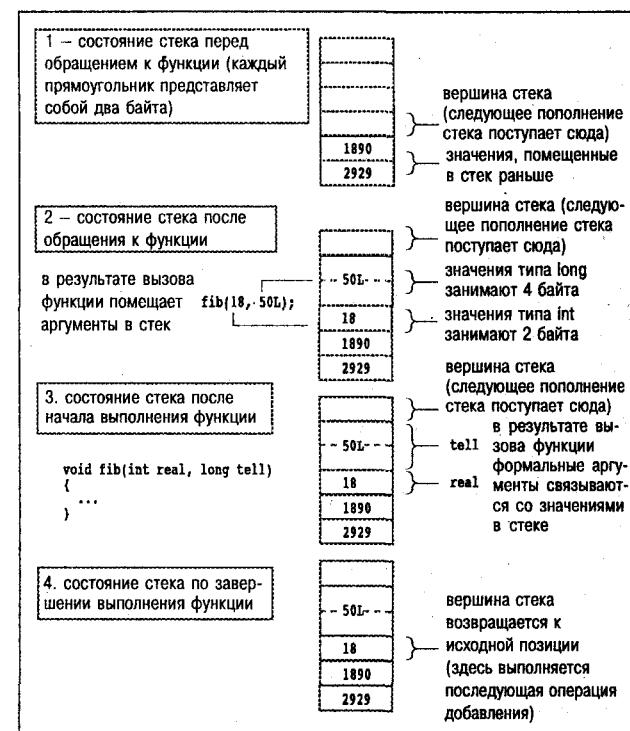


РИСУНОК 8.5 Передача аргументов с помощью стека.

Например, если вы применяете цикл `for`, то компилятор может сам принять решение использовать регистр для отслеживания изменения индекса цикла.

## Статический класс памяти

Переменные, которые принадлежат статическому классу памяти, существуют в течение всего времени выполнения программы, они обладают большей долговечностью, чем автоматические переменные. Поскольку число статических переменных не меняется на протяжении выполнения программы, сама программа не нуждается в специальных механизмах, подобных стеку, чтобы манипулировать ими. Компилятор просто резервирует фиксированный блок памяти для хранения всех статических переменных, и эти переменные доступны программе на протяжении всего времени ее выполнения. Более того, если вы явно не инициализировали статическую переменную, компилятор устанавливает ее равной нулю. Элементы статических массивов и структур устанавливаются равными нулю по умолчанию.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Классический стандарт K&R C не позволяет вам инициализировать автоматические массивы и структуры, однако он позволяет инициализировать статические массивы и структуры. ANSI C и C++ допускают инициализацию обоих этих типов. Однако некоторые ранние трансляторы C++ используют компиляторы языка C, которые не полностью совместимы с ANSI C. Если вы пользуетесь такого рода версией, вам, по-видимому, следует использовать один из трех видов статических классов памяти для инициализации массивов и структур.

В C++, как и в C, имеется три вида статических переменных: внешние, статические и внешние статические. Все это может показаться вам немного запутанным. К сожалению, C++ использует слово *static* в двух различных смыслах. В одном случае подразумевается пе-

ременная, которая существует на протяжении выполнения программы. В этом смысле все три вида являются статическими. В другом случае подразумевается, насколько широко известна эта переменная, т.е. имеется в виду диапазон доступа и связывание. Внешняя переменная доступна всем файлам программы (глобальный диапазон доступа, внешнее связывание), внешняя статическая переменная доступна всем функциям одного файла (глобальный диапазон доступа, внутреннее связывание), а статическая переменная, объявленная в блоке, ограничена этим блоком (диапазон доступа блока, внутреннее связывание). Свойства классов памяти, какими они были в эпоху отсутствия пространства имен, приведены в табл. 8.2. Вы уже сталкивались с некоторыми из этих сведений, сейчас мы рассмотрим эти вопросы более подробно.

## Внешние переменные

Внешние переменные потому и называются так, что они определены вне пределов функции и поэтому являются внешними по отношению к любой функции. Например, они могут быть объявлены перед функцией `main()`. Вы можете воспользоваться внешней переменной в любой функции, которая следует в файле за определением внешней переменной. Таким образом, внешние переменные считаются также глобальными по отношению с автоматическими переменными, которые являются локальными. Тем не менее, если вы определяете автоматическую переменную с тем же именем, что и внешняя переменная, именно автоматическая переменная попадает в диапазон доступа, когда программа выполняет эту конкретную функцию. Программа, представленная в листинге 8.16, может служить иллюстрацией этих трех моментов. Она также показывает, как можно пользоваться ключевым словом `extern` для повторного

Таблица 8.2 Классы памяти.

| Класс памяти        | Как создавался                                                                                                  | Диапазон доступа | Связывание | Продолжительность существования        |
|---------------------|-----------------------------------------------------------------------------------------------------------------|------------------|------------|----------------------------------------|
| Автоматический      | По умолчанию для параметров функции и переменных, объявленных в теле функции                                    | Локальный        | Внутреннее | Выполняется во время определения блока |
| Внешний             | По умолчанию для переменных, объявленных за пределами какой-либо из функций                                     | Глобальный       | Внешнее    | На протяжении выполнения программы     |
| Статический         | Путем присоединения ключевого слова <code>static</code> к описанию переменной, объявленной в теле функции       | Локальный        | Внутреннее | В течение выполнения программы         |
| Внешний статический | Путем присоединения ключевого слова <code>static</code> к описанию переменной, объявленной за пределами функции | Глобальный       | Внутреннее | Во время выполнения программы          |

**Листинг 8.16 Программа external.cpp.**

```

// external.cpp - внешние переменные
#include <iostream>
using namespace std;
// внешняя переменная
double warming = 0.3;
// прототипы функций
void update(double dt);
void local();
int main() // используется глобальная переменная
{
 cout << "Global warming" << warming << "degrees.\n";
 update(0.1); // вызов функции для изменения уровня подогрева
 cout << "Global warming" << warming << "degrees.\n";
 local(); // вызов функции для локального подогрева
 cout << "Global warming" << warming << "degrees.\n";
 return 0;
}
void update(double dt) // модифицирует глобальную переменную
{
 extern double warming; // необязательное повторное описание
 warming += dt;
 cout << "Updating global warming to" << warming;
 cout << "degrees.\n";
}
void local() // используется локальная переменная
{
 double warming = 0.8; // новая переменная делает внешнюю переменную невидимой
 cout << "Local warming = " << warming << "degrees.\n";
 // Доступ к глобальной переменной посредством операции определения диапазона доступа
 cout << "But global warming = " << ::warming;
 cout << "degrees.\n";
}

```

объявления внешней переменной, объявленной ранее, и как можно использовать операцию определения диапазона доступа для реализации доступа к скрытой в любых других случаях внешней переменной.

Результаты выполнения программы:

```

Global warming is 0.3 degrees.
Updating global warming to 0.4 degrees.
Global warming is 0.4 degrees.
Local warming = 0.8 degrees.
But global warming = 0.4 degrees.
Global warming is 0.4 degrees.

```

**Примечания к программе**

Результаты вычисления программы служат иллюстрацией того, что обе функции, `main()` и `update()`, имеют возможность доступа к внешней переменной `warming`. Обратите внимание на тот факт, что те изменения, которым функция `update()` подвергает переменную `warming`, проявляются в последующих случаях использования этой переменной.

Функция `update()` выполняет повторное описание переменной `warming`, используя для этой цели ключе-

вое слово `extern`. Это ключевое слово означает "использовать переменную по данному имени, определенному ранее внешним образом". Поскольку так или иначе функция `update()` должна делать именно это, в случае, когда вы по тем или иным причинам вообще не выполните объявление, это описание следует рассматривать как необязательное. Оно служит целям документирования того факта, что данная функция разработана с расчетом на использование внешней переменной. Исходное объявление

```
double warming = 0.3;
```

называется *описательным объявлением* или просто *определением*. В результате реализуется выделение памяти под конкретную переменную. Повторное объявление той же переменной

```
extern double warming;
```

называется *ссыльчным объявлением* или просто *объявлением*. Оно не выполняет выделение памяти, так как ссылается на существующую переменную. Вы можете воспользоваться ключевым словом `extern` только в

объявлении, касающемся переменной, описанной в какой-либо другой части программы (или в других функциях — подробнее об этом будет рассказано в последующих разделах). К тому же вы не можете инициализировать переменную в ссылочном объявлении:

```
extern double warming = 0.5; //НЕПРАВИЛЬНО
```

Вы можете инициализировать переменную в объявлении только в том случае, когда объявление размещает эту переменную в памяти, т.е. только в случае описательного объявления. В конце концов, термин *инициализация* означает присваивание значения ячейке памяти при выделении этой ячейки конкретной переменной.

Функция **local()** показывает, что, когда вы объявляете локальную переменную с тем же именем, что и у глобальной переменной, локальная переменная делает глобальную версию этой переменной невидимой. Функция **local()**, например, использует локальное определение переменной **warming**, когда она отображает значение переменной **warming**.

Язык C++ идет дальше С, когда предлагает *оператор определения диапазона доступа* (::). Когда этот оператор предшествует имени переменной, это значит, что используется глобальная версия этой переменной. Таким образом, функция **local()** отображает значение **warming**, равное 0.8, в то время как она же отображает значение ::**warming** равным 0.4. Вы еще встретите этот оператор при обсуждении пространств имен и классов.

### ЛОКАЛЬНАЯ ИЛИ ГЛОБАЛЬНАЯ ПЕРЕМЕННАЯ?

Теперь, когда вы имеете возможность выбирать, какие переменные использовать, глобальные или локальные, каким из них вы отдаите предпочтение? На первый взгляд глобальные переменные кажутся более привлекательными. Поскольку все функции имеют доступ к глобальным переменным, вам не нужно беспокоиться о том, как правильно передать аргументы. Однако такой легкий доступ достается дорогой ценой — за счет снижения надежности программ. Опыт программирования показывает — чем больше усилий затрачено программой на изоляцию данных от нежелательного доступа, тем благотворнее это отразится на целостности этих данных. В большинстве случаев следует пользоваться локальными переменными и передавать данные функциям только в случае необходимости, а не допускать такой же доступ к этим данным, какой обеспечивают глобальные переменные. Как вы сможете убедиться позже, объектно-ориентированное программирование делает очередной шаг в направлении углубления изоляции данных.

Однако и у глобальных переменных имеется своя область применения. В частности, у вас может быть блок данных, который используется несколькими функциями, например, массив с названиями месяцев или список атомных весов химических элементов. Класс внешней памяти больше других подходит для представления постоянных данных, так как в этом случае вы можете воспользоваться ключевым словом **const**, чтобы защитить данные от изменений.

```
const char * const months[12] =
{
 "January", "February", "March", "April",
 "May", "June", "July", "August",
 "September", "October", "November",
 "December"
};
```

Первое ключевое слово **const** защищает от изменений строку, а второе слово **const** гарантирует, что каждый указатель в массиве будет указывать на ту же строку, на какую он указывал первоначально.

## Модификатор static (локальные переменные)

Модификатор **static** может быть использован как с локальной переменной, так и с глобальной. Сейчас мы рассмотрим случай использования с локальной переменной. Если вы пользуетесь ею внутри блока, модификатор **static** обеспечивает принадлежность локальной переменной статическому классу памяти. Это означает, что если переменная известна в пределах блока, то она существует, даже если блок неактивен. Таким образом, статическая локальная переменная может сохранять свое значение и между двумя вызовами функции. (Статические переменные могут оказаться полезными для вашего собственного перевоплощения — вы можете использовать их для передачи секретного номера счета в швейцарском банке, чтобы затем воспользоваться им во время следующего визита в этот банк.) Кроме того, если вы производите инициализацию статической локальной переменной, то это достаточно выполнить только один раз, во время запуска программы. Последующие вызовы функции не приводят к повторной инициализации такой переменной, как это имеет место в случае с автоматическими переменными. Программа, представленная в листинге 8.17, служит иллюстрацией этих моментов.

Между прочим, программа демонстрирует один из способов ввода строк, размеры которых могут превосходить размеры массивов, предназначенных для их хранения. Метод ввода **cin.get(input,ArSize)**, как вы, должно быть, помните, производит чтение до конца строки или чтение **ArSize - 1** символов. Метод помещает новую строку во входную очередь. Данная программа читает символ, который следуют за вводом строки. Если это символ новой строки, то это означает, что считана вся строка. Если прочитанный символ не является символом новой строки, значит, во входной строке еще остались символы. Рассматриваемая программа использует цикл, чтобы отвергнуть оставшуюся часть строки, тем не менее, вы можете внести соответствующие изменения в программу, чтобы можно было обработать оставшуюся часть строки в следующем цикле ввода. Про-

грамма также использует то обстоятельство, что при попытке прочитать пустую строку посредством функции `get(char *, int)` вызывается проверка `cin` на соответствие значению `false`.

### Листинг 8.17 Программа static.cpp.

```
// static.cpp - использование статической
// локальной переменной
#include <iostream>
using namespace std;
// constants
const int ArSize = 10;
// прототип функции
void strcount(const char * str);

int main()
{
 char input[ArSize];
 char next;

 cout << "Enter a line:\n";
 cin.get(input, ArSize);
 while (cin)
 {
 cin.get(next);
 while (next != '\n') // строка не
 // помещается!
 cin.get(next);
 strcount(input);
 cout << "Enter next line (empty line
 ↪to quit):\n";
 cin.get(input, ArSize);
 }
 cout << "Bye\n";
 return 0;
}

void strcount(const char * str)
{
 static int total = 0; // статическая
 // локальная переменная
 int count = 0; // автоматическая
 // локальная переменная
 cout << "\"" << str << "\" contains ";
 while (*str++) // переход в конец
 // строки
 count++;
 total += count;
 cout << count << "characters\n";
 cout << total << "characters total\n";
}
```

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Некоторые более ранние компиляторы не реализуют требование, согласно которому в случае, если функция `cin.get(char *, int)` читает пустую строку, она устанавливает бит признака ошибки, что приводит к осуществлению проверки того, имеет ли `cin` значение `false`. В этом случае вы можете заменить проверку

```
while (cin)
```

на:

```
while (input[0])
```

Либо в случае проверки, которая срабатывает как в старых, так и в новых версиях, выполните следующее:

```
while (cin && input[0])
```

Результаты выполнения программы:

```
Enter a line:
nice pants
"nice pant" contains 9 characters
9 characters total
Enter next line (empty line to quit):
thanks
"thanks" contains 6 characters
15 characters total
Enter next line (empty line to quit):
parting is such sweet sorrow
"parting i" contains 9 characters
24 characters total
Enter next line (empty line to quit):
ok
"ok" contains 2 characters
26 characters total
Enter next line (empty line to quit):
Bye
```

Обратите внимание на тот факт, что поскольку размер массива равен 10, программа не читает более девяти символов из каждой строки. Кроме того, заметьте, что автоматическая переменная `count` инициализируется значением 0 всякий раз, когда происходит обращение к функции. Тем не менее, значение статической переменной `total` устанавливается равным 0 только один раз, в самом начале. После этого переменная `total` сохраняет свое значение и в промежутке между двумя обращениями, благодаря чему она способна вычислять текущий итог.

### Связывание и внешние переменные

Применение модификатора `static` ко внешней переменной имеет смысл в многофайловых программах. В этом контексте статическая внешняя переменная является локальной по отношению к файлу, в котором она содержится. Для нее характерно внутреннее связывание. Однако обычная внешняя переменная обладает внешним связыванием, что означает, что она может быть использована в различных файлах. При внешнем связывании один и только один файл может содержать внешнее определение переменной. Все другие файлы, в которых будет использоваться эта переменная, должны применять ключевое слово `extern` в ссылочном объявлении (рис. 8.6).

Если в файле отсутствует объявление `extern` для переменной, он не может использовать внешнюю переменную, определенную где-нибудь в другом месте:

```
// файл 1
int errors = 20; // глобальное объявление
...

// файл 2
... // отсутствует объявление extern
 // int errors
void frobisch()
{
 cout << errors; // неудачная попытка
 // воспользоваться переменной errors
 ...
```

```
// file1.cpp
#include <iostream>
using namespace std;

// function prototypes
#include "mystuff.h"

// defining an external variable
int process_status = 0;

void promise ();
int main()
{
 ...
}

void promise ()
{
 ...
}
```

В этом файле используется переменная `process_status`, поэтому компилятор должен отвести под нее пространство памяти

```
// file2.cpp
#include <iostream>
using namespace std;

// function prototypes
#include "mystuff.h"

// referencing an external variable
extern int process_status;

int manipulate(int n)
{
 ...
}

char * remark(char * str)
{
 ...
}
```

В этом файле используется модификатор `extern`, инструктирующий программу относительно применения переменной `process_status`, которая определена в другом файле

```
// файл 2
static int errors = 5; // известна только
 // файлу 2

void froobish()
{
 cout << errors; // использует
 // переменную errors, объявленную в
 // файле 2
 ...
}
```

### ПОМНИТЕ

В многофайловой программе вы можете объявить внешнюю переменную только в одном файле. Все остальные файлы, использующие эту переменную, должны объявить эту переменную с использованием ключевого слова `extern`.

Чтобы обеспечить совместное использование данных различными частями многофайловой программы, используйте обычную внешнюю переменную. Чтобы обеспечить совместное использование данных различными функциями, сосредоточенными в одном файле, используйте статическую внешнюю переменную. (Пространство имен представляют в этом отношении альтернативный метод.) Наряду с этим, если вы переводите внешнюю переменную в категорию статических, то вас не затрагивает проблема конфликта ее имени с внешними переменными, содержащимися в других файлах.

Программы, представленные в листингах 8.18 и 8.19 показывают, как C++ манипулирует внешними и статическими внешними переменными. В программе из листинга 8.18 (`twofile1.cpp`) объявлены внешние переменные `tom` и `dick` и статическая внешняя переменная `harry`. Функция `main()` в этом файле отображает адреса этих трех переменных, а затем обращается к функции `remote_access()`, которая определена во втором файле. В листинге 8.19 (`twofile2.cpp`) показан подобный файл. В дополнение к определению функции `remote_access()` этот файл использует ключевое слово `extern`, чтобы совместно использовать переменную `tom` из первого файла. Далее этот файл объявляет статическую переменную с именем `dick`. Модификатор `static` делает эту переменную локальной по отношению к файлу и перекрывает глобальное определение. Затем этот файл определяет внешнюю переменную с именем `harry`. Он может себе это позволить, не вступая в конфликт с переменной `harry` из первого файла, поскольку первая переменная `harry` обладает только внешним связыванием. После этого функция `remote_access()` отображает адреса этих трех переменных, так что вы можете сравнить их с адресами соответствующих переменных из первого файла. Не забывайте, что нужно компилировать и связывать оба файла, чтобы получить полную версию программы.

**РИСУНОК 8.6 Описательное и ссылочное объявления.**

Если в файле дается определение второй внешней переменной с тем же именем, то это ошибка:

```
// файл 1
int errors = 20; // внешнее объявление...

// файл 2
int errors; // неправильное объявление
void froobish()
{
 cout << errors; // неудачная попытка
 // воспользоваться переменной errors
 ...
}
```

Правильный подход состоит в использовании ключевого слова `extern` во втором файле:

```
// файл 1
int errors = 20; // внешнее объявление
...

// файл 2
extern int errors; // ссылка на переменную
 // errors из файла 1
void froobish()
{
 cout << errors; // использует
 // переменную errors, определенную
 // в файле 1
 ...
}
```

Однако если в файле объявляется статическая внешняя переменная с тем же именем, что и обычная внешняя переменная, уже объявленная в другом файле, то в диапазон доступа этого файла попадает статическая версия:

```
// файл 1
int errors = 20; // внешнее объявление
...

```

**Листинг 8.18 Программа twofile1.cpp.**

```
// twofile1.cpp - внешние и статические внешние переменные
#include <iostream> // компилируется с двумя file2.cpp
using namespace std;
int tom = 3; // определение внешней переменной
int dick = 30; // определение внешней переменной
static int harry = 300; // определение статической внешней переменной

// прототип функции
void remote_access();

int main()
{
// ПРИМЕЧАНИЕ: некоторые версии требуют, чтобы вы привели адреса к типу unsigned
 cout << "main() reports the following addresses:\n";
 cout << &tom << " = &tom, " << &dick << " = &dick, ";
 cout << &harry << " = &harry\n";
 remote_access();
 return 0;
}
```

**Листинг 8.19 Программа twofile2.cpp.**

```
// twofile2.cpp - внешние и статические внешние переменные
#include <iostream>
using namespace std;
extern int tom; // переменная tom определена в другом месте
static int dick = 10; // перекрывает внешнюю переменную dick
int harry = 200; // определение внешней переменной,
 // без конфликта с переменной harry из файла twofile1

void remote_access()
{
// ПРИМЕЧАНИЕ: некоторые версии требуют, чтобы вы привели адреса к типу unsigned
 cout << "remote_access() reports the following addresses:\n";
 cout << &tom << " = &tom, " << &dick << " = &dick, ";
 cout << &harry << " = &harry\n";
}
```

Результаты выполнения этих программ:

```
main() reports the following addresses:
00450E50 = &tom, 00450E54 = &dick, 00450E58 = &harry
remote_access() reports the following addresses:
00450E50 = &tom, 00450F40 = &dick, 00450F44 = &harry
```

## Спецификаторы классов памяти: **const**, **volatile** и **mutable**

Некоторые ключевые слова C++, называемые *спецификаторами*, являются носителями дополнительной информации о памяти. Наиболее широко применяется спецификатор **const**, и вы уже знаете, для чего он предназначен. Он указывает, что память, будучи однажды инициализирована, не может быть изменена программой. Чуть позже мы вернемся к обсуждению спецификатора **const**.

Ключевое слово **volatile** указывает, что значение в ячейке памяти может быть изменено, даже если в про-

граммном коде нет ничего такого, что может модифицировать содержимое ячейки. Такое утверждение звучит таинственно, но все объясняется достаточно просто. Например, пусть у вас имеется указатель на аппаратную область памяти, в которой хранятся показания времени или информация из последовательного порта. В этом случае аппаратные средства, а не программа, изменяют содержимое этой области. Назначение этого ключевого слова состоит в оптимизации возможностей компилятора. Например, предположим, что компилятор обнаружил, что программа использует значение некоторой переменной дважды на протяжении выполнения нескольких операторов. Вместо того чтобы заставить

программу дважды предпринимать поиск этого значения, компилятор может запомнить это значение в реестре. Такая оптимизация предполагает, что значение этой переменной не изменяется между двумя случаями ее использования. Если вы не объявили эту переменную как **volatile**, то компилятор вправе считать, что может проводить такого рода оптимизацию. Если вы объявили конкретную переменную как **volatile**, вы тем самым сообщаеете компилятору, что такую оптимизацию проводить не следует.

С недавних пор в C++ реализован новый спецификатор: **mutable**. Вы можете употреблять его с целью объявить, что некоторый конкретный элемент той или иной структуры (или класса) может быть изменен, даже если эта структура (класс) были объявлены как **const**. В качестве примера рассмотрим следующий программный код:

```
struct data
{
 char name[30];
 mutable int accesses;
 ...
};

const data veep =
{ "Claybourne Clodde", 0, ... };
strcpy(veep.name, "Joye Joux");
 // не допускается
veep.accesses++; // допускается
```

Спецификатор **const** структуры **veep** препятствует изменению ее элементов со стороны программы, однако спецификатор **mutable**, предшествующий элементу **accesses**, снимает с элемента **accesses** это ограничение.

В этой книге спецификаторы **volatile** и **mutable** не используются, зато далее мы больше узнаем о спецификаторе **const**.

### Более подробно о спецификаторе **const**

В C++ (но не в C) спецификатор **const** привносит небольшие изменения в классы памяти, используемые по умолчанию. В то время как глобальная переменная по умолчанию обладает внешним связыванием, глобальная переменная **const** по умолчанию обладает внутренним связыванием. Другими словами, в C++ глобальное определение **const** рассматривается как статическое глобальное определение. Вы можете считать, что ключевому слову **const**, использованному во внешнем определении, неявно предшествует ключевое слово **static**.

```
const int fingers = 10; // то же, что и
 // static const int fingers;
int main(void)
...
```

Чтобы облегчить вам жизнь, в C++ правила, регламентирующие использование постоянных типов, были несколько изменены. Предположим, например, что вы установили константу, которую хотели бы поместить в заголовочный файл, и что вы используете этот заголовочный файл в нескольких файлах одной и той же программы. После того как препроцессор включит содержимое этого заголовочного файла в каждый исходный файл, во всех исходных файлах появятся такие определения:

```
const int fingers = 10;
const char * warning = "Wak!";
```

Если определения глобального спецификатора **const** обладают внешним связыванием, как обычные переменные, то это свидетельствует об ошибке, так что вы можете объявить глобальную переменную только в одном файле. Другими словами, только один файл может содержать представленное выше объявление, в то время как в других файлах должны быть предусмотрены ссылочные объявления с использованием ключевого слова **extern**. Более того, только объявления без ключевого слова **extern** обеспечивают инициализацию значений:

```
// потребуется extern, если const обладает
// внешним связыванием
extern const int fingers; // не может быть
 // инициализирована
extern const char * warning;
```

Итак, вам понадобится один набор определений для одного файла и другой набор для остальных файлов. В то же время для данных **const** с внешним определением характерно внутреннее связывание, и вы можете использовать одно и то же определение во всех файлах.

Внутреннее связывание также означает, что каждый файл получает собственный набор констант, а не использует их совместно с другими файлами. Каждое определение является приватным по отношению к файлу, который его содержит. Таким образом, если вы включаете один и тот же заголовочный файл в два файла с исходными программными кодами, оба они получают один и тот же набор констант.

Если по какой-либо причине вы хотите, чтобы константа имела внешнее связывание, то можете воспользоваться ключевым словом **extern**, чтобы перекрыть внутреннее связывание, установленное по умолчанию:

```
extern const states = 50; // внешнее
 // связывание
```

Чтобы объявить константу во всех файлах, которые ее используют, нужно воспользоваться ключевым словом **extern**. В этом заключается отличие от обычной внешней переменной, при объявлении которой вы не используете ключевого слова **extern**, но употребляете **extern** в

тех файлах, которые используют эту переменную. Таким образом, в отличие от обычных переменных, вы получаете возможность инициализировать значение `extern const`. И вы обязаны это делать, так как данные `const` требуют инициализации.

Когда вы объявляете `const` в рамках конкретной функции или блока, она имеет диапазон доступа блока, это означает, что эта константа может быть использована только тогда, когда программа выполняет код этого блока. Это также означает, что вы можете создавать константы в функции или в блоке, не заботясь о том, что ее имя может вступить в конфликт с константами, определенными где-нибудь в других местах программы.

## Классы памяти и функции

Функциям тоже выделяются классы памяти, однако у них выбор более ограниченный, чем у переменных. C++, как и C, не позволяет объявить одну функцию внутри другой. Таким образом, все функции автоматически получают статический класс памяти, а это означает, что они существуют, пока выполняется программа. По умолчанию функции считаются внешними в том смысле, что для них характерно внешнее связывание и что они могут использоваться различными файлами. Фактически можно использовать ключевое слово `extern` в прототипе функции, чтобы указать, что данная функция определена в другом файле, однако это не обязательно. (Чтобы программа могла найти функцию в другом файле, этот файл должен быть одним из тех, который компилируется как часть программы, или библиотечным файлом, поиск которого осуществляется редактором связей.) Кроме того, можно воспользоваться ключевым словом `static`, чтобы придать функции внутреннее связывание, ограничивая ее использование одним файлом. Можно применить это ключевое слово к прототипу или к определению функции:

```
static int private(double x);
...
static int private(double x)
{
 ...
}
```

Это означает, что функция известна только в пределах данного файла и что вы можете использовать это же имя для какой-либо функции в другом файле. Что касается переменных, то статическая функция перекрывает внешнее определение файла, содержащего статическое объявление. Если вы определяете собственную функцию `strlen()` в одном из файлов многофайловой программы и объявляете эту функцию как `static`, то именно этот файл использует данное вами определение функции `strlen()`, в то время как другие файлы пользуются би-

лиотечной версией этой функции. Если вы даете определение собственной функции `strlen()` и не объявляете ее как `static`, то компилятор использует вашу версию во всех файлах программы. (См. примечание "Где C++ находит функции?".)

Встроенные функции ведут себя не так, как обычные функции. По умолчанию они обладают внутренним связыванием и в силу этого обстоятельства являются локальными по отношению к содержащим их файлам. По этой причине целесообразно помещать определения встроенных функций в заголовочный файл.

### ГДЕ C++ НАХОДИТ ФУНКЦИИ?

Предположим, вы обращаетесь к функции, которая находится в конкретном файле программы. Где C++ ищет определение этой функции? Если прототип этой функции в данном файле показывает, что функция имеет тип `static`, компилятор проверяет на наличие функции только этот файл. В противном случае компилятор (включая также и редактор связей) просматривает все файлы программы. Если компилятор находит два определения, он передает вам сообщение об ошибке, поскольку у вас должно быть только одно определение внешней функции. Если ему вообще не удастся найти ни одного определения этой функции в ваших файлах, он производит просмотр библиотек. Отсюда следует, что, если вы дадите определение функции с именем, совпадающим с именем библиотечной функции, компилятор воспользуется вашей, а не библиотечной версией. (При этом, однако, возникают проблемы, если вы используете заголовочный файл, в котором содержится объявление библиотечной функции, и если этот прототип не соответствует вашей версии.) Для некоторых компиляторов и редакторов связей требуются явные инструкции, указывающие, какие библиотеки следует просматривать.

## Языковое связывание

Существует другая форма связывания, именуемая *языковым связыванием*, которая затрагивает и функции. Сделаем сначала небольшое пояснение. Редактору связей требуется уникальное символьное имя для каждой отдельной функции. В C это реализуется просто, поскольку в C под одним конкретным именем может быть только одна функция. Следовательно, для внутренних целей компилятор языка C может преобразовывать такое имя C-функции, как `spiff`, в `_spiff`. Подход, принятый в C, получил название *связывания в языке C*. Однако в C++ может быть сразу несколько функций под одним и тем же именем, которые должны быть оттранслированы в отдельные неповторяющиеся символьные имена. Таким образом, компилятор C++ погружается в процесс, получивший название *искажение имен*, который генерирует различные символьные имена перегруженных функций. Например, он может преобразовать имя `spiff(int)`, скажем, в `_spiff_i` и имя `spiff(double, double)` в `_spiff_d_d`. Исповедуемый в C++ подход называется *связыванием в языке C++*.

Когда редактор связей осуществляет поиск функции C++, соответствующей некоторому конкретному функциональному вызову, он использует метод просмотра, отличный от метода, используемого для поиска функции, соответствующей вызову функции в языке C. Но предположим, что вы намерены использовать предварительно откомпилированную функцию из библиотеки языка C в программе, написанной на C++? Например, предположим, в вашей программе имеется такой код:

```
spiff(22); // пользователь намерен
// использовать функцию spiff(int) из
// библиотеки языка С
```

Символьным именем этой функции в файле библиотеки C является `_spiff`, однако для нашего гипотетического редактора связей соглашение, регламентирующее поиск в C++, предусматривает поиск символьного имени `name_spiff_i`. Чтобы обойти эту проблему, вы можете использовать прототип функции, указывающий, каким протоколом нужно воспользоваться:

```
extern "C" void spiff(int); // использовать
// протокол языка С для поиска имени
extern void spoff(int); // использовать
// протокол языка С++ для поиска имени
extern "C++" void spaff(int);
// использовать протокол языка С++
// для поиска имени
```

Первый прототип использует связывание языка C. Второй и третий используют связывание языка C++. Второй делает это по умолчанию, а третий задает связывание явно.

## Классы памяти и динамическое распределение

Классы памяти характеризуют память, распределяемую для переменных (включая массивы и структуры) и функций. Они не могут быть применены к памяти, распределенной посредством оператора `new` языка C++ (или более ранней функции языка C `malloc()`). Мы называем такой вид памяти *динамической памятью*. Как вы видели в главе 4, управление динамической памятью осуществляется операторами `new` и `delete`, а не правилами, определяющими диапазоны доступа и связывание. Таким образом, динамическая память может выделяться одной функции и освобождаться из другой функции. В отличие от автоматической памяти, динамическая память не подчиняется правилу LIFO (last in first out — последним пришел, первым обслужен). Порядок выделения и освобождения памяти зависит от того, когда и как были использованы операторы `new` и `delete`. Как правило, компилятор использует три различные порции памяти: одну — для статических переменных (эта порция памяти мо-

жет быть разбита на разделы), одну — для автоматических переменных и одну — для динамической памяти.

Несмотря на то что концепция классов памяти не применима к динамической памяти, она применима к переменным типа указатель, используемым для отслеживания динамической памяти. Например, предположим, что внутри функции имеется такой оператор:

```
float * p_fees = new float [20];
```

80 байтов памяти (в предположении, что тип `float` содержит четыре байта), выделенных оператором `new`, остаются занятыми до тех пор, пока оператор `delete` не освободит их. Однако указатель `p_fees` перестает существовать, когда закончится выполнение функции, содержащей это объявление. Если вы хотите, чтобы 80 байтов выделенной ей памяти стали доступными другой функции, вы должны передать или возвратить адрес этой памяти той функции. С другой стороны, если вы сделаете это же объявление внешним, то указатель `p_fees` станет доступным всем функциям, которые следуют в файле за этим объявлением. Воспользовавшись конструкцией

```
extern float * p_fees;
```

во втором файле, вы сделаете этот указатель доступным и в этом файле.



### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Память, распределенная с помощью оператора `new`, как правило, освобождается, когда заканчивается выполнение программы. Тем не менее, это не всегда так. Под DOS, например, при некоторых обстоятельствах запрос, требующий выделения крупного блока памяти, может привести к созданию блока, который не удаляется автоматически при окончании выполнения программы.

## Пространства имен

Имена в C++ могут быть присвоены переменным, функциям, структурам, перечислениям, классам, а также элементам классов и структур. Когда программные проекты становятся чересчур громоздкими, возрастает вероятность возникновения конфликта имен. Если вы используете для создания библиотеки класса более одного источника, то можете столкнуться с проблемой конфликта имен. Например, две библиотеки могут определить классы с именами `List`, `Tree` и `Node`, но при этом не соблюдать правила совместимости. Возможно, вы захотите использовать класс `List` из одной библиотеки и класс `Tree` из другой, и при этом каждый из них ожидает взаимодействия с собственной версией класса `Node`. Такие конфликты называются проблемами пространства имен.

Стандарт C++ предусматривает средства, обеспечивающие более совершенное управление диапазонами

доступа имен. Во время написания данной книги эти средства все еще пробивали себе путь на рынок программных продуктов, и может случиться так, что вы используете компилятор, который их не поддерживает. Однако если текущая версия вашего компилятора не поддерживает пространство имен, то следующая, скорее всего, будет их поддерживать.

## Традиционные пространства имен языка C++

Прежде чем приступить к изучению новых средств, ознакомимся со свойствами пространства имен, которые уже реализованы в языке C++, и введем некоторые терминологические понятия. Это поможет вам быстрее освоить понятие пространства имен.

Первым таким термином является *область объявления*. Область объявления — это область, в которой могут осуществляться объявления. Например, вы можете объявить глобальную переменную вне пределов какой-либо функции. Областью объявлений для этой переменной является файл, в котором она объявлена. Если вы объявляете переменную внутри одной из функций, ее областью объявления будет самый дальний внутренний блок, в которой она объявлена.

Вторым таким термином является *потенциальный диапазон доступа*. Потенциальный диапазон доступа переменной простирается от точки, в которой она была

объявлена, до конца ее области объявления. Таким образом, потенциальный диапазон доступа более узкий, чем область объявления, поскольку вы не можете использовать переменную в программе ранее той точки, в которой она впервые была определена.

Однако переменная может оказаться видимой не в каждой точке потенциального диапазона доступа. Например, она может быть перекрыта другой переменной с тем же именем, объявленной во вложенной области объявлений. Например, локальная переменная, объявленная в функции (областью объявления служит функция) перекрывает глобальную переменную, объявленную в том же файле (областью объявления служит сам файл). Та порция программы, которая фактически может видеть эту переменную, называется *диапазоном доступа*, в этом смысле мы использовали этот термин до сих пор. Рисунки 8.7 и 8.8 служат иллюстрацией применимости терминов области объявления, потенциального диапазона доступа и диапазона доступа.

Правила C++, касающиеся глобальных и локальных переменных, определяют своего рода иерархию пространства имен. В каждой области определения могут быть определены имена, независимые от имен, объявленных в других областях определения. Локальная переменная, объявленная в одной функции, не вступает в конфликт с локальной переменной, объявленной в другой функции.

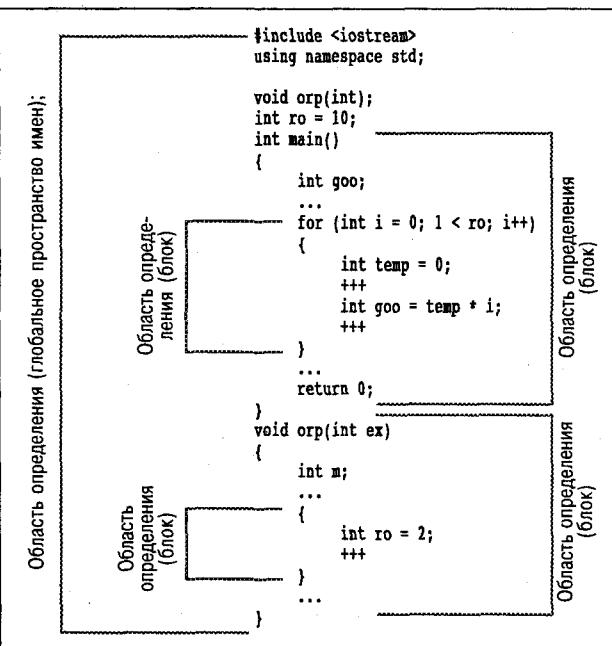


РИСУНОК 8.7 Области объявления.

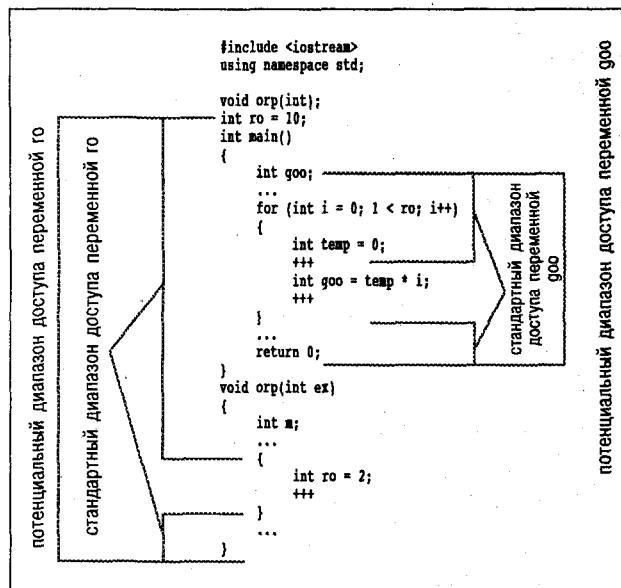


РИСУНОК 8.8 Потенциальный и стандартный диапазоны доступа.

## Новые свойства пространства имен

В настоящее время в языке C++ появилась возможность создавать именованные пространства имен путем определения области определения нового вида, одно из основных назначений которой состоит в выделении области памяти, в которой осуществляется объявление имен. Имена из одного пространства памяти не конфликтуют с теми же именами, объявленными в других пространствах имен, при этом существуют механизмы, позволяющие другим частям программы использовать объекты данных, объявленные в пространстве имен. Например, в приведенном ниже программном коде используется новое ключевое слово **namespace** в целях создания двух пространств имен **Jack** и **Jill**.

```
namespace Jack {
 double pal;
 void fetch();
 int pal;
 struct Well { ... };
}

namespace Jill {
 double bucket(double n) { ... }
 double fetch();
 int pal;
 struct Hill { ... };
}
```

Пространства имен могут находиться на глобальном уровне или внутри других пространств имен, однако они не могут быть помещены в блок. Следовательно, имя, объявленное в пространстве имен, обладает внешним связыванием по умолчанию (пока оно не ссылается на константу или встроенную функцию).

Кроме пространств имен, объявленных пользователем, существует еще одно пространство имен — глобальное. Это соответствует области объявлений на уровне файла, следовательно, то, что раньше подразумевалось под глобальными переменными, сейчас описывается как часть глобального пространства имен.

Имена из одного пространства имен не могут конфликтовать с именами из другого пространства имен. Таким образом, имя **fetch** в пространстве имен **Jack** вполне может существовать с именем **fetch** в пространстве **Jill**, а **Hill** из **Jill** может мирно существовать с внешним именем **Hill**. Правила, регламентирующие объявления и определения в пространствах имен, те же, что и правила, регламентирующие глобальные объявления и определения.

Пространства имен *открыты*, это означает, что можно включать новые имена в существующие пространства имен. Например, оператор

```
namespace Jill {
 char * goose(const char *);
}
```

добавляет имя **goose** к списку имен в пространстве имен **Jill**.

Аналогично исходное пространство имен **Jack** содержит прототип функции **fetch()**. Вы можете поместить программный код функции далее в файле (или в другом файле), снова воспользовавшись пространством имен **Jack**:

```
namespace Jack {
 void fetch()
 {
 ...
 }
}
```

Разумеется, нужен метод доступа к именам в конкретном пространстве имен. Простейший путь заключается в использовании оператора определения диапазона доступа (::), позволяющего уточнить имя, указав его вместе с пространством имени:

```
Jack::pail = 12.34; // использование
 // переменной
Jill::Hill mole; // создается объект Queue
Jack::fetch(); // использование функции
```

Имя без добавлений, такое как, например, **pail**, называется *неуточненным именем*, в то время как имя из пространства имен, такое как **Jack:pail**, называется *уточненным именем*.

## **Объявления использования и директивы *using***

Необходимость уточнять имена всякий раз, когда они используются, — не очень привлекательная перспектива, поэтому C++ предлагает два механизма для упрощения использования имен из пространства имен. Первое называется *объявлением использования*, в рамках которого уточненное имя предваряется ключевым словом `using`:

```
using Jill::fetch; // объявление
// использования
```

Объявление использования добавляет отдельное имя в область объявления. Например, объявление использования `Jill::fetch` в `main()` добавляет `fetch` в область объявлений, определенную функцией `main()`. Сделав такое объявление, вы можете использовать имя `fetch` вместо `Jill::fetch`.

```
namespace Jill {
 double bucket(double n) { ... }
 double fetch;
 struct Hill { ... } ;
}
char fetch;
int main()
{
 using Jill::fetch; // поместить fetch
 // в локальное пространство имен
```

```

double fetch; // Ошибка! Уже имеется
 // локальное имя fetch
cin >> fetch; // считывать значение в
 // Jill::fetch
cin >> ::fetch; // считывать
 // значение в глобальную
 // переменную fetch
}

```

Поскольку объявление использования добавляет имя в локальную область объявлений, этот пример препятствует созданию другой переменной с именем `fetch`. Кроме того, как и любая другая локальная переменная, `fetch` перекрывает глобальную переменную с таким же именем. Размещение объявлений использования на внешнем уровне приводит к тому, что соответствующее имя переменной появляется в глобальном пространстве имен.

Второй новый механизм получил название *директивы using*. Он заключается в том, что имени из пространства имен предшествуют ключевые слова `using namespace`, благодаря чему все имена из пространства имен становятся доступными без необходимости использования оператора определения области видимости.

```
using namespace Jack; // делает все имена
 // в пространстве имен Jack доступными
```

При размещении директивы на глобальном уровне имена пространства имен становятся глобально доступными. Вы уже много раз видели, как действует этот механизм:

```
#include <iostream> // размещает имена в
 // пространстве имен std
using namespace std; // делает имена
 // глобально доступными
```

Вы можете сделать имена из пространства имен локально доступными, помещая директиву использования в локальную область объявлений.

Применение директив `using` для импорта всех имен сразу — это не равнозначно использованию многочисленных объявлений использования. Это, скорее, массовое применение оператора определения диапазона доступа. Когда вы прибегаете к помощи объявления использования, это равносильно использованию имени, объявленного в сфере действия объявления использования. Если некоторое имя уже было объявлено в функции, то вы не можете импортировать то же самое имя без объявления использования. Однако если вы применяете директиву `using`, то результат такого применения будет подобен тому, как если бы вы объявили соответствующие имена в наименьшей области объявлений, содержащей как объявление использования, так и само пространство имен. В рассматриваемом ниже примере

рассматривается глобальное пространство имен. Вы можете воспользоваться директивой `using` для импорта конкретного имени, которое уже было объявлено в некоторой функции, но при этом локальное имя будет перекрывать имя из пространства имен точно так, как оно перерывало бы глобальную переменную с тем же именем. Однако вы все еще можете использовать оператор определения области видимости:

```
namespace Jill {
 double bucket(double n) { ... }
 double fetch;
 struct Hill { ... };
}
char fetch; // глобальное пространство
 // имен
int main()
{
 using namespace Jill; // импортирувать
 // имена из пространства имен
 struct Hill Thrill; // создать
 // структуру Jill::Hill
 double water = bucket(2);
 // использовать имя Jill::bucket();
 double fetch; // это не ошибка;
 // отменяется имя Jill::fetch
 cin >> fetch; // считать значения в
 // локальную переменную fetch
 cin >> ::fetch; // считать значение в
 // глобальную переменную fetch
 cin >> Jill::fetch; // считать
 // значение в переменную Jill::fetch
 ...
}
```

В рассматриваемом примере имя `Jill::fetch` помещено в локальное пространство имен. Оно не имеет локальной области видимости, следовательно, оно не перекрывает глобального имени `fetch`. Тем не менее, обе переменные `fetch` становятся доступными, если воспользоваться оператором определения области видимости. Вы, возможно, пожелаете сравнить этот пример с предыдущим, в котором применяется объявление использования.

#### ПОМНИТЕ

Предположим, что одно и то же имя определено как в пространстве имен, так и области объявлений. Если вы попытаетесь применить объявление использования для того, чтобы поместить имя из пространства имен в область определений, оба имени вступят в конфликт и вы получите сообщение об ошибке. Если вы используете директиву использования для переноса имени из пространства имен в область объявлений, то локальная версия этого имени перекроет версию из пространства имен.

Говоря по существу, применение объявления использования более безопасно, поскольку оно показывает именно те имена, которые вы делаете доступными.

И если такое имя вступит в конфликт с локальным именем, компилятор непременно сообщит вам об этом. Директива `using` добавляет все имена без исключений, даже те, которые, возможно, вам не нужны. Если локальное имя вступает в конфликт, то оно перекрывает версию имени из пространства имен, но об этом вы не будете знать. Следовательно, открытый характер пространств имен означает, что полный список имен пространства имен может распространяться на несколько областей, в результате чего трудно узнать, какие именно имена вы добавляете.

Что можно сказать о подходе, используемом в примерах, которые рассматриваются в данной книге?

```
#include <iostream>
using namespace std;
```

Во-первых, заголовочный файл `iostream` все помещается в пространство имен `std`. Затем следующая строка экспортирует все, что содержится в этом пространстве имен, в глобальное пространство имен. Таким образом, этот подход просто заставляет вспомнить времена, предшествовавшие появлению пространства имен. Основное логическое обоснование такого подхода — это целесообразность. Это нетрудно сделать, и, если в вашей системе нет пространства имен, вы можете заменить предыдущие две строки на исходную форму:

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

Однако надежды сторонников пространства имен основываются на том, что пользователи могут оказаться более разборчивыми и будут применять либо оператор определения области видимости, либо объявление использования. Другими словами, они откажутся от использования следующих объявлений:

```
using namespace std; // избегать
// использования этой конструкции
// как слишком неопределенной
```

Вместо нее выполните следующее:

```
int x;
std::cin >> x;
std::cout << x << std::endl;
```

Или следующее:

```
using std::cin;
using std::cout;
using std::endl;
int x;
cin >> x;
cout << x << endl;
```

Вы можете воспользоваться вложенными пространствами имен, как показано ниже, для создания пространства имен, содержащих объявления, которыми вы обычно пользуетесь.

## Немного больше о свойствах пространства имен

Предусмотрена возможность вложения объявлений в пространстве имен:

```
namespace elements
{
 namespace fire
 {
 int flame;
 ...
 }
 float water;
}
```

В рассматриваемом случае вы обращаетесь к переменной `flame` как к переменной `elements::fire::flame`. Аналогично вы можете сделать внутренние имена доступными посредством такой директивы `using`:

```
using namespace elements::fire;
```

Кроме того, можно пользоваться директивами `using` и объявления использования внутри пространства имен:

```
namespace myth
{
 using Jill::fetch;
 using namespace elements;
 using std::cout;
 using std::cin;
}
```

Предположим, вы хотите получить доступ к переменной `Jill::fetch`. В связи с тем что `Jill::fetch` сейчас является частью пространства `myth`, в котором ее можно называть как `fetch`, вы можете получить к ней доступ следующим путем:

```
std::cin >> myth::fetch;
```

Разумеется, поскольку она также является частью пространства имен `Jill`, вы все еще можете обратиться к ней по имени `Jill::fetch`:

```
std::cout << Jill::fetch; // воспроизвести
// значение, считанное в переменную
// myth::fetch
```

Вы можете также сделать это, исключив любой конфликт локальных переменных:

```
using namespace myth;
cin >> fetch; // на самом деле это
// std::cin и Jill::fetch
```

Теперь рассмотрим возможность применения директивы использования в пространстве `myth`. Директива использования обладает свойством *транзитивности*. Мы говорим, что операция *op* транзитивна, если из A *op* B и B *op* C следует A *op* C. Например, оператор `>` обладает свойством транзитивности. (Иначе говоря, из того, что A больше B и B больше C следует, что A больше C.)

В данном контексте результат состоит в том, что выполнение оператора

```
using namespace myth;
```

приводит к тому, что благодаря применению директивы `using` добавляется пространство имен `elements`, это аналогично следующему объявлению:

```
using namespace myth;
using namespace elements;
```

Можно назначить псевдоним пространству имен. Например, предположим, что имеется пространство имен, определенное следующим образом:

```
namespace my_very_favorite_things { ... };
```

Вы можете использовать имя `mvft` в качестве псевдонима `my_very_favorite_things` следующим образом:

```
namespace mvft = my_very_favorite_things;
```

Можно применить этот метод для того, чтобы упростить вложенные имена:

```
namespace MEF = myth::elements::fire;
using MEF::flame;
```

## Неименованные пространства имен

Вы можете создать *неименованное пространство имен*, не включая имя из пространства имен:

```
namespace // неименованное пространство
 // имен
{
 int ice;
 int bandycoot;
}
```

Результат этих объявлений такой, как если бы за ними следовала директива `using`, т.е. имена, объявленные в этом пространстве имен, находятся в потенциальной области видимости, которая простирается до границы области объявлений, содержащей неименованное пространство имен. В этом отношении их можно рассматривать как глобальные переменные. Тем не менее, поскольку пространство имен не имеет имени, вы не можете явно применить директиву `using` или объявление использования, чтобы сделать доступ к этим именам возможным отовсюду. В частности, невозможно использовать имена из неименованного пространства имен в файле, отличном от того, что содержит объявление пространства имен. Это может служить альтернативой использования внешних статических переменных.

## Пространства имен и будущее

В то время как программисты осваивают такое средство, как пространство имен, сохраняются и привычные при-

емы программирования. В частности, многие надеются, что использование глобального пространства имен со временем сойдет на нет и с помощью механизма пространства имен будут разработаны библиотеки классов. И в самом деле, в текущей версии C++ уже предлагается размещать стандартные библиотечные функции в пространстве имен с именем `std`.

Как уже было отмечено, изменения в именах заголовочных файлов отражают эти перемены. Файлы заголовков старых образцов, такие как `iostream.h`, не нуждаются в использовании пространств имен, однако новый заголовочный файл `iostream` должен пользоваться пространством имен `std`.

## Резюме

Язык C++ расширил возможности функций языка С. Используя ключевое слово `inline` с определением функции и помещая это определение впереди первого обращения к этой функции, вы предлагаете компилятору C++ обращаться с этой функцией как со встроенной. Иначе говоря, вместо того чтобы заставлять управление переходить к отдельному разделу программного кода с целью выполнить эту функцию, компилятор встраивает вместо каждого вызова функции соответствующий код. Этот механизм встраивания должен быть использован только в тех случаях, когда функция сравнительно короткая.

Ссылочная переменная представляет собой некоторую скрытую форму указателя, который позволяет вам создавать псевдоним (второе имя) переменной. Ссылочные переменные используются главным образом как аргументы тех функций, которые производят обработку объектов данных типа структур и классов.

Прототипы C++ предоставляют возможность устанавливать значения аргументов, используемых по умолчанию. Если в обращении к функции пропущен соответствующий аргумент, программа использует его значение, заданное по умолчанию. Если в обращении к функции значение аргумента указано, программа использует его вместо значения, заданного по умолчанию. Аргументы, определенные по умолчанию, могут быть представлены в списке аргументов в только в определенном порядке — справа налево. Таким образом, если вы указываете значение по умолчанию для конкретного аргумента, то при этом должны указать значения по умолчанию для всех других аргументов, находящихся в обращении справа от этого аргумента.

Сигнатурой функции является ее список аргументов. Вы можете определить две функции под одним и тем же именем при условии, что они обладают различными сигнатурами. Это называется полиморфизмом

функций или перегрузкой функций. Как правило, вы перегружаете функцию для того, чтобы выполнить одну и ту же операцию над данными различных типов.

Шаблоны функций автоматизируют процесс перегрузки функций. Вы даете определение функции, используя для этой цели обобщенный тип и конкретный алгоритм, а компилятор генерирует соответствующие определения функций для конкретных типов аргументов, которыми вы пользуетесь в программе.

C++ поощряет разработку программ, использующих одновременно нескольких файлов. Эффективной организационной стратегией в этом отношении является использование заголовочного файла для определения типов пользователя и прототипов для функций, манипулирующих типами пользователя. Используйте отдельный файл для исходных программных кодов, составляющих определения функций. Файл заголовков и файл исходных кодов совместно определяют и реализуют тип данных, определенный пользователем, а также то, как он может быть использован. Функция `main()` и другие функции, пользующиеся услугами этих функций, могут быть помещены в третий файл.

Классы памяти C++ определяют, как долго переменные остаются в памяти и какие части программы имеют к ним доступ (область видимости и связывание). Автоматическими являются те переменные, которые определены в пределах блока, такого как, например, тело функции или блок внутри блока. Они существуют и известны только тогда, когда программа выполняет операторы в блоке, который содержит их определения.

Статические переменные существуют на протяжении всего периода выполнения программы. Переменная, определенная за пределами какой-либо функции, принадлежит внешнему классу памяти. Она известна всем функциям в файле, которые появляются в этом файле вслед за ее определением (диапазон доступа файла). Чтобы другой файл мог пользоваться такой переменной, он должен объявить ее, используя при этом ключевое слово `extern`. Переменная, объявленная вне какой-либо функции, но снабженная ключевым словом `static`, обладает диапазоном доступа файла, но недоступна другим файлам (внутреннее связывание). Переменная, определенная внутри блока, но при этом снабженная ключевым словом `static`, является локальной по отношению к этому блоку, тем не менее, она сохраняет свое значение на всем протяжении выполнения программы.

По умолчанию все функции в C++ обладают внешним классом памяти, так что они могут совместно использоваться сразу несколькими файлами. Однако для встроенных функций и функций, снабженных ключевым словом `static`, характерно внутреннее связывание, их

использование ограничено файлом, который содержит их определения.

Пространства имен предоставляют возможность указывать именованные области памяти, в которых можно объявлять идентификаторы. Они предназначены для снижения вероятности конфликтов имен, что особенно важно для больших программ, использующих программные коды, разработанные различными поставщиками программных продуктов. Идентификаторы в пространствах имен могут стать доступными при использовании оператора определения диапазона доступа благодаря применению объявления использования или директивы `using`.

## Вопросы для повторения

1. Какие функции можно причислить к категории встроенных функций?
2. Предположим, что функция `song()` обладает таким прототипом:  
`void song(char * name, int times);`
  - a. Как модифицировать этот прототип, чтобы значение переменной `times`, заданное по умолчанию, было равно 1?
  - b. Какие изменения вы внесете в определение функции?
  - c. Можете ли вы назначить значение "O, My Papa", принятое по умолчанию, переменной `name`?
3. Создайте перегруженную версию функции `iquote()`, которая отображает аргументы, заключенные в кавычки. Напишите три версии: одну — для аргумента типа `int`, другую — для аргумента типа `double` и третью — для аргумента типа `string`.
4. Имеем такую структуру шаблона:  

```
struct box
{
 char maker[40];
 float height;
 float width;
 float length;
 float volume;
};
```

  - a. Создайте функцию, которая ссылается на структуру `box` как на свой формальный аргумент и отображает значение каждого элемента структуры.
  - b. Создайте функцию, которая ссылается на структуру `box` как на свой формальный аргумент и определяет элемент `volume` как произведение трех других измерений.
5. Предположим, что мы хотели бы получить представленные ниже результаты. Может ли каждый из

них быть получен с помощью аргументов, заданных по умолчанию, путем перегрузки функций, тем и другим способом или можно обойтись без этих средств. Приведите примеры соответствующих прототипов.

- a. функция **mass(density, volume)** возвращает массу объекта, имеющего плотность **density** и объем **volume**, а функция **mass(density)** возвращает массу тела, имеющего плотность **density** и объем 1.0 м<sup>3</sup>. Все величины имеют тип **double**.
- b. В результате вызова функции **repeat(10, "I'm OK")** отображается указанная строка десять раз, в то время как функция **repeat("But you're kind of stupid")** отображает заданную строку пять раз.
- c. В результате вызова функции **verage(3,6)** возвращается среднее значение типа **int** двух аргументов типа **int**, а в результате вызова **average(3.0, 6.0)** возвращается среднее значение типа **double** двух величин типа **double**.
- d. В результате вызова функции **mangle("I'm glad to meet you")** возвращается символ | или указатель на строку "I'm mad to greet you", в зависимости от того, присваиваете ли вы возвращаемое значение переменной **char** или переменной **char\***.
- e. В результате вызова функции **average(3,6)** возвращается среднее значение типа **int** двух аргументов типа **int**, когда оно производится в первом файле, и возвращается среднее значение типа **double** двух аргументов типа **int**, когда он производится во втором файле той же программы.
6. Составьте шаблон функции, которая возвращает больший из двух ее аргументов.
7. Используя шаблон из вопроса 6 и структуру **box** из вопроса 4 определите специализацию, которая получает два аргумента типа **box** и возвращает тот из них, у которого объем больше.
8. Каким классом памяти вы воспользуетесь в следующих ситуациях?
  - a. **homer** — формальный аргумент (параметр) функции.
  - б. Должно быть обеспечено совместное использование переменной **secret** двумя файлами.
  - с. Переменная **topsecret** должна быть использована функцией из одного файла, но должна быть скрыта от других файлов.
  - д. Переменная **beencalled** фиксирует, сколько раз была вызвана функция, которая ее содержит.
9. Проанализируйте, в чем разница между объявлением использования и директивой **using**.

## Упражнения по программированию

1. Создайте функцию, которая, если задан один аргумент, а именно адрес строки, выводит эту строку один раз. Однако если задан второй аргумент типа **int**, не равный нулю, то эта функция выводит строку столько раз, сколько было осуществлено вызовов этой функции к моменту ее данного вызова. (Обратите внимание на тот факт, что количество выводимых строк не равно значению второго аргумента, оно равно числу вызовов функции к моменту последнего вызова.) Согласен, это не слишком полезная функция, но практика заставит вас применить некоторые из методов, рассмотренных в данной главе. Воспользуйтесь этой функцией в простой программе, которая способна показать, как эта функция работает.
2. Структура **CandyBar** содержит три элемента. Первый элемент содержит фирменное название конфеты. Второй элемент содержит вес (который может принимать дробное значение) конфеты, а третий элемент представляет число калорий (целое значение) в конфете. Напишите программу, использующую функцию, у которой в качестве аргументов выступает ссылка на структуру **CandyBar**, указатель на **char**, переменная типа **double** и переменная типа **int** и которая использует три последние величины для задания значений соответствующих элементов структуры. Три последних аргумента по умолчанию должны иметь значения "Millennium Munch" 2.85 и 350. Кроме того, программа должна использовать функцию, в качестве аргумента которой выступает ссылка на **CandyBar** и которая отображает содержимое этой структуры. Используйте спецификатор **const** там, где считаете нужным.
3. Ниже представлен каркас программы. Довершите написание программы, составив соответствующие функции и прототипы. Обратите внимание на то, что в программе должны быть две функции **show()**, каждая из них использует аргументы, заданные по умолчанию. Используйте спецификатор **const** там, где считаете нужным. Обратите также внимание на тот факт, что **set()** должна использовать **new** в целях выделения достаточного пространства памяти для хранения заданной строки. Используемые здесь методы аналогичны методам, применяемым при задании и реализации классов. (Возможно, вам придется поменять имена файлов заголовков и удалить директивы **using**, что зависит от используемого компилятора.)

```

#include <iostream>
using namespace std;
#include <cstring> // для вызова strlen(),
 // strcpy()
struct stringy {
 char * str; // указывает на строку
 int ct; // длина строки (не содержит
 // '\0')
};
// прототипы set(), show(), and show()
// попадают сюда
int main()
{
 stringy beany;
 char testing[] = "Reality isn't what
 it used to be.";
 set(beany, testing); // первым
 // аргументом является ссылка,
 // выделяет пространство для
 // хранения копии testing,
 // использует элемент str строки
 // beany как указатель на новый
 // блок, копирует testing в новый
 // блок и присваивается значение
 // элементу ct строки beany
 show(beany); // печатает элемент
 // типа строка один раз
 show(beany, 2); // печатает элемент
 // типа строка дважды
 testing[0] = 'D';
 testing[1] = 'u';
 show(testing); // печатает строку
 // testing один раз
 show(testing, 3); // печатает строку
 // testing три раза
 show("Done!");
 return 0;
}

```

4. Задан заголовочный файл:

```

// golf.h - вместо ре8-3.cpp
const int Len = 40;
struct golf
{
 char fullname[Len];
 int handicap;
};

// функция требует от пользователя
// сообщить ей имя и гандикап и
// присваивает элементам с введенные
// значения - возвращает 1, если введено
// имя, и 0, если введена пустая строка
int setgolf(golf & g);

// функция вводит в структуру golf
// полученное имя и гандикап переданные
// значения используются как аргументы
// функции
void setgolf(golf & g, const char * name,
int hc);

// функция устанавливает новое значение
// гандикапа
void handicap(golf & g, int hc);

// функция отображает содержимое структуры
// golf
void showgolf(const golf & g);

```

Скомпилируйте несколько программ на базе этого заголовочного файла. Один файл с именем **golf.cpp** должен обеспечить подходящие определения функций, которые будут соответствовать прототипам, содержащимся в заголовке файла. Второй файл должен содержать **main()** и обеспечивать реализацию всех свойств прототипированных функций. Например, цикл должен потребовать ввода массива структур **golf** и прекращать ввод, когда массив будет заполнен, либо когда вместо имени игрока в гольф пользователь вводит пустую строку. Чтобы получить доступ к структуре **golf**, функция **main()** должна использовать только прототипированные функции.

5. Создайте шаблон функции **max5()**, которая использует в качестве своего аргумента массив из пяти элементов типа **T** и возвращает наибольший элемент этого массива. (Поскольку размер массива фиксирован, эту операцию можно выполнять в жестко-заданном цикле, а не использовать соответствующий аргумент.) Проверьте его в программе, которая использует массив, состоящий из 5 значений типа **5 int**, и массив, содержащий 5 значений **double**.
6. Сделайте шаблонную функцию **maxn()**, которая принимает в качестве аргумента массив элементов типа **T** и целое число, представляющее собой количество элементов в массиве, и которая возвращает элемент с наибольшим значением в этом массиве. Проверьте ее в программе, которая использует шаблон данной функции с массивом из шести значений типа **int** и с массивом из четырех значений типа **double**. Программа также должна включать специализацию, которая использует в качестве аргумента массив указателей на **char** и число указателей в качестве второго аргумента и которая возвращает адрес самой длинной строки. Если имеется более одной строки наибольшей длины, функция возвращает адрес первой из них. Выполнить проверку специализации на примере массива, состоящего из пяти указателей на строки.

# Объекты и классы

**В этой главе рассматривается следующее:**

- Процедурное и объектно-ориентированное программирование
- Концепция класса
- Определение и реализация класса
- Общедоступный и приватный доступ к классу
- Элементы данных класса
- Методы класса (функции-элементы класса)
- Создание и использование объектов класса
- Конструкторы и деструкторы класса
- Функции-элементы `const`
- Указатель `this`
- Создание массивов объектов
- Диапазон доступа к классу
- Абстрактные типы данных (ADT)

Для объектно-ориентированного программирования характерен особый концептуальный подход к разработке программ, а язык C++ представляет собой расширение языка С благодаря внедрению новых средств, которые облегчают применение этого подхода. Наиболее важными инструментальными средствами ООП являются следующие:

- Абстрагирование
- Инкапсуляция и сокрытие данных
- Полиморфизм
- Наследование
- Повторное использование программных кодов

Класс представляет собой единственное наиболее важное усовершенствование языка C++, предназначенное для реализации этих свойств и связывания их в единое целое. В этой главе мы начнем рассматривать классы. Здесь вы получите первое представление об абстрагировании, инкапсуляции и сокрытии данных, а также узнаете, как классы реализуют эти свойства. Вы научитесь правильно определить класс, узнаете, что собой представляет класс с общедоступным и приватным разделами, рассмотрите процесс создания функций-элементов, которые выполняют обработку данных класса. Кроме того, в этой главе будут описаны конструкторы и деструкторы, каковыми являются функции-элементы специального вида, предназначенные для создания и удаления объектов, принадлежащих классу. И наконец,

вы научитесь работать с указателем `this`, который является важным компонентом, необходимым для программирования классов. В последующих главах рассматривается перегрузка операторов (еще одна разновидность полиморфизма), а также наследование, которое служит основой для повторного использования программных кодов.

## Процедурное и объектно-ориентированное программирование

Хотя время от времени мы отаем должное перспективе применения в программировании принципов ООП, тем не менее, мы все еще придерживаемся стандартного процедурного подхода, характерного для таких языков, как C, Pascal или BASIC. Рассмотрим пример, который показывает, насколько принципы ООП отличаются от принципов процедурного программирования.

Поскольку вы являетесь самым младшим участником команды "Жанр Джайантс" (Genre Giants — Гиганты жанра) по софтболу, вам поручается ведение статистических данных по играм команды. Естественно, неоценимую помощь в этом оказывает компьютер. Если вы приверженец процедурного программирования, то, скорее всего, применяемый алгоритм будет таким, как описано ниже.

Итак, требуется ввести имя, число раз владения битой, число ударов, средний баттинг (Для тех, кто не следит за соревнованиями по бейсболу или по софтболу, сообщаем, что средний баттинг — это среднее число уда-

ров, деленное на регламентированное количество раз владения битой. Время владения битой заканчивается, когда игрок попадает на базу или когда мяч уходит в аут, однако некоторые события, такие как, например, пробежка, не считаются фактическим владением битой.) Учитываются также другие основные статистические данные по каждому игроку. Ожидается, что компьютер должен существенно облегчить задачу, так что прежде всего заставим его вычислить некоторые из упомянутых выше показателей, такие как, например, средний баттинг. Итак, я хочу, чтобы специальная программа выдавала мне нужные результаты. Что нужно сделать, чтобы реализовать подобный замысел? Я считаю, что необходимо правильно организовать программу и воспользоваться услугами функций. Следовательно, нужно сделать так, чтобы функция `main()` обращалась к соответствующей функции, осуществляющей ввод, к функции, выполняющей основные вычисления, и к функции, отображающей результаты этих вычислений. Постойте-ка, а что произойдет, когда я получу результаты следующей игры? Я не хочу снова начинать "с нуля". Допустим, что можно добавить в программу функцию, которая выполнит корректировку статистических данных. Чуть не забыл, мне, по-видимому, понадобится меню для функции `main()`, чтобы иметь возможность выбрать одну из следующих операций: ввод, вычисления, корректировка и отображение данных. Еще одна проблема — а в каком виде будут представлены данные? Можно использовать один массив строк для хранения имен игроков, другой массив для хранения количества раз владений битой каждым игроком, третий массив для хранения числа ударов и т.д. Нет, так дело не пойдет. Я могу спроектировать структуру, чтобы хранить в ней всю эту информацию для каждого конкретного игрока, а затем использовать массив таких структур для хранения этих данных, касающихся всей команды.

Одним словом, сначала все внимание уделяется процедурам, которым вы намерены следовать, а затем вы думаете о том, как представить данные. (Примечание. В силу того обстоятельства, что эта программа не используется на протяжении всего сезона, вы, скорее всего, захотите иметь возможность сохранять данные в файле и впоследствии считывать их. Однако, поскольку работа с файлами еще не рассматривалась, эту проблему мы пока не будем затрагивать.)

Теперь оценим, какие у нас перспективы, если примерить "проблемно-ориентированную шляпу" (привлекательного полиморфного фасона). В этом случае нужно начать с обдумывания всех вопросов, касающихся данных. Более того, необходимо продумать не только то, как представить данные, но и то, как их использовать.

#### НЕКОТОРЫЕ ДОПОЛНИТЕЛЬНЫЕ СООБРАЖЕНИЯ

Так все-таки что нужно отслеживать? Естественно, необходимо контролировать игроков. Следовательно, мне нужен объект, который давал бы представление об игроке в целом, а не только его средний баттинг или количество раз владений битой. Именно так, это будет мой базовый элемент данных, объект, который представляет имя игрока и статистические данные о нем. Я должен иметь в своем распоряжении методы манипулирования этим объектом. К тому же, согласно моим расчетам, потребуется метод ввода информации в этот объект данных. Какие-то из этих данных, в частности средний баттинг, вычисляет компьютер, следовательно, можно включить методы для выполнения соответствующих вычислений. Программа должна выполнять эти вычисления автоматически, пользователь не должен помнить о том, чтобы вовремя запустить соответствующую программу. Кроме того, будут нужны средства для модификации и отображения информации. Таким образом, пользователь получает в свое распоряжение три способа взаимодействия с данными: инициализация, обновление и отображение. Таким должен быть интерфейс пользователя.

Одним словом, вы создаете для себя представление об объекте таким, каким его видит пользователь, концентрируя свое внимание на данных, необходимых для описания объекта, и на операциях, которые описывают взаимодействие пользователя с этими данными. После того как вы разработаете этот интерфейс, начинаете размышлять о том, как реализовать этот интерфейс и организовать хранение данных. И наконец, вы собираете спроектированную вами программу в единый программный модуль.

## Абстрагирование и классы

Жизнь полна сложных вещей, и один из способов решения возникающих сложных проблем заключается в построении упрощающих абстракций. Каждый из нас представляет собой некое количество атомов. Некоторые исследователи интеллекта готовы заявить, что и наш разум — это совокупность полуавтономных агентов. Но гораздо проще думать о себе как о едином организме. В теории вычислительных систем абстракция является решающим шагом в представлении информации и ее взаимодействии с пользователем. Иначе говоря, вы создаете абстрактное представление самых важных операционных особенностей проблемы и формулируете решение этих проблем, используя одни и те же термины. В примере со статистическими данными по футбольу подобный интерфейс описывает, как пользователь инициализирует, корректирует и отображает данные. От абстракции всего один шаг до типа, определенного пользователем, каковым в C++ является проект класса, который реализует этот интерфейс.

## Что представляет собой тип

Давайте подумаем над тем, из чего состоит тип. Например, что собой представляет недотепа? Если вы придерживаетесь распространенных стереотипов, то можете описать недотепу, используя для этой цели зрительные представления, — эдакий увалень в очках с толстыми стеклами, с набитыми ручками карманами и т.д. Немного поразмышляв, вы, возможно, придет к заключению, что понятие "недотепа" лучше определить, приняв во внимание его способность к действию, например, как он поведет себя в затруднительной ситуации на людях. Мы имеем такую же ситуацию, если не прибегать к помощи далеко идущих аналогий, с процедурными языками типа С. Сначала вы склонны думать о типе данных, опираясь на какие-то внешние признаки, — как они хранятся в памяти. Тип данных **char**, например, занимает один байт памяти, а тип **double** довольно часто занимает восемь байтов памяти. Однако непродолжительные размышления приводят нас к заключению, что тип данных определяется в терминах операций, которые могут быть выполнены над этими данными. Например, тип **int** может быть использован всеми арифметическими операциями. Вы можете складывать, вычитать, умножать и делить целые числа. Вы также можете применять к целым числам операцию деления по модулю (%).

А теперь рассмотрим указатели. Указатель может затребовать не больше памяти, чем тип данных **int**. Он вполне может иметь внутреннее представление в виде целого числа. Однако указатель не допускает выполнения операций, возможных при работе с целыми числами. Вы не можете, например, взять два указателя и перемножить их друг с другом. Такое действие не имеет смысла, поэтому в С++ оно не реализовано. Таким образом, когда вы объявляете переменную как тип **int** или как указатель на переменную типа **float**, вы не только выделяете ей память, но и устанавливаете, какие операции могут быть выполнены над этой переменной. Одним словом, спецификация основного типа позволяет определить следующее:

- Какой объем памяти необходим для размещения соответствующего объекта данных.
- Какие операции или методы могут быть выполнены над объектом данных.

Для встроенных типов данных эта информация заложена в компилятор. Но когда в С++ задается тип, определяемый пользователем, нужно самостоятельно задать эту информацию. В качестве вознаграждения за эту дополнительную работу вы получаете выигрыш в производительности и гибкости, что позволяет создавать новые типы данных, в большей степени отвечающие требованиям приложения.

## Класс

В языке С++ класс представляет собой платформу для перевода абстракции в тип, определяемый пользователем. Он сочетает в себе представление данных с методами упорядочивания этих данных в компактные пакеты. Рассмотрим класс, который представляет пакет акций.

Сначала нужно немного подумать о том, как представить пакет акций. Мы могли бы взять одну акцию из пакета ценных бумаг в качестве базовой единицы и определить класс, представляющий акцию. Однако отсюда следует, что вам потребуется 100 объектов для представления 100 акций, а это нецелесообразно с практической точки зрения. Вместо этого попытаемся представить текущие вклады субъекта в виде конкретного пакета акций, приняв его за базовую единицу. Реалистический подход к решению этой проблемы предусматривает ведение записей о таких событиях, как исходная покупная цена и дата покупки, для отчетности перед налоговыми органами. Кроме того, нужно контролировать такие события, как, например, дробление пакета акций. На первый взгляд это выглядит несколько претенциозно для нашей первой попытки определить класс, в силу этого обстоятельства примем идеализированное упрощенное представление о сути дела. В частности, ограничим перечень операций, которые нам предстоит выполнить, следующим списком:

- Получить пакет в компании.
- Приобрести дополнительные акции того же пакета.
- Продать пакет.
- Корректировать среднюю стоимость одной акции пакета.
- Отображать данные о пакете акций.

Мы можем использовать этот список в целях определения общедоступного интерфейса для класса пакета акций, оставляя реализацию дополнительных свойств тем, кто этим интересуется, в качестве упражнения. Для поддержки этого интерфейса нам нужно хранить некоторые виды информации. И снова мы применим упрощенный подход. В частности, не будем тратить усилия на проведение в соответствие со стандартами США оценки акций с точностью до одной восьмой доллара. Мы будем хранить такую информацию:

- Наименование компании
- Число акций в пакете
- Цена каждой акции
- Общая стоимость всех акций пакета

Далее дадим определения этого класса. В большинстве случаев спецификация класса состоит из двух частей:

- *Объявления класса*, в котором описаны компоненты данных с помощью терминов элементов данных, и общедоступный интерфейс, описанный на языке функций-элементов.
- *Определение методов класса*, которые описывают, как реализуются конкретные функции-элементы данного класса.

Проще говоря, определение класса является общим представлением класса, в то время как определения методов добавляют необходимые подробности.

В программе, представленной в листинге 9.1, приводится объявление экспериментального класса **Stock**. (Чтобы легче можно было отличать классы от других объектов, мы будем следовать вполне обычному, но отнюдь не универсальному соглашению, предусматривающему написание имен классов с заглавной буквы.) Вы убедитесь в том, что оно во многом подобно объявлению структуры, в которую добавлены дополнительные элементы, такие как функции-элементы или общедоступные и приватные разделы. Немного позже мы усовершенствуем это объявление (так что не надо использовать его в качестве модели). Сначала просто посмотрим, как работает данное определение.

#### Листинг 9.1 Первая часть файла stocks.cpp.

```
// начало файла stocks.cpp file
class Stock // объявление класса
{
private:
 char company[30];
 int shares;
 double share_val;
 double total_val;
 void set_tot()
 { total_val = shares * share_val; }
public:
 void acquire(const char * co,
 int n, double pr);
 void buy(int num, double price);
 void sell(int num, double price);
 void update(double price);
 void show();
}; // обратите внимание на точку с запятой в конце
```

Более подробно мы рассмотрим конкретные особенности классов далее, а сначала исследуем более общие свойства. Прежде всего отметим, что ключевое слово **class** в языке C++ идентифицирует этот программный код как код, определяющий конструкцию класса. Синтаксис отождествляет **Stock** с именем типа этого нового класса. Это объявление позволяет нам объявлять пере-

менные, именуемые *объектами* или *экземплярами* типа **Stock**. Каждый отдельный объект представляет собой отдельный вклад. Например, объявления

```
Stock sally;
Stock solly;
```

создают два объекта типа **Stock** с именами **sally** и **solly**. Объект **sally**, например, мог представлять вклады Sally в конкретной компании.

Далее обратите внимание на то обстоятельство, что информация, которую мы решили сохранить, поступает в форме элементов данных класса, таких как **company** и **shares**. Элемент данных **company** объекта **sally**, например, содержит имя компании, элемент данных **share** — число акций, которым владеет Салли, элемент **share\_val** — стоимость каждой акции, а элемент **total\_val** — значение общей стоимости всех акций. То же самое можно сказать и о тех операциях, которые по нашему замыслу должны быть представлены в виде функций-элементов, таких как **sell()** и **update()**. Функции-элементы класса получили название *методов класса*. Вместо их может быть объявлена функция-элемент, такая как, например, **set\_tot()**, либо она может быть представлена прототипом, как и остальные функции этого класса. Полное определение других функций-элементов будет дано далее, однако чтобы описать интерфейсы функций, достаточно и прототипов. Связывание данных и методов в один функциональный модуль — самое замечательное свойство класса. Благодаря такой конструкции объект **Stock** автоматически устанавливает правила, регламентирующие использование этого объекта.

Вы уже видели, как классы **istream** и **ostream** обозначаются функциями-элементами, такими как **get()** и **getline()**. Прототипы функций в объявлении класса **Stock** показывают, каким образом создаются функции-элементы. В заголовочном файле **iostream**, например, имеется прототип функции **getline()** в объявлении класса **istream**.

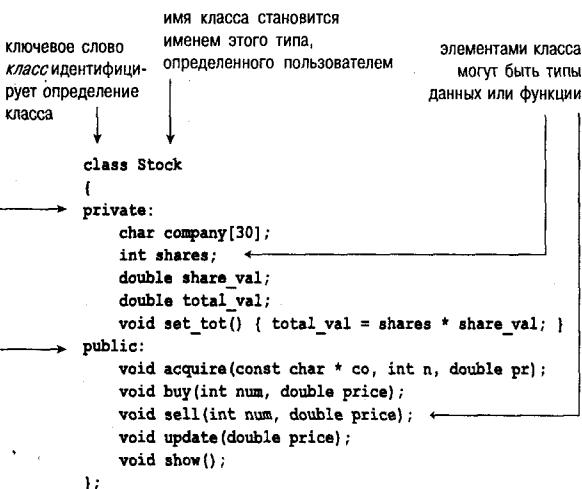
Новыми являются также ключевые слова **private** и **public**. Эти метки описывают *управление доступом* к элементам классов. Любая программа, которая использует объект конкретного класса, может получить непосредственный доступ к общедоступной части класса. Программа может получить доступ к приватным элементам объекта только путем использования общедоступных функций-элементов (или, как вы увидите в главе 10, путем использования дружественной функции). Например, единственный способ, позволяющий внести изменения в элемент **shares** класса **Stock**, заключается в использовании одной из функций-элементов класса **Stock**. Таким образом, общедоступные функции-элементы выступают как посредники между программой и приватными элементами объекта; они обеспечивают интерфейс

между объектом и программой. Такая изоляция данных от непосредственного доступа, реализованная программой, называется *сокрытием данных*. (В языке C++ имеется третье ключевое слово, **protected**, предназначенное для управления доступом, которое мы рассмотрим во время изучения наследования класса в главе 12) (рис. 9.1). Несмотря на то что сокрытие данных может оказаться беспричинным поступком, скажем, с точки зрения перспектив выпуска новых акций, в программировании эта практика вполне себя оправдывает, так как она обеспечивает сохранение целостности данных.

Конструкция класса сделана такой, чтобы отделить общедоступный интерфейс от специфики реализации. Общедоступный интерфейс представляет собой абстрактный компонент этой конструкции. Размещение деталей реализации в одном месте и отделение их от абстракции называется *инкапсуляцией*. *Сокрытие данных* (размещение данных в приватном разделе класса) представляет собой пример инкапсуляции. Таким же примером является сокрытие деталей реализации в приватном разделе, как это делает класс **Stock** с функцией **set\_tot()**. Другим примером инкапсуляции может служить обычная практика размещения определений функций класса в файле, отделенном от объявления классов.

Обратите внимание на тот факт, что сокрытие данных позволяет не только предотвратить прямой доступ к данным, но и освобождает вас от необходимости следить за представлением данных.

ключевое слово *private* идентифицирует элементы класса, доступ к которым можно осуществить путем использования общедоступных функций-элементов (сокрытие данных)



ключевое слово *public* идентифицирует элементы класса, которые составляют общедоступный интерфейс для класса (абстракция)

## ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ И С++

Объектно-ориентированное программирование представляет собой стиль программирования, которым в той или иной степени вы можете воспользоваться при работе в любом языке. Разумеется, вы можете включать многие идеи ООП в обычные программы, написанные на языке С. В частности, в главе 8 приводится пример (листинги 8.12 – 8.14), где заголовочный файл содержит прототип структуры вместе с прототипами функций, предназначенных для манипулирования этой структурой. Следовательно, функция **main()** просто определяет переменные типа этой структуры и использует ассоциированные функции для работы с этими переменными; **main()** не осуществляет непосредственный доступ к элементам структуры. По существу, в этом примере определяется абстрактный тип, который помещает формат памяти и прототипы функций в заголовочный файл, скрывая фактическое представление данных от функции **main()**. В С++, однако, имеются средства, специально предназначенные для реализации подхода ООП. Благодаря этому вы можете продвинуть процесс на несколько шагов дальше, чем это возможно в среде языка С. Во-первых, при размещении представления данных и прототипов функций в одном объявлении класса, а не в файле, можно объединить описание, в этом случае все находится в одном описании класса. Во-вторых, перевод представления данных в разряд приватных приводит к тому, что доступ к данным осуществляется только путем использования функций, получивших на это санкцию. Если, как и в примере программы, написанной на С, функция **main()** осуществляет непосредственный доступ к элементу структуры, он противоречит духу ООП, но при этом не нарушает каких-либо правил языка С. В то же время попытка осуществить прямой доступ, скажем, к элементу **shares** объекта **Stock** не нарушает правил языка С++, а компилятор, тем не менее, зафиксирует ее как ошибку.

Например, функция-элемент **show()** среди многих других данных отображают общую сумму вклада. Эта величина может храниться как часть соответствующего объекта, что имеет место в рассматриваемом примере, либо она может быть при необходимости вычислена. С точки зрения пользователя не имеет никакого значения, какой подход будет использован. Что нужно знать обязательно — так это то, какие действия выполняют функции-элементы. Иначе говоря, требуется знать, какие типы аргументов принимает конкретная функция-элемент и какой тип имеет возвращаемое ею значение, если таковое имеется. Принцип заключается в том, что детали реализации отделяются от конструкции интерфейса. Если в дальнейшем вы найдете более подходящий путь для реализации представления данных или деталей функций-элементов, можно внести изменения в эти детали, не затрагивая программного интерфейса, благодаря чему эти программы становятся более удобными в работе.

### Общедоступный или приватный?

Элементы класса можно объявить независимо от того, являются ли они элементами данных или функциями-

РИСУНОК 9.1 Класс *Stock*.

элементами, и независимо от того, находятся ли они в общедоступном или в приватных разделах класса. Однако одним из основных принципов ООП является скрытие данных, поэтому элементы данных, как правило, размещаются в приватном разделе. Функции-элементы, которые образуют интерфейс класса, размещаются в общедоступном разделе; в противном случае вы не сможете вызвать эти функции из программы. Как показывает объявление класса `Stock`, можно также поместить функции-элементы в приватный раздел. Вы не можете обратиться к этим функциям прямо из программы, однако их могут использовать общедоступные методы. Как правило, приватные функции-элементы используются для манипулирования деталями реализации, которые не являются составной частью публичного интерфейса.

Вовсе не требуется прибегать к помощи ключевого слова `private` в объявлениях классов, поскольку управление доступом к объектам класса, задаваемое этим ключевым словом, принято по умолчанию:

```
class World
{
 float mass; // приватная переменная,
 // заданная по умолчанию
 char name[20]; // приватный массив,
 // заданный по умолчанию
public:
 void tellall(void);
 ...
};
```

Тем не менее, будет явно использоваться метка `private`, чтобы подчеркнуть важность понятия скрытия данных.

### КЛАССЫ И СТРУКТУРЫ

Описания классов во многом совпадают с объявлениями структур, отличие состоит в том, что добавляются функции-элементы и метки `public` и `private`. Фактически C++ распространяет на структуры те же свойства, какими обладают классы. Единственное отличие состоит в том, что типом доступа к структуре по умолчанию будет `public`, в то время как типом доступа к классу по умолчанию будет `private`. Программисты, работающие в среде C++, обычно используют классы для реализации описаний класса, в то время как использование структур ограничивается представлением чистых объектов данных или, время от времени, классов без приватных компонентов.

## Реализация классов и функций-элементов

Перед нами все еще стоит задача выполнения второй части спецификации класса: составить программные коды для тех функций-элементов, которые представлены прототипом в объявлении класса. Рассмотрим следующую проблему. Определения функций-элементов во многом аналогичны определениям обычных функций. Каждое такое определение состоит из заголовка функ-

ции и тела функции. Они могут иметь возвращаемые типы и аргументы. В то же время они обладают двумя специфическими характеристиками:

- Когда вы определяете функцию-элемент, вы используете оператор определения диапазона доступа (`::`) для идентификации класса, которому эта функция принадлежит.
- Методы класса могут осуществлять доступ к компонентам класса типа `private`.

Перейдем к обсуждению этих вопросов.

Прежде всего, в заголовке функции-элемента используется оператор определения диапазона доступа (`::`) с целью указать, какому классу эта функция принадлежит. Например, заголовок функции-элемента `update()` имеет вид:

```
void Stock::update(double price)
```

Такая форма записи означает, что мы определяем функцию `update()`, которая является элементом класса `Stock`. Но эта запись не только идентифицирует функцию `update()` как функцию-элемент, она также означает, что мы имеем возможность воспользоваться этим именем для обозначения функции-элемента другого класса. Например, функция `update()` класса `Buffoon` будет иметь такой заголовок:

```
void Buffoon::update()
```

Таким образом, оператор определения диапазона доступа идентифицирует класс, которому принадлежит определение метода. Мы говорим, что идентификатор `update()` обладает *диапазоном доступа класса*. Другие функции-элементы класса `Stock` могут при необходимости воспользоваться методом `update()` без употребления оператора определения диапазона доступа. Это объясняется тем, что они принадлежат к одному и тому же классу, обеспечивая попадание функции `update()` в диапазон доступа. Использование функции `update()` за пределами областей объявления класса и определений методов требует принятия специальных мер, к рассмотрению которых мы вскоре перейдем.

Одна из особенностей имен методов заключается в том, что полное имя метода конкретного класса содержит в себе имя этого класса. Мы говорим, что `Stock::update()` — это *уточненное имя* функции. Просто `update()`, с другой стороны, является аббревиатурой (*неуточненным именем*) полного имени, как раз это имя может быть использовано в диапазоне доступа класса.

Второй специальной характеристикой методов является тот факт, что метод может получить доступ к приватным элементам класса. Например, метод `show()` вполне может использовать следующий программный код:

```

cout << "Company:" << company
<< "Shares:" << shares << '\n'
<< "Share Price: $" << share_val
<< "Total Worth: $" << total_val << '\n';

```

В этом программном коде `company`, `shares` и т.п. — это приватные элементы данных класса `Stock`. Если вы попытаетесь воспользоваться функциями, не являющимися элементами класса `Stock`, для доступа к этим элементам данных, компилятор намеренно заблокирует вам доступ. (Однако дружественные функции, которые обсуждаются в главе 10, составляют в этом случае приятное исключение.)

Имея в виду эти два соображения, мы сможем обеспечить выполнение методов класса, как показано в листинге 9.2. Определения этих методов могут находиться в отдельном файле или в одном файле с определением класса. Поскольку мы продвигаемся от простого к сложному, предположим, что определения находятся в одном и том же файле с объявлением класса и следуют сразу же за ним. Это самый легкий, но отнюдь не самый лучший способ, позволяющий обеспечить доступ к объявлению класса со стороны определений методов. (Наилучший способ, который мы применим в данной главе несколько позже, состоит в использовании заголовочного файла для хранения объявления класса и использования файла с исходным программным кодом для хранения определений функций-элементов этого класса.)

### Листинг 9.2 Программа stocks.cpp.

```

// more stocks.cpp - реализация
// функций-элементов класса
#include <iostream>
using namespace std;
#include <cstdlib> //или stdlib.h для exit()
#include <cstring> //или string.h для strcpy()
void Stock::acquire(const char * co,
 int n, double pr)
{
 strcpy(company, co, 29); //при необходимости
 //выполняется усечение строки co
 company[29] = '\0';
 shares = n;
 share_val = pr;
 set_tot();
}
void Stock::buy(int num, double price)
{
 shares += num;
 share_val = price;
 set_tot();
}
void Stock::sell(int num, double price)
{
 if (num > shares)
 {
 cerr << "You can't sell more than
 "you have!\n";
 exit(1);
 }

```

```

 shares -= num;
 share_val = price;
 set_tot();
}
void Stock::update(double price)
{
 share_val = price;
 set_tot();
}
void Stock::show()
{
 cout << "Company:" << company
 << "Shares:" << shares << '\n'
 << "Share Price: $" << share_val
 << "Total Worth: $" << total_val
 << '\n';
}

```

### Примечания, касающиеся функций-элементов

Функция `acquire()` выполняет обработку первоначального вклада конкретной компании, в то время как функции `buy()` и `sell()` осуществляют операции по добавлению или снятию сумм с существующего вклада. Если пользователь предпринимает попытку продать больше акций, чем у него есть, функция `sell()` обращается к функции `exit()`, которая прекращает выполнение программы. (Исключения, которые обсуждаются в главе 14, позволяют дать более гибкий ответ.) Четыре функции-элемента устанавливают или переустанавливают значение элемента `total_val`. Вместо того чтобы выполнять эти вычисления четыре раза подряд, класс каждый раз обращается к функции `set_tot()`. Поскольку эта функция является просто средством реализации программного кода, а вовсе не частью общедоступного интерфейса, класс переводит функцию-элемент `set_tot()` в категорию приватных. Если бы вычисления оказались пространными, это позволило бы сэкономить усилия при вводе текста. Однако основное достоинство такого подхода состоит в том, что благодаря осуществлению вызова функции вместо создания вычислительных программных кодов вы каждый раз гарантированно выполняете одни и те же вычисления. Таким образом, если вам нужно внести изменения в вычисления (что маловероятно в рассматриваемом случае), это требуется выполнить только один раз.

Метод `acquire()` использует функцию `strcpy()`, чтобы скопировать строку. На тот случай, если вы забудете выполнить ввод, в результате вызова `strcpy(s2, s1, n)` копируется либо значение `s1` в `s2`, либо `n` символов из `s1` в `s2`, в зависимости от того, что произойдет раньше. Если `s1` содержит меньше `n` символов, то функция `strcpy()` заполняет оставшуюся часть `s2` пустыми символами. Другими словами, при вызове `strcpy(firstname, "Tim", 6)` копируются символы `T`, `i` и `m` в `firstname`, а затем добавляются три пустых символа, чтобы в сум-

ме было шесть символов. Однако, если `s1` имеет большую, чем `n`, длину, нулевые символы не добавляются. Иначе говоря, функция `strcpy(firstname, "Priscilla", 4)` просто копирует символы P, r, i и s в переменную `firstname`, превращая ее в символьный массив, но поскольку заключительный пустой символ отсутствует, ее нельзя рассматривать как строку. Поэтому функция `acquire()` помещает пустой символ в конец массива, чтобы гарантированно создать строку.

### ОБЪЕКТ `CERR` И ФУНКЦИЯ `EXIT()`

Объект `cerr`, как и `cout`, – это объект `ostream`. Различие заключается в том, что переадресация операционной системы затрагивает `cout`, но отнюдь не `cerr`. Объект `cerr` используется для выдачи сообщений об ошибках. Следовательно, если вы перенаправите вывод программы в файл и при этом имеется ошибка, то на экран все равно выводится сообщение об ошибке. Функция `exit()` прекращает выполнение программы. Обычно ненулевые аргументы используются для того, чтобы показать, что программа завершилась нестандартно. Тем не менее, для достижения максимальной степени переносимости в рамках стандарта ANSI вы можете использовать конструкции `EXIT_SUCCESS` и `EXIT_FAILURE` в качестве возвращаемого значения. Вызов функции `exit()` из функции `main()` дает тот же результат, что и вызов `return` из `main()`, но, в отличие от `return`, функция `exit()` завершает выполнение программы независимо от функции, из которой она была вызвана. Заголовочный файл `cstdlib` (ранее `stdlib.h`) содержит прототип этой функции и определяет переносимые возвращаемые значения.

### Встроенные методы

Любая функция с определением в объявлении класса автоматически становится встроенной. Таким образом, `Stock::set_tot()` является встроенной функцией. В объявлениях классах часто используются встроенные функции для небольших функций-элементов, а функция `set_tot()` соответствует этому требованию.

Вы можете, если захотите, определить функцию-элемент вне объявления соответствующего класса и, тем не менее, сделать ее встроенной. Чтобы выполнить это, достаточно воспользоваться спецификатором `inline` во время определения функции в разделе реализации класса:

```
class Stock
{
private:
 ...
 void set_tot(); // определение
 // хранится отдельно
public:
 ...
 inline void Stock::set_tot() // использование
 // спецификатора inline в определении
 {
 total_val = shares * share_val;
 }
}
```

Поскольку встроенные функции обладают внутренним связыванием, они известны только файлу, в котором они объявлены. Простейший способ сделать так, чтобы определения встроенных функций стали доступными для всех файлов в многофайловой программе, состоит во включении определения встроенной функции во все заголовочные файлы, в которых определен соответствующий класс. (В некоторых системах разработки программ имеются эффективные редакторы связей, которые допускают, чтобы определения встроенных функций можно было включать в отдельные файлы реализации.)

Между прочим, согласно *правилу подстановки*, определение метода в объявлении класса эквивалентно замене определения метода прототипом и последующей перезаписи этого определения в качестве встроенной функции непосредственно после объявления класса. Иначе говоря, первоначальное определение функции `set_tot()` эквивалентно определению, которое было рассмотрено выше.

### Выбор объекта

Теперь мы переходим к одному из самых важных аспектов использования объектов: как применить метод классов к объекту. В программном коде

```
shares += num;
```

используется элемент `shares` для одного из объектов. Однако для какого объекта? Это очень интересный вопрос! Чтобы получить на него ответ, рассмотрим сначала, как создаются объекты. Простейший путь состоит в объявлении переменных класса:

```
Stock kate, joe;
```

Этот программный код создает два объекта класса `Stock`, один с именем `kate`, а другой с именем `joe`.

Далее рассмотрим, как использовать функции-элементы для работы с одним из этих объектов. Ответ, как и в случае со структурами и элементами структур, следует искать в использовании оператора принадлежности:

```
kate.show(); // объект kate обращается
 // к функции-элементу
joe.show(); // объект joe обращается
 // к функции-элементу
```

В результате первого вызова начинает выполняться функция `show()` как элемент объекта `kate`. Это означает, что этот метод интерпретирует `shares` как `kate.shares` и `share_val` – как `kate.share_val`. Аналогично, вызов `joe.show()` заставляет метод `show()` интерпретировать `shares` и `share_val` соответственно как `joe.shares` и `joe.share_val`.

## ПОМНИТЕ

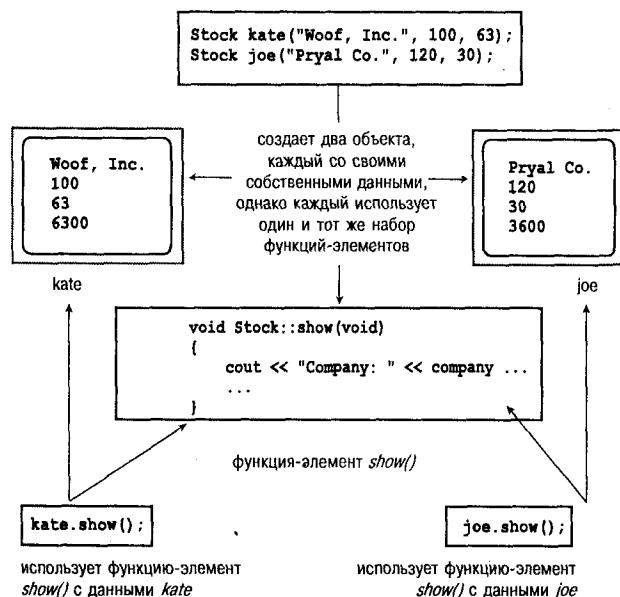
Когда вы обращаетесь к функции-элементу, она использует элементы данных конкретного объекта, применяемого для вызова функции-элемента.

Аналогично вызов функции `kate.sell()` приводит к вызову функции `set_tot()`, как если бы это была функция `kate.set_tot()`, передавая этой функции нужные ей данные из объекта `kate`.

Каждый новый создаваемый объект включает область памяти для хранения собственных внутренних переменных, элементов класса. Однако все объекты одного класса совместно используют одни и те же методы класса, при этом существует одна копия каждого метода. Предположим, например, что `kate` и `joe` — это объекты класса `Stock`. Далее, `kate.shares` занимает один фрагмент памяти, а `joe.shares` — другой фрагмент памяти. Но в то же время как `kate.show()`, так и `joe.show()` вызывают один и тот же метод, т.е. оба выполняют один и тот же блок программных кодов. Они всего лишь применяют этот код к различным данным. Вызов функции-элемента реализует то, что в некоторых языках ООП называется *отправкой сообщения*. Таким образом, при отправке одного и того же сообщения двум различным объектам вызывается один и тот же метод, но применяется он к двум различным объектам (рис. 9.2).

## Использование классов

Вы уже знаете, как определять класс и методы класса. Следующий шаг состоит в написании программы, которая создает и использует объекты соответствующего класса.



В задачу C++ входит использование классов, которые максимально используют встроенные типы, такие как `int` и `char`. Вы можете создать объект класса посредством объявления переменной класса или с помощью спецификатора `new` таким образом, чтобы построить объект типа класс. Можно передавать объекты как аргументы, возвращать их как возвращаемые значения функции и присваивать один объект другому. В C++ имеются средства для инициализации объектов, средства обучения `cin` и `cout`, обеспечивающие распознавание объектов, и даже средства автоматического преобразования типов между объектами подобных классов. Вы еще не научились пользоваться всеми этими средствами, поэтому начнем с наиболее простых из них. В самом деле, вы уже видели, как объявить объект класса и вызвать функцию-элемент. Эти приемы в сочетании с объявлением класса и объявлениями функций-элементов составляют законченную программу, представленную в листинге 9.3. Она создает объект класса `Stock`, которому присвоено имя `stock1`. Программа довольно проста, однако она позволяет проверить все свойства, которыми мы наделили этот класс.

### Листинг 9.3 Полный текст программы stocks.cpp.

```
// stocks.cpp - завершенная программа
#include <iostream>
using namespace std;
#include <cstdlib> // или stdlib.h для
 // функции exit()
#include <cstring> // или string.h для
 // функции strcpy()
class Stock
{
private:
 char company[30];
 int shares;
 double share_val;
 double total_val;
 void set_tot()
 {
 total_val = shares * share_val;
 }
public:
 void acquire(const char * co,
 int n, double pr);
 void buy(int num, double price);
 void sell(int num, double price);
 void update(double price);
 void show();
};

void Stock::acquire(const char * co,
 int n, double pr)
{
 strcpy(company, co); //при необходимости
 //выполняется усечение строки со
 company[29] = '\0';
 shares = n;
 share_val = pr;
 set_tot();
}
```

РИСУНОК 9.2 Объекты, данные и функции-элементы.

```

void Stock::buy(int num, double price)
{
 shares += num;
 share_val = price;
 set_tot();
}
void Stock::sell(int num, double price)
{
 if (num > shares)
 {
 cerr << "You can't sell more
 than you have!\n";
 exit(1);
 }
 shares -= num;
 share_val = price;
 set_tot();
}
void Stock::update(double price)
{
 share_val = price;
 set_tot();
}
void Stock::show()
{
 cout << "Company:" << company
 << "Shares:" << shares << '\n'
 << "Share Price: $" << share_val
 << "Total Worth: $" << total_val
 << '\n';
}
int main()
{
 Stock stock1;
 stock1.acquire("NanoSmart", 20, 12.50);
 cout.precision(2); //формат #.##
 cout.setf(ios_base::fixed); //формат #.##
 cout.setf(ios_base::showpoint); //формат #.##
 stock1.show();
 stock1.buy(15, 18.25);
 stock1.show();
 return 0;
}

```

Данная программа использует три команды форматирования. Результатом выполнения этих команд является отображение двух цифр справа от десятичной точки, включая нули в младших разрядах. Фактически в условиях сложившейся практики необходимы только первые две команды, а в более ранних реализациях достаточно было только первой и третьей команд. Использование всех трех команд форматирования дает один и тот же результат в обеих реализациях. За подробностями обращайтесь к главе 16. А тем временем предлагаем выходные данные программы:

```

Company: NanoSmart Shares: 20
Share Price: $12.50 Total Worth: $250.00
Company: NanoSmart Shares: 35
Share Price: $18.25 Total Worth: $638.75

```

Обратите внимание на то обстоятельство, что функция **main()** представляет собой всего лишь платформу для тестирования конструкции класса **Stock**. При условии, что класс функционирует так, как требуется, мы можем сейчас использовать класс **Stock** как тип, определенный пользователем, в других программах. Важное значение при использовании новых типов имеет понимание того, что делают функции-элементы, при этом нет нужды беспокоиться о том, как реализовать ту или иную деталь. Для справки обратитесь к примечанию.

### МОДЕЛЬ КЛИЕНТ-СЕРВЕР

Программисты, работающие с ООП, часто обсуждают структуру программы, используя при этом терминологию модели "клиент-сервер". При таком представлении на концептуальном уровне клиентом является программа, которая использует соответствующий класс. Объявление класса, включая и методы класса, составляют сервер, который представляет собой ресурс, доступный программе, которая в нем нуждается. Клиент использует сервер только с помощью интерфейса, определенного как общедоступный. Это означает, что единственной обязанностью клиента или программиста, создававшего программу-клиент, является необходимость досконального изучения этого интерфейса. Обязанностью сервера или разработчика сервера является необходимость следить за тем, чтобы сервер надежно и точно выполнял операции, предусматриваемые этим интерфейсом. Любые изменения, которые разработчик сервера вносит в проект класса, должны касаться только деталей реализации, но не интерфейса. Это позволяет программистам совершенствовать клиентское и серверное программное обеспечение независимо друг от друга, не опасаясь того, что изменения, внесенные в сервер, приведут к непредсказуемым последствиям в поведении программы-клиента.

### Текущее состояние дел

Первым шагом в определении конструкции класса является объявление класса. Объявление класса формируется по образцу объявления структуры и может содержать элементы данных и функции-элементы. В таком объявлении имеется приватный раздел, а доступ к элементам, объявленным в этом разделе, возможен только при использовании соответствующих функций-элементов. В объявлении имеется также общедоступный раздел, к которому разрешен доступ непосредственно из программы с использованием объектов класса. Как правило, элементы данных входят в приватный раздел, а функции-элементы попадают в общедоступный раздел, таким образом, объявление класса имеет такой вид:

```

class ИмяКласса
{
private:
 объявления элементов данных
public:
 прототипы функций-элементов
};

```

Содержимое общедоступного раздела составляют абстрактную часть конструкции, т.е. общедоступный интерфейс. Инкапсуляция данных в приватном разделе сохраняет целостность данных и называется сокрытием данных. Итак, класс в C++ позволяет достичнуть целей ООП — речь идет об абстрации, сокрытии данных и инкапсуляции.

Вторым шагом в определении конструкции класса является реализация функций-элементов этого класса. Вы можете воспользоваться полным определением функции вместо прототипа функции в объявлении класса, однако обычная практика, за исключением небольших функций, состоит в использовании отдельных определений функций. В таком случае вам нужно применять оператор определения диапазона доступа, чтобы показать, какому классу принадлежит функция-элемент. Например, предположим, что в классе **Bozo** имеется функция-элемент с именем **Retort()**, которая возвращает указатель на **char**. Заголовок функции будет выглядеть следующим образом:

```
char * Bozo::Retort();
```

Другими словами, **Retort()** — это не просто функция типа **char \***, но функция типа **char \***, которая принадлежит классу **Bozo**. Полное или, иначе говоря, уточненное имя функции — **Bozo::Retort()**. С другой стороны, имя **Retort()** — это аббревиатура уточненного имени, она может быть использована только в особых случаях, например, в программном коде метода класса. Чтобы дать другое описание подобного рода ситуации, достаточно сказать, что имя **Retort** включает диапазон доступа класса, следовательно, необходим оператор определения диапазона доступа, чтобы уточнить это имя, если оно используется вне объявления класса и метода класса.

Чтобы создать объект, которым является конкретный образец или экземпляр класса, используйте имя класса, как если бы это имя было именем типа:

```
Bozo bozetta;
```

Это вполне возможно, поскольку класс является типом, объявленным пользователем.

Функция-элемент класса или метод может быть вызван только объектом класса. Вы можете сделать это, воспользовавшись точкой или оператором принадлежности:

```
cout << Bozetta.Retort();
```

Это выражение вызывает функцию-элемент **Retort()**, но всякий раз, когда программный код этой функции обращается к конкретному элементу данных, функция использует значение, которое этот элемент имеет в объекте **bozetta**.

## Деструкторы и конструкторы классов

Между тем, с классом **Stock** нам еще придется поработать. Существуют специальные функции, получившие названия *конструкторов* и *деструкторов*, которые обычно резервируются за каждым классом. Посмотрим, для чего они нужны и как их создавать.

Одна из целей C++ заключается в том, чтобы сделать использование объектов класса таким же простым, как и использование стандартных типов. Тем не менее, вы не можете инициализировать объект **Stock** тем же способом, что и обычные типы **int** или **struct**:

```
int year = 2001; // правильно
struct thing
{
 char * pn;
 int m;
};
thing amabob = {"wodget", -23}; //правильно
Stock hot = { "Sukie's Autos, Inc.", // неверно!
 200, 50.25} ;
```

Причина, по которой вы не можете инициализировать объект **Stock** таким способом, заключается в том, что разделы данных имеют статус приватных, что означает, что программа не может получить прямой доступ к элементам данных. Как вы уже могли убедиться, единственный способ, благодаря которому программа может получить доступ к элементам данных, — это использование функций-элементов. В связи с этим необходимо разработать соответствующую функцию-элемент, если вы намерены выполнить инициализацию объекта. (Вы можете инициализировать объект класса, как показано ранее, если определите элементы данных как общедоступные, а не как приватные, но определение общедоступных данных противоречит одной из основных целей применения класса — сокрытию данных.)

В общем случае лучше всего, когда все объекты инициализируются во время их создания. Например, рассмотрим следующий программный код:

```
Stock gift;
gift.buy(10, 24.75);
```

В рассматриваемой реализации класса **Stock** объект **gift** не имеет значения для элемента **company**. Конструкция класса предполагает, что вызовет функцию **acquire()**, прежде чем обращаться к другим функциям-элементам, однако каких-либо специальных средств для реализации этого предположения не существует. Один из способов, позволяющих обойти эту трудность, заключается в том, чтобы объекты были инициализированы автоматически в момент их создания. Чтобы осуществить эту идею, предусмотрены специальные функции-элементы, называемые *конструкторами классов*, в задачу которых вхо-

дит построение новых объектов и присвоение элементам данных этих объектов начальных значений. Точнее, C++ назначает имена этим функциям-элементам и предлагает синтаксис для их использования, а вы обеспечиваете метод их определения. Это имя совпадает с именем класса. Например, потенциальным конструктором класса **Stock** является функция-элемент с именем **Stock()**. Прототип конструктора и заголовок обладают весьма интересным свойством — несмотря на то что у конструктора нет возвращаемого значения, она не объявляется как тип **void**. По существу, у конструктора нет объявленного типа.

## Объявление и определение конструкторов

А теперь создадим конструктор **Stock**. Поскольку предусмотрено, что объект **Stock** получает три значения из внешнего мира, необходимо назначать этому конструктору три аргумента. (Четвертое значение, элемент **total\_val**, вычисляется по значениям элементов **shares** и **share\_val**, поэтому не требуется передавать его конструктору.) Вполне возможно, что вы намерены всего лишь задать значение для элемента **company**, а другим элементам присвоить нулевое значение. Это можно сделать с помощью аргументов, заданных по умолчанию (см. главу 8). Следовательно, прототип будет иметь следующий вид:

```
// прототип конструктора с некоторыми
// аргументами, заданными по умолчанию
Stock(const char * co, int n = 0,
 double pr = 0.0);
```

Первый аргумент является указателем на строку, которая используется для инициализации элементов класса символьного массива **company**. Аргументы **n** и **pr** передают значения элементам **shares** и **share\_val**. Обратите внимание на то, что возвращаемый тип отсутствует. Прототип находится в общедоступном разделе объявления класса.

Ниже представлено возможное определение конструктора:

```
// определение конструктора
Stock::Stock(const char * co,
 int n, double pr)
{
 strncpy(company, co, 29);
 company[29] = '\0';
 shares = n;
 share_val = pr;
 set_tot();
}
```

Это тот же программный код, который мы использовали в функции **acquire()**. Различие заключается в том, что программа автоматически вызывает конструктор в тот момент, когда она объявляет объект.



### ПРЕДОСТЕРЖЕНИЕ

Достаточно часто эти новые конструкторы предпринимают попытки использовать имена элементов класса в качестве аргументов конструктора:

```
Stock::Stock(const char * company, int shares,
 double share_val) // неправильно!
{
 ...
}
```

Так делать нельзя. Аргументы конструктора не представляют собой элементов класса, они представляют собой значения, которые присваиваются элементам класса. Следовательно, их имена должны отличаться друг от друга.

## Использование конструктора

В C++ предусмотрены два способа инициализации объекта с использованием конструктора. Первый из них — это явный вызов конструктора:

```
Stock food = Stock("World Cabbage", 250, 1.25);
```

С помощью этой команды элементу **company** объекта **food** присваивается строка **"World Cabbage"**, элементу **shares** — значение **250** и т.д.

Второй способ предусматривает неявный вызов конструктора:

```
Stock garment("Furry Mason", 50, 2.5);
```

Более компактная форма вызова эквивалентна следующему явному обращению:

```
Stock garment = Stock("Furry Mason", 50, 2.5);
```

C++ использует конструктор класса всякий раз, когда создается объект этого класса, даже если вы используете спецификатор **new** в целях динамического распределения памяти. Конструктор со спецификатором **new** используется следующим образом:

```
Stock *pstock = new Stock("Electroshock Games",
 18, 19.0);
```

Этот оператор создает объект **Stock**, инициализирует его значениями, переданными с помощью аргументов, и присваивает адрес объекта указателю **pstock**. В этом случае у объекта нет имени, однако можно воспользоваться указателем при работе с объектом. Мы отложим обсуждение указателей до главы 10.

Использование конструкторов отличается от применения других методов класса. Обычно объект используется для вызова метода:

```
stock1.show(); // объект stock1 вызывает
 // метод show()
```

Тем не менее, вы не можете воспользоваться объектом, чтобы вызвать конструктор, так как до тех пор, пока конструктор не закончит работу по построению конкрет-

ногого объекта, такого объекта не существует. Конструктор не может быть вызван объектом, конструктор используется для создания объекта.

## Конструктор, заданный по умолчанию

*Конструктором, заданным по умолчанию*, является конструктор, используемый для построения объекта, когда явные значения для инициализации отсутствуют. Другими словами, конструктор используется для объявлений такого рода:

```
Stock stock1; // используется конструктор,
// заданный по умолчанию
```

Но позвольте, программа, представленная в листинге 9.3, уже делала это! Причина того, что этот оператор все-таки работает, заключается в том, что, если нет никаких конструкторов, C++ автоматически использует конструкторы, заданные по умолчанию. Таким конструктором является версия по умолчанию конструктора, заданного по умолчанию, и она не выполняет никаких действий. Для класса **Stock** он будет выглядеть следующим образом:

```
Stock::Stock() { }
```

Окончательный результат состоит в том, что объект **stock1** создается без инициализации его элементов, точно так же как оператор

```
int x;
```

создает переменную **x**, не присваивая ей значения. То обстоятельство, что конструктор, заданный по умолчанию, не имеет аргументов, отражает тот факт, что в объявлении не появляются никакие значения.

Интересно отметить, что компилятор предоставляет конструктор, заданный по умолчанию, только в том случае, если вы не определите никакого конструктора. После того как вы назначите конкретный конструктор конкретному классу, обязанность по предоставлению конструктора, заданного по умолчанию, переходит от компилятора к вам. Если вы воспользуетесь конструктором, который не используется по умолчанию, таким как, например, **Stock(const char \* co, int n, double pr)**, и не предложите своей собственной версии конструктора, заданного по умолчанию, то объявление вида

```
Stock stock1; // невозможно с текущим
// конструктором
```

вызовет ошибку. Причина такого поведения заключается в том, что у вас может возникнуть необходимость сделать невозможным создание неинициализированных объектов. С другой стороны, вам может понадобиться создавать объекты без явной инициализации. В таком случае придется определить собственный конструктор.

Это должен быть конструктор, которому не нужны аргументы. Вы можете определить конструктор по умолчанию двумя способами. Один из них состоит в том, чтобы присвоить значения, заданные по умолчанию, всем аргументам существующего конструктора:

```
Stock(const char * co = "Error",
 int n = 0, double pr = 0.0);
```

Второй способ предусматривает использование перегрузки функции для определения второго конструктора, который при этом не имеет аргументов:

```
Stock();
```

(В более ранних версиях C++ можно было воспользоваться только вторым методом построения конструктора, заданного по умолчанию.)

На практике обычно требуется инициализировать объекты, чтобы быть уверенным, что все элементы начинаются с известных, корректно выбранных значений. Таким образом, конструктор, заданный по умолчанию, как правило, осуществляет неявную инициализацию значений всех элементов. В данном случае, например, можно определить конструктор для класса **Stock** следующим образом:

```
Stock::Stock()
{
 strcpy(company, "no name");
 shares = 0;
 share_val = 0.0;
 total_val = 0.0;
}
```

### СОВЕТ

Когда проектируется класс, обычно требуется задать конструктор по умолчанию, который неявно инициализирует все элементы класса.

После того как вы воспользовались одним из методов (не указывая аргументов или значений, заданных по умолчанию для всех аргументов), чтобы создать конструктор по умолчанию, можно объявить переменные объекта, не выполняя их явной инициализации:

```
// неявно вызывает конструктор,
// заданный по умолчанию
Stock first;

// вызывает его явно
Stock first = Stock();

// вызывает его неявно
Stock *prelief = new Stock;
```

Тем не менее, не дайте ввести себя в заблуждение неявной формой конструктора, не используемого по умолчанию:

```
// вызывает конструктор
Stock first("Concrete Conglomerate");
```

```
// объявляет функцию
Stock second();

// вызывает конструктор,
// заданный по умолчанию
Stock third;
```

Первое объявление вызывает конструктор, который не является конструктором, заданным по умолчанию, т.е. конструктор, который принимает аргументы. Второе объявление утверждает, что `second()` — функция, которая возвращает объект `Stock`. При неявном вызове конструктора, заданного по умолчанию, не употребляйте круглые скобки.

## Деструкторы

Когда вы используете конструктор для построения объекта, программа берет на себя обязанность отслеживать этот объект до тех пор, пока он не выполнит возложенную на него задачу. В этот момент программа автоматически вызывает специальную функцию-элемент, которая называется *деструктор*. Деструктор должен уничтожить весь оставшийся "мусор", таким образом, он служит конструктивным целям. Например, если ваш конструктор использует спецификатор `new` при распределении памяти, деструктор с помощью оператора `delete` освобождает память. Конструктор `Stock` не делает ничего экстравагантного, например, не использует спецификатор `new`, поэтому ему фактически не нужен деструктор. Однако совсем неплохо, чтобы такая функция была, по крайней мере, для выполнения последующих проверок класса.

Как и конструктор, деструктор имеет специальное имя: имя класса, которому предшествует тильда (~). Таким образом, деструктор класса `Stock` имеет имя `~Stock()`. Итак, подобно конструктору, деструктор не имеет возвращаемого значения и объявленного типа. В отличие от конструктора, у деструктора не может быть аргументов. Следовательно, прототип деструктора класса `Stock` должен быть таким:

```
~Stock();
```

Поскольку у деструктора `Stock` нет особо важных обязанностей, мы можем закодировать его как функцию, которая не выполняет никаких действий.

```
Stock::~Stock()
{
}
```

Однако только для того, чтобы вы могли увидеть, когда производится обращение к деструктору, запишем его в таком виде:

```
Stock::~Stock() // деструктор класса
{
 cout << "Bye, " << company << "\n";
}
```

Когда следует обращаться к деструктору? Это решение принимает компилятор, ваш программный код не должен содержать явных обращений к деструктору. Если вы создаете объект класса статической памяти, то его деструктор вызывается автоматически в момент окончания выполнения программы. Если вы создаете объект класса автоматической памяти, что, собственно говоря, мы и делали раньше, то его деструктор вызывается автоматически, когда программа выходит из блока программного кода, в котором объект был определен. Если объект создается с использованием спецификатора `new`, он размещается в динамически распределяемой области памяти или в свободной памяти, а его деструктор вызывается автоматически, когда используется оператор `delete` для освобождения памяти. И наконец, программа может создавать временные объекты для того, чтобы выполнять определенные операции; в этом случае программа автоматически вызывает деструктор, чтобы тот удалил объект, когда программа прекращает использование этого объекта.

Поскольку деструктор вызывается автоматически, когда объект класса прекращает функционировать, деструктор должен быть наготове. Если вы не позаботились о своем деструкторе, компилятор предоставит вам деструктор, заданный по умолчанию, который не выполняет никаких действий.

## Совершенствование класса Stock

Следующий шаг состоит во включении конструкторов и деструкторов в определения классов и методов. На этот раз мы будем придерживаться обычной практики, сложившейся в C++, и организуем программу в виде нескольких отдельных файлов. Мы поместим описание рассматриваемого класса в заголовочный файл с именем `stock1.h`. (Как следует из этого имени, мы имеем в виду, что в будущем появятся следующие версии.) Методы класса помещаются в файл с именем `stock1.cpp`. В общем, заголовочный файл, содержащий объявление класса, и файл исходного программного кода, содержащий определения методов, должны иметь одно и то же базовое имя, чтобы можно было отслеживать, какие файлы принадлежат друг другу. Использование отдельных файлов для объявления класса и для функций-элементов отделяет абстрактное определение интерфейса (объявление класса) от деталей реализации (определения функций-элементов). Вы, например, можете реализовать объявление класса как заголовочный текстовый файл, а определения функций реализовывать как скомпилированные коды. И наконец, используя эти ресурсы, мы помещаем программу в третий файл, который называется `usestock1.cpp`.

## Заголовочный файл

В листинге 9.4 показан заголовочный файл. Из него прототипы конструкторов и деструкторов помещаются в объявление исходного класса. Помимо этого, он не включает функцию `acquire()`, которая теперь, когда у класса есть конструкторы, больше не нужна.

### Листинг 9.4 Программа stock1.h.

```
// stock1.h
#ifndef _STOCK1_H_
#define _STOCK1_H_
class Stock
{
private:
 char company[30];
 int shares;
 double share_val;
 double total_val;
 void set_tot()
 { total_val = shares * share_val; }
public:
 Stock(); // конструктор по умолчанию
 Stock(const char * co,
 int n = 0, double pr = 0.0);
 ~Stock(); // деструктор noisey
 void buy(int num, double price);
 void sell(int num, double price);
 void update(double price);
 void show();
};
#endif
```

## УПРАВЛЕНИЕ ЗАГОЛОВОЧНЫМИ ФАЙЛАМИ

Заголовочный файл следует включать в тот или иной файл только один раз. Это, по-видимому, нетрудно запомнить, однако вполне может случиться так, что в файле окажется сразу несколько заголовочных файлов, а вы об этом даже не будете знать. Например, вы можете использовать заголовочный файл, который сам содержит другой заголовочный файл. Имеется стандартный метод C/C++, который позволяет избежать нескольких включений заголовочных файлов. В его основу положена директива `#ifndef` (аббревиатура от слов `if not defined` – если не определен) препроцессора. Сегмент кода вида

```
#ifndef _STOCK1_H_
...
#endif
```

означает: выполнить операторы, заключенные между `#ifndef` и `#endif`, если только имя `_STOCK1_H_` не было определено раньше с помощью директивы `#define` препроцессора.

Обычно оператор `#define` используется для построения символьных констант, как, например, в следующем случае:

```
#define MAXIMUM 4096
```

Однако простое использование оператора `#define` с именем достаточно разве что для того, чтобы установить, что имя определено:

```
#define _STOCK1_H_
```

Метод, который использует программа, представленная в листинге 9.4, предназначен для того, чтобы удалить содержимое этого файла в `#ifndef`:

```
#ifndef _STOCK1_H_
#define _STOCK1_H_
// поместить содержимое файла включения
// в этом месте
#endif
```

Первый раз, когда компилятор встречает этот файл, имя `_STOCK1_H_` не должно быть определенным. Мы выбираем имя, опираясь на имя файла включения с несколькими символами подчеркивания, вставленными таким образом, чтобы это имя не могло совпасть с именем, определенным где-нибудь в другом месте. Если такое случается, то компилятор просматривает, что содержится между `#ifndef` и `#endif`, а это как раз то, что нам нужно. В процессе этого просмотра компилятор читает строку, определяющую `_STOCK1_H_`. Если компилятор затем обнаруживает еще одно включение файла `stock1.h` в одном и том же файле, он отмечает, что имя `_STOCK1_H_` определено, и переходит к строке, которая следует за `#endif`. Обратите внимание на то, что этот метод не препятствует компилятору включать файл дважды. Вместо этого он вынуждает компилятор игнорировать содержимое всех последующих включений, кроме первого. Большая часть заголовочных файлов в стандартном C и C++ используют эту схему.

## Файл реализации

В листинге 9.5 представлены определения методов. Здесь размещается файл `stock1.h`, реализующий условия для объявления класса. (Напомним, что заключение имени файла в двойные кавычки вместо скобок означает, что компилятор осуществляет его поиск в том месте, где были обнаружены исходные файлы.) Кроме того, в программе, представленной в этом листинге, используются системные файлы `iostream` и `cstring`, поскольку указанные выше методы используют `cin`, `cout` и `strncpy()`. Этот файл добавляет определения методов конструктора и деструктора в уже имеющиеся методы.

### Листинг 9.5 Программа stock1.cpp.

```
// stock1.cpp - методы класса Stock
#include <iostream>
#include <cstdlib> //или stdlib.h для exit()
#include <cstring> //или string.h для strncpy()
using namespace std;
#include "stock1.h"
// конструкторы
Stock::Stock() // конструктор по умолчанию
{
 strcpy(company, "no name");
 shares = 0;
 share_val = 0.0;
 total_val = 0.0;
}
Stock::Stock(const char * co, int n, double pr)
{
```

```

strncpy(company, co, 29);
company[29] = '\0';
shares = n;
share_val = pr;
set_tot();
}

// деструктор класса
Stock::~Stock() // деструктор класса verbose
{
 cout << "Bye, " << company << "!\n";
}

// другие методы
void Stock::buy(int num, double price)
{
 shares += num;
 share_val = price;
 set_tot();
}

void Stock::sell(int num, double price)
{
 if (num > shares)
 {
 cerr << "You can't sell more than
 you have!\n";
 exit(1);
 }
 shares -= num;
 share_val = price;
 set_tot();
}

void Stock::update(double price)
{
 share_val = price;
 set_tot();
}

void Stock::show()
{
 cout << "Company: " << company
 << "Shares: " << shares << '\n'
 << "Share Price: $" << share_val
 << "Total Worth: $" << total_val
 << '\n';
}

```

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Вы могли бы использовать файлы `stdlib.h` и `string.h` вместо `cstdlib` и `cstring`.

#### Клиентский файл

В программу, представленную в листинге 9.6 включен небольшой модуль для тестирования новых методов. Подобно программе `stock1.cpp`, она содержит файл `stock1.h`, посредством которого осуществляется объявление классов. Эта программа демонстрирует конструкторы и деструкторы. Она также использует те же команды форматирования, которые выполнялись в программе, представленной в листинге 9.3. Чтобы скомпилировать программу в полном составе, воспользуйтесь методом, ориентированным на многофайловые программы (см. главы 1 и 8).

#### Листинг 9.6 Программа `usestok1.cpp`.

```

// usestok1.cpp - использование класса Stock
#include <iostream>
using namespace std;
#include "stock1.h"

int main()
{
 // использование конструкторов для
 // построения новых объектов
 Stock stock1("NanoSmart", 12, 20.0); // синтаксис 1
 Stock stock2 = Stock ("Boffo Objects", 2, 2.0); // синтаксис 2
 cout.precision(2); // формат #.##
 cout.setf(ios_base::fixed,
 ios_base::floatfield); // формат #.##
 cout.setf(ios_base::showpoint); //формат #.##
 stock1.show();
 stock2.show();
 stock2 = stock1; // назначение объекта

 // использование конструктора для
 // переустановки объекта
 stock1 = Stock("Nifty Foods", 10, 50.0); // объект temp
 cout << "After stock reshuffle:\n";
 stock1.show();
 stock2.show();
 return 0;
}

```

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Вы могли бы воспользоваться более старой версией `ios::` вместо `ios_base::`.

Результаты выполнения программы:

```

Company: NanoSmart Shares: 12
Share Price: $20.00 Total Worth: $240.00
Company: Boffo Objects Shares: 2
Share Price: $2.00 Total Worth: $4.00
Bye, Nifty Foods!
After stock reshuffle:
Company: Nifty Foods Shares: 10
Share Price: $50.00 Total Worth: $500.00
Company: NanoSmart Shares: 12
Share Price: $20.00 Total Worth: $240.00
Bye, NanoSmart!
Bye, Nifty Foods!

```

#### Примечания к программе

Оператор

`Stock stock1("NanoSmart", 12, 20.0);`

создает объект класса `Stock` под именем `stock1` и инициализирует его элементы данных заданными значениями. Оператор

`Stock stock2 = Stock ("Boffo Objects", 2, 2.0);`

использует вторую разновидность синтаксиса для построения и инициализации объекта с именем `stock2`.

Вы можете использовать конструктор не только для инициализации нового объекта. Например, в составе функции `main()` имеется такой оператор:

```
stock1 = Stock("Nifty Foods", 10, 50.0);
```

Объект `stock1` уже существует. Таким образом, вместо того чтобы инициализировать объект `stock1`, этот оператор присваивает новые значения данному объекту. Он выполняет это, вынуждая конструктор построить новый, временный объект и в дальнейшем осуществить копирование содержимого этого нового объекта в объект `stock1`.

#### Оператор

```
stock2 = stock1; // присвоение объекта
```

показывает, что вы можете присвоить один объект другому объекту того же типа. Как и в случае присваивания структур, в процессе присваивания объекта типа класс по умолчанию копируются элементы одного объекта в элементы другого. В этом случае первоначальное содержимое объекта `stock2` затирается.

#### ПОМНИТЕ

Когда вы присваиваете один объект другому объекту того же класса, C++ по умолчанию копирует содержимое каждого элемента исходного объекта в соответствующий элемент данных целевого объекта.

Обратите внимание на то обстоятельство, что в результате выполнения программы сначала отображается *Bye, Nifty Foods!*, а затем уже содержимое нового объекта `stock1`. После этого в самом конце программа говорит: *Bye, NanoSmart!* и *Bye, Nifty Foods!*. Откуда исходят эти трогательные прощальные слова? Напомним вам, что в деструкторе имеется оператор вывода, обеспечивающий этот эффект, чтобы вы могли видеть, когда вызван деструктор. (Это всего лишь средство обучения, а не обычное средство проектирования!) Два заключительных прощальных приветствия имеют место, когда прекращается выполнение функции `main()`, при этом два локальных объекта (`stock1` и `stock2`), которые объявляет программа, не попадают в диапазон доступа. Поскольку подобного рода автоматические переменные поступают в стек, объект, созданный последним, удаляется первым, а объект, созданный первым, удаляется последним. (Обратите внимание на то, что строка "NanoSmart" первоначально находилась в `stock1`, однако позднее была переведена в `stock2`.)

Однако что можно сказать о первом прощальном слове, обращенном к Nifty Foods? Когда программа использует конструктор для присвоения атрибутов Nifty Food объекту `stock1`, вспомните, что программа сначала создает временный, безымянный объект, в котором сохраняются эти значения. Затем эти значения копируются

в объект `stock1`. По завершению этого, когда все задачи, возложенные на временный объект, будут выполнены, программа вызывает деструктор, чтобы удалить его. Первый удаленный Nifty Foods — это временный объект, а второй удаленный Nifty Foods — это объект `stock1`. Между прочим, C++ специально не сообщает, что временный объект больше не существует. В нашей версии компилятора (Microsoft Visual C++ 5.0) объект удаляется сразу, как только в нем отпадет необходимость, в то время как Turbo C++ 2.0 удаляет временные объекты только после того, как прекратится выполнение функции.

Этот незначительный эпизод показывает, что между двумя приводимыми ниже операторами существует принципиальное различие:

```
Stock stock2 = Stock("Boffo Objects", 2, 2.0);
// временный объект
stock1 = Stock("Nifty Foods", 10, 50.0);
```

Первый из представленных операторов выполняет инициализацию; он создает объект с заданным значением. Второй оператор является оператором присваивания. Он создает временный объект и затем копирует его в существующий. Этот оператор менее эффективен, чем оператор инициализации. (Тем не менее, компилятору предоставлена возможность реализации формы инициализации (рассматривалось ранее применительно к `stock2`) путем создания временного объекта с последующим копированием его содержания в `stock2`.)

#### СОВЕТ

Если у вас имеется возможность устанавливать значения объекта путем инициализации или присваивания, выбирайте инициализацию. Как правило, она более эффективна.

Результаты подсчета, отображенные функцией `show()` при завершении выполнения программы, показывают, что обе операции — присваивание и восстановление объектов с помощью конструктора и последующего присваивания — выполняются.

#### Функции-элементы типа `const`

Рассмотрим следующие фрагменты программного кода:

```
const Stock land =
 Stock("Kludgehorn Properties");
land.show();
```

В рассматриваемой версии C++ компилятор отвергает вторую строку. Почему? Да потому, что код функции `show()` не дает гарантии, что не будет модифицирован вызывающий объект, который, будучи объявленным как `const`, не должен подвергаться изменениям. Вы уже решали раньше подобного рода проблемы, объявляя аргументы функции ссылками `const` или

указателями `const`. Но при этом перед нами встают синтаксические проблемы: метод `show()` вообще не имеет аргументов. Вместо этого объект, который он использует, предоставляет ему неявно путем вызова метода. Все, что необходимо в таких случаях, — это новое синтаксическое средство, которое гарантирует, что функция не внесет изменений в вызывающий ее объект. В C++ эта проблема решается с помощью ключевого слова `const`, которое ставится после скобок функции. Другими словами, объявление `show()` принимает такой вид:

```
void show() const; // обещает не вносить
 // изменений в
 // вызывающий объект
```

Аналогично начальная часть определения функции принимает такой вид:

```
void stock::show() const // обещает не
 // вносить изменений в
 // вызывающий объект
```

Функции, объявленные и определенные таким способом, называются функциями-элементами типа `const`. Подобно тому как вы используете ссылки и указатели типа `const` в качестве формальных аргументов функции там, где считаете нужным, необходимо размещать методы класса в категории `const` всякий раз, когда нужно, чтобы они не изменяли вызывающий объект. Мы будем впредь пользоваться этим правилом.

## Обзор конструкторов и деструкторов

Сейчас, когда мы обогатились опытом нескольких примеров использования конструкторов и деструкторов, вы, возможно, хотите остановиться и усвоить полученный материал. Чтобы помочь вам в этом деле, приведем обобщенное описание этих методов.

Конструктор — это функция-элемент класса специального назначения, которая вызывается всякий раз, когда создается некоторый объект этого класса. Конструктор какого-либо класса имеет то же имя, что и его класс. Однако благодаря перегрузке функций вы можете иметь сразу несколько конструкторов с одним и тем же именем при условии, что каждый из них имеет собственную сигнатуру, иначе говоря, собственный список аргументов. Кроме того, конструктор не имеет объявленного типа. Как правило, конструктор используется для инициализации элементов объекта класса. Проводимая вами инициализация должна соответствовать списку аргументов конструктора. Например, предположим, что у класса `Bozo` имеется следующий прототип конструктора класса:

```
// прототип конструктора
Bozo(char * fname, char * lname);
```

Его нужно использовать для инициализации новых объектов следующим образом:

```
// первичная форма
Bozo bozetta = bozo("Bozetta", "Biggens");
// укороченная форма
Bozo fufu("Fufu", "O'Dweeb");
// динамический объект
Bozo *pc = new Bozo("Popo", "Le Peu");
```

Если конструктор имеет только один аргумент, то этот конструктор вызывается, когда вы инициализируете объект значением, которое имеет тот же тип, что и аргумент конструктора. Например, предположим, что у вас имеется прототип этого конструктора:

```
Bozo(int age);
```

Вы можете использовать любую из представленных ниже форм инициализации объекта:

```
Bozo dribble = bozo(44); // первичная форма
Bozo roon(66); // вторичная форма
Bozo tubby = 32; // специальная форма
 // для конструкторов
 // с одним аргументом
```

Фактически третий пример представляет новый способ инициализации, мы его раньше не рассматривали, однако сейчас, по-видимому, самый подходящий момент сообщить вам об этом.

В главе 10 описан способ отключения этого свойства.

### ПОМНИТЕ

Конструктор, который вы можете использовать с единственным аргументом, предоставляет возможность использовать синтаксис присваивания для инициализации объекта некоторым значением:

```
Classname object = value;
```

Конструктор, заданный по умолчанию, не имеет аргументов, он используется в тех случаях, когда вы создаете объект без явной его инициализации. Если вы не можете воспользоваться каким-либо из конструкторов, компилятор выделяет конструктор, заданный по умолчанию. В противном случае нужно воспользоваться своим собственным конструктором. Он может вообще не иметь аргументов, иначе должны быть заданы значения по умолчанию для всех аргументов:

```
// прототип конструктора по умолчанию
Bozo();
// значение по умолчанию для класса Bistro
Bistro(const char * s = "Chez Zero");
```

Эта программа использует конструктор, заданный по умолчанию, для неинициализированных объектов:

```
// используется значение по умолчанию
Bozo bubu;
```

```
// используется значение по умолчанию
Bozo *pb = new Bozo;
```

Как и в процессе построения объекта, программа вызывает конструктор, а деструктор вызывается, когда нужно уничтожить объект. Каждый класс может иметь только один деструктор. Он не имеет возвращаемого типа, он не может даже иметь тип `void`; у него нет аргументов, а его имя является именем соответствующего класса, которому предшествует тильда. Деструктор класса `Bozo`, например, имеет следующий прототип:

```
~Bozo(); // деструктор класса
```

Деструкторы класса становятся необходимыми, когда конструкторы класса используют оператор `new`.

## Работа с указателем `this`

Продолжим работу с классом `Stock`. До сих пор каждая функция-элемент этого класса имела дело с одним-единственным объектом. Этим объектом был объект, который обращался к ней. Однако иногда возникает потребность в методе, который работает с двумя объектами, и эта задача решается с использованием специального указателя C++, получившего имя `this`. Рассмотрим, при каких обстоятельствах может возникнуть такая потребность.

Несмотря на то что объявление класса `Stock` отображает данные, ему не хватает аналитических возможностей. Например, проанализировав выходные данные функции `show()`, вы сможете сказать, какой из ваших вкладов превосходит по величине все остальные. Однако программа этого сделать не может, поскольку у нее нет непосредственного доступа к значению `total_val`. Самый прямой способ, позволяющий программе иметь сведения о хранимых данных, — предоставить в ее распоряжение методы, возвращающие значения. Как правило, для этой цели используется встроенный программный код:

```
class Stock
{
private:
 ...
 double total_val;
 ...
public:
 double total() const
 { return total_val; }
 ...
};
```

По существу, это определение превращает `total_val` в область памяти только для чтения, если дело касается прямого доступа программы к данным.

Добавив эту функцию в объявление класса, вы можете предоставить программе возможность проверить

ряд вкладов, чтобы найти тот из них, который превосходит все остальные по стоимости. Однако воспользуйтесь другим подходом, поскольку только так вы сможете достаточно хорошо изучить указатель `this`. Этот подход состоит в том, что определяется функция-элемент, которая просматривает два объекта `Stock` и возвращает ссылку на тот из них, который больше по величине. При попытке реализовать этот подход возникает целый ряд интересных вопросов, и сейчас мы перейдем к их рассмотрению.

Во-первых, как будет выглядеть функция-элемент, выполняющая сравнение двух объектов? Предположим, например, что вы решили назвать такой метод `topval()`. Далее, при вызове функции `stock1.topval()` обеспечивается доступ к данным объекта `stock1`, в то время как сообщение `stock2.topval()` получает доступ к данным объекта `stock2`. Если вы хотите, чтобы указанный выше метод провел сравнение двух объектов, то необходимо передать второй объект как аргумент. В целях повышения производительности передайте этот аргумент по ссылке. Иначе говоря, пусть метод `topval()` использует аргумент типа `const Stock &`.

Во-вторых, как вы возвратите ответ этого метода в вызывающую программу? Наиболее простой способ — заставить метод возвратить ссылку на объект, который имеет наибольшую общую стоимость. Таким образом, метод, выполняющий сравнение, должен иметь следующий прототип:

```
const Stock & topval(const Stock & s) const;
```

Эта функция осуществляет неявный доступ к одному из объектов, к другому объекту она получает прямой доступ, при этом она возвращает ссылку на один из двух объектов. Ключевое слово `const` в круглых скобках означает, что функция не вносит изменений в объект, к которому она получает прямой доступ, а ключевое слово `const`, которое следует сразу за скобками, означает, что функция не подвергает модификации объект, к которому она осуществляет неявный доступ. Поскольку функция возвращает ссылку на один из объектов `const`, возвращаемый тип также имеет ссылку `const`.

Предположим далее, что вы хотите сравнить объекты `stock1` и `stock2` класса `Stock` и присвоить тот из них, который имеет большую общую стоимость, объекту `top`. Для этого можно воспользоваться любым из следующих двух операторов:

```
·top = stock1.topval(stock2);
·top = stock2.topval(stock1);
```

Первый из них получает прямой доступ к объекту `stock1` и неявный доступ к объекту `stock2`, в то время как второй получает прямой доступ к объекту `stock1` и неявный доступ к объекту `stock2` (рис. 9.3).

Каким бы ни был доступ, этот метод сравнивает два объекта и возвращает ссылку на тот из них, у которого общая стоимость больше.

По существу, такой способ записи может привести к путанице. Было бы намного проще, если бы вы каким-то образом могли использовать оператор сравнения `>` для сравнения двух объектов. Это можно сделать с помощью перегрузки операторов, которая рассматривается в главе 10.

Между тем, мы еще не все выяснили из того, как реализовать функцию `topval()`. При этом возникает небольшая проблема. Ниже представлена частичная реализация, которая позволяет нам вникнуть в суть этой проблемы.

```
const Stock & Stock::topval(const Stock & s) const
{
 if (s.total_val > total_val)
 return s; //аргумент типа объект
 else
 return ?????; //объект, вызывающий
 //метод topval
}
```

Здесь `s.total_val` — это общее значение для объекта, переданного в качестве аргумента, а `total_val` — общее значение для объекта, которому было отправлено сообщение. Если значение `s.total_val` больше значения `total_val`, то функция возвращает `s`. В противном случае она возвращает объект, использованный для вызова метода. (В среде ООП говорят, что это объект, которому передается сообщение `topval`.) Проблема заключается в том, как назвать этот объект? Если вы присвоите ему имя `stock1.topval(stock2)`, то `s` будет ссылкой на `stock2`

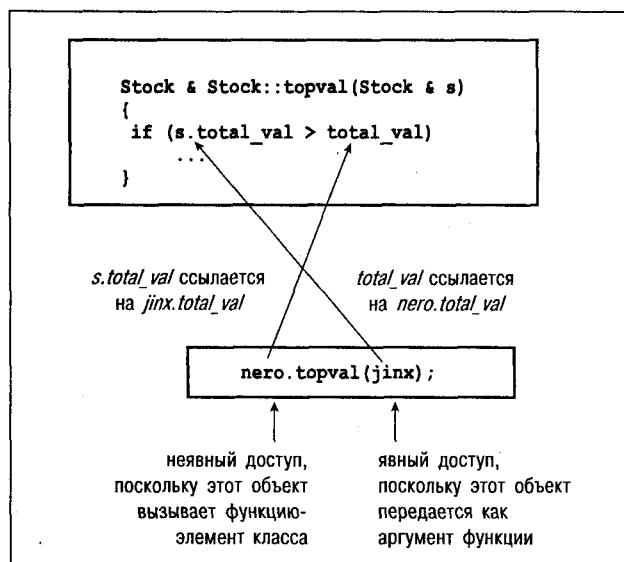


РИСУНОК 9.3 Осуществление доступа к двум объектам посредством функции-элемента

(т.е. псевдоним для `stock2`), однако псевдонима у `stock1` нет.

В C++ эта проблема решается путем использования специального указателя `this`. Указатель `this` ссылается на объект, который используется для вызова функции-элемента. (По существу, указатель `this` передается как скрытый аргумент рассматриваемого метода.) Следовательно, при вызове функции `stock1.topval(stock2)` устанавливается указатель `this` на адрес объекта `stock1` и обеспечивается доступ к этому указателю со стороны метода `topval()`. Аналогично при вызове функции `stock2.topval(stock1)` устанавливается указатель `this` на адрес объекта `stock2`. В общем случае для всех методов класса указатель `this` указывает на адрес объекта, который вызывает этот метод. И в самом деле, `total_val` в методе `topval()` — всего лишь сокращенное обозначение от `this->total_val`. (Напомним, что в главе 4 вы пользовались оператором `->` для доступа к элементам структур посредством указателя. То же самое справедливо и для элементов класса.) (рис. 9.4).

#### УКАЗАТЕЛЬ THIS

Каждая функция-элемент, включая конструкторы и деструкторы, имеет указатель `this`. Характерной особенностью указателя `this` является то, что он указывает на вызывающий объект. Если у какого-либо метода возникает необходимость ссылки на объект как на единое целое, он может воспользоваться выражением `*this`. Спецификатор `const`, помещенный сразу за скобками, в которые заключены аргументы, истолковывается в рассматриваемом случае указатель `this` как `const`; в такой ситуации вы не можете использовать `this`, чтобы изменить значение объекта.

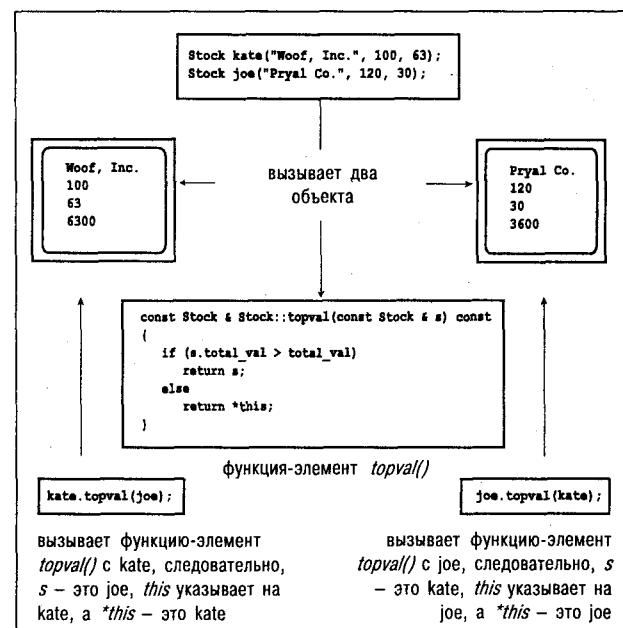


РИСУНОК 9.4 Указатель `this` ссылается на вызывающий объект.

Однако то, что вы хотите возвратить, — это вовсе не указатель `this`, так как `this` выражает адрес объекта. Вы же хотите вернуть сам объект, а он обозначается как `*this`. (Напомним, что в результате применения оператора разыменования `*` к указателю получается величина, на которую указывает указатель.) Теперь вы можете завершить определение метода, используя `*this` как псевдоним вызывающего объекта.

```
const Stock & Stock::topval(const Stock & s) const
{
 if (s.total_val > total_val)
 return s; //объект в качестве аргумента
 else
 return *this; //вызывающий объект
}
```

Тот факт, что возвращаемый тип является ссылкой, означает, что возвращаемый объект сам является вызывающим объектом, и ни в коем случае не означает, что механизм возврата передает копию. В листинге 9.7 мы видим новый заголовочный файл.

#### Листинг 9.7 Файл stock2.h.

```
// stock2.h
#ifndef _STOCK2_H_
#define _STOCK2_H_
class Stock
{
private:
 char company[30];
 int shares;
 double share_val;
 double total_val;
 void set_tot()
 {
 total_val = shares * share_val;
 }
public:
 Stock(); //конструктор по умолчанию
 Stock(const char * co, int n, double pr);
 ~Stock() {} //деструктор, который не
 //выполняет никаких действий
 void buy(int num, double price);
 void sell(int num, double price);
 void update(double price);
 void show() const;
 const Stock & topval(const Stock & s) const;
};
#endif
```

В листинге 9.8 представлен файл модифицированных методов класса. Он содержит новый метод `topval()`. Кроме того, теперь, когда вы знаете, как работает метод деструктора, можно заменить его на более совершенную версию.

#### Листинг 9.8 Программа stock2.cpp.

```
// stock2.cpp - методы класса Stock
#include <iostream>
using namespace std;
```

```
#include <cstdlib> // для функции exit()
#include <cstring> // для функции strcpy()
#include "stock2.h"

// конструкторы
Stock::Stock()
{
 strcpy(company, "no name");
 shares = 0;
 share_val = 0.0;
 total_val = 0.0;
}

Stock::Stock(const char * co, int n, double pr)
{
 strcpy(company, co);
 shares = n;
 share_val = pr;
 set_tot();
}

void Stock::buy(int num, double price)
{
 shares += num;
 share_val = price;
 set_tot();
}

void Stock::sell(int num, double price)
{
 if (num > shares)
 {
 cerr << "You can't sell more
 than you have!\n";
 exit(1);
 }
 shares -= num;
 share_val = price;
 set_tot();
}

void Stock::update(double price)
{
 share_val = price;
 set_tot();
}

void Stock::show() const
{
 cout << "Company: " << company
 << "Shares: " << shares << '\n'
 << "Share Price: $" << share_val
 << "Total Worth: $" << total_val << '\n';
}

const Stock & Stock::topval(const Stock & s) const
{
 if (s.total_val > total_val)
 return s;
 else
 return *this;
}
```

Разумеется, мы хотим знать, работает ли указатель `this`, и самым подходящим местом для использования нового метода является программа с массивом объектов, а для этого нам придется перейти к обсуждению следующей темы.

## Массив объектов

Довольно часто, как показывает пример класса `Stock`, возникает необходимость иметь несколько объектов одного и того же класса. Вы можете создавать отдельные переменные типа объект, что мы и делали до сих пор в рассматриваемых выше примерах, однако гораздо целесообразнее создавать массивы объектов. Это можно сравнить с решительным скачком в неизведанное, однако в действительности вы объявляете массив объектов таким же способом, каким бы вы объявили массив любых других стандартных типов.:

```
Stock mystuff[4]; //создается массив из
//четырех объектов класса Stock
```

Напомним, что программа всегда обращается к конструкторам классов, определенным по умолчанию, когда она создает объекты класса, которые не инициализированы явно. Данное объявление требует либо чтобы класс вообще явно не определял никаких конструкторов, поскольку в этом случае используется неявно конструктор, определенный по умолчанию, но не выполняющий никаких действий, либо, как в данном случае, чтобы конструктор по умолчанию был определен явно. Каждый элемент — `mystuff[0]`, `mystuff[1]` и т.п. — это объект класса `Stock`, и, следовательно, он может быть использован с методами класса `Stock`:

```
//применить функцию update() к первому элементу
mystuff[0].update();
//применить функцию show() к четвертому элементу
mystuff[3].show();

// сравнивать второй и третий элементы
Stock tops = mystuff[2].topval(mystuff[1]);
```

Вы можете использовать конструктор для инициализации элементов этого массива. В этом случае нужно вызывать конструктор для каждого отдельного элемента:

```
const int STKS = 4;
Stock stocks[STKS] = {
 Stock("NanoSmart", 12.5, 20),
 Stock("Boffo Objects", 200, 2.0),
 Stock("Monolithic Obelisks", 130, 3.25),
 Stock("Fleep Enterprises", 60, 6.5)
};
```

В приведенном выше программном коде для инициализации массива используется стандартная форма: список значений, отделенных друг от друга запятыми и заключенных в фигурные скобки. В данном случае при вызове метода конструктора отображается каждое значение. Если у класса имеется несколько конструкторов, вы можете применять различные конструкторы для различных элементов:

```
const int STKS = 10;
Stock stocks[STKS] = {
 Stock("NanoSmart", 12.5, 20),
 Stock(),
 Stock("Monolithic Obelisks", 130, 3.25),
};
```

В данном случае инициализация объектов `stocks[0]` и `stocks[2]` выполняется с использованием конструктора `Stock(const char * co, int n, double pr)`, а инициализация `stocks[1]` — с помощью конструктора `Stock()`. Поскольку такое объявление только частично инициализирует массив, остальные семь элементов инициализируются с помощью конструктора, заданного по умолчанию.

Схема инициализации массива объектов все еще использует конструктор, заданный по умолчанию, для создания элементов этого массива. Затем конструктор в фигурных скобках создает временные объекты, содержимое которых копируется в виде вектора. Таким образом, класс должен иметь конструктор, заданный по умолчанию, если вы хотите создавать массивы объектов класса.

### ПРЕДОСТЕРЖЕНИЕ

Если вы хотите создать массив объектов некоторого класса, то этот класс должен иметь в своем распоряжении конструктор, заданный по умолчанию.

Программа, представленная в листинге 9.9, реализует эти идеи в небольшом коде, который инициализирует массив из четырех элементов, отображает их содержимое и проверяет эти элементы с целью выбрать элемент с наибольшим общим значением. Поскольку функция `topval()` проверяет одновременно только два объекта, в программе употребляется цикл `for` для проверки всего массива. Используйте заголовочный файл и файл методов, представленные соответственно в листингах 9.7 и 9.8.

### Листинг 9.9 Программа usestok2.cpp.

```
// usestok2.cpp — используется класс Stock
#include <iostream>
using namespace std;
#include "stock2.h"
const int STKS = 4;
int main()
{
//создание массива инициализированных объектов
 Stock stocks[STKS] = {
 Stock("NanoSmart", 12, 20.0),
 Stock("Boffo Objects", 200, 2.0),
 Stock("Monolithic Obelisks", 130, 3.25),
 Stock("Fleep Enterprises", 60, 6.5)
 };

 cout.precision(2); // формат #.##
 cout.setf(ios_base::fixed,
 ios_base::floatfield); // формат #.##
 cout.setf(ios_base::showpoint); //формат #.##
```

```

cout << "Stock holdings:\n";
int st;
for (st = 0; st < STKS; st++)
 stocks[st].show();
Stock top = stocks[0];
for (st = 1; st < STKS; st++)
 top = top.topval(stocks[st]);
cout << "\nMost valuable holding:\n";
top.show();
return 0;
}

```

```

<< "Share Price: $" << this->share_val
<< "Total Worth: $"
<< this->total_val << '\n';
}

```

Следовательно, этот код преобразует спецификатор **Stock::** в аргумент функции, который является указателем на **Stock**, а затем использует этот указатель для доступа к элементам класса.

Аналогично препроцессор преобразует вызовы функции типа

```
top.show(); B show(&top);
```

Таким образом, указателю **this** присвоен адрес вызывающего объекта. (Фактические детали могут оказаться более сложными и запутанными.)

## Диапазон доступа класса

В главе 8 рассматривается глобальный диапазон доступа, иначе говоря, область доступа файла и локальный диапазон доступа, иначе говоря, область доступа блока. Напомним, вы можете использовать перемененную с глобальным диапазоном доступа в любом месте файла, который содержит ее определение, в то время как переменная с локальным диапазоном доступа является локальной по отношению к блоку, который содержит ее определение. Имена функций также могут иметь глобальный диапазон доступа, но они никогда не имеют локального диапазона доступа. Классы C++ привносят новый вид области доступа: **диапазон доступа класса**. Диапазон доступа конкретного класса применяется к именам, определенным в этом классе, например, к именам-элементам данных класса и функциям-элементам класса. Таким образом, вы можете использовать те же имена элементов данных класса в других классах и при этом избегать конфликтов имен: элемент данных **shares** класса **Stock** — переменная, которая отличается от элемента данных **shares** класса **JobRide**. Кроме того, диапазон доступа класса определяет то, что вы не можете получить прямой доступ к элементам класса из внешнего мира. Это правило действует даже в отношении общедоступных функций-элементов. Иначе говоря, чтобы обратиться к общедоступной функции-элементу, нужно воспользоваться объектом:

```

// создается объект
Stock sleeper("Exclusive Ore", 100, 0.25);
// объект используется для обращения
// к функции-элементу
sleeper.show();
// неправильно - вы не можете
// напрямую обратиться к методу
show();

```

## ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Вам, возможно, следует воспользоваться файлами заголовков **stdlib.h** и **string.h** вместо **cstdlib** и **cstring**. Кроме того, вам, вероятно, придется употребить более старый **ios::** вместо **ios\_base::**.

Результаты выполнения рассматриваемой программы:

```

Stock holdings:
Company: NanoSmart Shares: 12
 Share Price: $20.00 Total Worth:
 $240.00
Company: Boffo Objects Shares: 200
 Share Price: $2.00 Total Worth: $400.00
Company: Monolithic Obelisks Shares: 130
 Share Price: $3.25 Total Worth: $422.50
Company: Fleep Enterprises Shares: 60
 Share Price: $6.50 Total Worth: $390.00
Most valuable holding:
Company: Monolithic Obelisks Shares: 130
 Share Price: $3.25 Total Worth: $422.50

```

Следует обратить внимание на одну особенность: основной объем работы приходится на разработку класса. После того как эта работа выполнена, написание программы не представляет собой трудностей.

Между прочим, умение пользоваться указателем **this** позволяет заглянуть на "кухню" языка C++. Например, препроцессор C++ **cfront** преобразует программы, написанные на C++, в программы, написанные на C. Чтобы успешно работать с определениями методов, все, что нужно сделать, это перевести определение метода в C++, например, такое:

```

void Stock::show() const
{
 cout << "Company: " << company
 << "Shares: " << shares << '\n'
 << "Share Price: $" << share_val
 << "Total Worth: $" << total_val
 << '\n';
}

```

в следующее определение в C:

```

void show(const Stock * this)
{
 cout << "Company: " << this->company
 << "Shares: " << this->shares << '\n'

```

Аналогично при определении функции-элемента нужно воспользоваться оператором определения диапазона доступа:

```
void Stock::update(double price)
{
 ...
}
```

Итак, в объявлении класса или в определении функции-элемента вы можете использовать не перегруженное деталями имя (неуточненное имя), например, когда `sell()` вызывает функцию-элемент `set_tot()`. В противном случае, когда вы используете имя элемента класса, нужно воспользоваться оператором прямого членства (`.`), оператором косвенного членства (`->`) или оператором определения диапазона доступа (`::`), в зависимости от контекста.

В некоторых случаях желательно иметь символьные константы диапазона доступа класса. Например, при объявлении класса `Stock` был использован литерал `30`, чтобы указать размер массива `company`. Кроме того, поскольку константа одна и та же для всех объектов, было бы совсем неплохо создать единую константу, которой могли бы пользоваться все объекты. Можно предположить, что решением может служить следующий фрагмент:

```
class Stock
{
private:
 const int Len = 30; //объявить константу?
 char company[Len];
 ...
}
```

Но это не срабатывает, поскольку объявляющий класс описывает, как должен выглядеть объект, но не создает его. Отсюда следует: пока вы не создадите объект, место для хранения переменной не существует. Существуют, однако, способы достижения практически того же результата.

Прежде всего, вы можете объявить перечисление внутри класса. Перечисление, заданное в классе объявления, обладает диапазоном доступа класса, следовательно, вы можете воспользоваться перечислением, чтобы назначать символьные имена с диапазоном доступа класса целым константам. Это означает, что вы можете начать объявление класса `Stock` следующим образом:

```
class Stock
{
private:
 enum { Len = 30 } ; //специфическая для
 //данного класса константа
 char company[Len];
 ...
}
```

Обратите внимание на тот факт, что объявление перечисления таким способом не приводит к созданию элементов данных класса. Иначе говоря, каждый отдельный объект не содержит в себе перечисление. Скорее, `Len` является всего лишь символьным именем, которое компилятор заменяет константой `30`, когда сталкивается с ним в диапазоне доступа класса.

Поскольку перечисление используется только для создания символьной константы и при этом не преследовалась цель создания переменных типа перечисление, нет необходимости создавать метку перечисления. Между прочим, в рамках многих реализаций класс `ios_base` делает нечто подобное в своем общедоступном разделе; таковым является источник идентификаторов, например, `ios_base::fixed`. В данном случае `fixed` — это перечисление, объявленное в классе `ios_base`.

Недавно в C++ был реализован второй путь определения констант внутри класса — использование ключевого слова `static`:

```
class Stock
{
private:
 static const int Len = 30; //объявление
 //константы!
 char company[Len];
 ...
}
```

В этом фрагменте программного кода создается единственная константа с именем `Len`, которая хранится вместе с другими статическими переменными, а не внутри объекта. Следовательно, существует только одна константа `Len`, которая совместно используется всеми объектами класса `Stock`. В главе 11 проводится дальнейший анализ элементов класса типа `static`. Вы можете использовать этот метод только для объявления `static` целочисленных значений и перечислений. Вы не можете хранить константу типа `double` в таком виде.

## Абстрактный тип данных

Класс `Stock` достаточно специфичен. Однако часто программисты определяют классы для того, чтобы те представляли более общие идеи. Например, классы представляют собой хорошее средство реализации того, что специалисты в теории вычислительных машин называют *абстрактными типами данных* или сокращенно *ADT* (*abstract data types*). Как следует из названия, *ADT* описывает тип данных в наиболее общей форме, без привлечения языковых понятий или деталей реализации. Рассмотрим в качестве примера обычный стек. Стек представляет собой способ хранения данных, в рамках которого данные всегда добавляются или удаляются из вершины стека. Программы, написанные на C++, например, используют стек для работы с автоматически-

ми переменными. По мере возникновения автоматических переменных они добавляются в вершину стека. Когда их действие прекращается, они удаляются из стека.

Опишем свойства стека в наиболее общем, абстрактном виде. Прежде всего, стек содержит несколько элементов. (Это свойство делает его *контейнером*, а это еще более абстрактное понятие.) Стеком можно манипулировать следующим образом:

- Создать пустой стек.
- Добавить элемент в вершину стека ("затолкнуть" элемент в стек).
- Удалить элемент из вершины стека ("вытолкнуть" элемент из стека).
- Проверить, заполнен ли стек.
- Проверить, пуст ли стек.

Можно сопоставить это описание с объявлением класса, в котором общедоступная функция-элемент обеспечивает интерфейс, реализующий операции со стеком. Приватные элементы данных принимают на себя функции по сохранению данных, помещенных в стек. Понятие класса хорошо согласуется с подходом ADT.

Раздел **private** должен взять на себя задачу выбора способа представления данных. Например, вы можете использовать обычный массив, массив динамически распределенной памяти или некоторые другие, более совершенные структуры данных, такие как связанный список. Однако общедоступный интерфейс должен скрывать точное представление данных. Наоборот, он должен быть выражен в самых общих формулировках, таких как создание стека, "заталкивание" элемента, "выталкивание" элемента и т.д. В листинге 9.10 представлен один из подходов. Предполагается, что реализован тип **bool**. Если ваша система не включает такой тип данных, вы можете использовать тип **int**, 0 и 1 вместо **bool**, **false** и **true**.

#### Листинг 9.10 Программа stack.h.

```
// stack.h - определение стека ADT в классе
#ifndef _STACK_H_
#define _STACK_H_
typedef unsigned long Item;
class Stack
{
private:
 enum { MAX = 10 }; //константа, специфичная
 //для данного класса
 Item items[MAX]; //содержит элементы стека
 int top; //указатель элемента,
 //находящегося в вершине стека
public:
 Stack();
 bool isempty() const;
 bool isfull() const;
```

```
// функция push() возвращает false,
// если стек уже заполнен, и true -
// в противном случае
bool push(const Item & item); //добавить
 //элемент в стек
// функция pop() возвращает false, если стек
// уже пуст, и true - в противном случае
bool pop(Item & item); // "вытолкнуть"
 //элемент из вершины стека
};
```

---

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если в вашей системе тип **bool** не реализован, вы можете воспользоваться типом данных **int**, 0 и 1 вместо **bool**, **false** и **true**. С другой стороны, ваша система, возможно, поддерживает нестандартную форму, такую как **boolean** или **Boolean**, которая использовалась ранее.

В этом примере раздел **private** показывает, что стек реализован с использованием массива, однако общедоступный раздел это не демонстрирует. Следовательно, можно заменить массив на динамический массив без внесения изменений в интерфейс класса. Это означает, что изменение способа реализации стека не требует вносить изменения в программные коды, которые пользуются этим стеком. Вам достаточно лишь перекомпилировать программный код стека и связать его с имеющимися программными кодами.

Интерфейс избыточен в том плане, что функции **pop()** и **push()** возвращают информацию о состоянии стека (пуст или заполнен), вместо того, чтобы иметь тип **void**. Это ставит программиста перед выбором: либо найти способ решения проблемы переполнения стека, либо очистить стек. Пользователь может воспользоваться функциями **isempty()** и **isfull()**, чтобы проверить состояние стека, прежде чем предпринимать попытку изменить стек, либо использовать значения, возвращенные функциями **push()** и **pop()**, чтобы убедиться, что операция выполнена успешно.

Вместо того, чтобы давать определение стека, используя терминологию некоторого конкретного типа, класс описывает его, опираясь на терминологию обобщенного типа **Item**. В этом случае заголовочный файл употребляет термин **typedef**, чтобы сделать тип **Item** таким, как и **unsigned long**. Если вам нужен, скажем, стек для значений **double** или значений типа структура, вы можете изменить **typedef** и оставить объявления класса и метода без изменений. Шаблоны классов (см. главу 13) представляют собой более мощный метод изоляции типов данных при проектировании класса.

Далее попробуем реализовать методы класса. Листинг 9.11 предоставляет нам такую возможность.

**Листинг 9.11 Программа stack.cpp.**

```
// stack.cpp - функция-элемент стека
#include "stack.h"
Stack::Stack() // построить пустой стек
{
 top = 0;
}
bool Stack::isempty() const
{
 return top == 0;
}
bool Stack::isfull() const
{
 return top == MAX;
}
bool Stack::push(const Item & item)
{
 if (top < MAX)
 {
 items[top++] = item;
 return true;
 }
 else
 return false;
}
bool Stack::pop(Item & item)
{
 if (top > 0)
 {
 item = items[--top];
 return true;
 }
 else
 return false;
}
```

Конструктор, заданный по умолчанию, гарантирует, что все стеки создаются пустыми. Программный код функций `pop()` и `push()` гарантирует, что управление вершиной стека осуществляется должным образом. Гарантии, подобные этой, представляют собой факторы, которые придают проблемно-ориентированному программированию больше надежности. Предположим, однако, что вместо этого вы создали отдельный массив для представления стека и независимую переменную для представления индекса в вершине. Далее вашей обязанностью становится получение нужного кода каждый раз именно в тот момент, когда вы создаете новый стек. Без тех средств защиты, которые предлагают приватные данные, всегда сохраняется опасность возникновения некоторых программных ошибок, которые могут повлиять на изменения в данные.

Проверим, как работает этот стек. Программа, представленная в листинге 9.12, моделирует действия служащего, который обрабатывает заказы на покупку, беря из ящика ту квитанцию, которая лежит сверху, т.е. использует подход LIFO (последним пришел, первым обслужен), реализованный в стеке.

**Листинг 9.12 Программа stacker.cpp.**

```
// stacker.cpp - проверочный класс Stack
#include <iostream>
using namespace std;
#include <cctype> // или ctype.h
#include "stack.h"
int main()
{
 Stack st; // построить пустой стек
 char c;
 unsigned long po;
 cout << "Please enter A to add a purchase order,\n"
 << "P to process a PO, or Q to quit.\n";
 while (cin >> c && toupper(c) != 'Q')
 {
 while (cin.get() != '\n')
 continue;
 if (!isalpha(c))
 {
 cout << '\a';
 continue;
 }
 switch(c)
 {
 case 'A':
 case 'a': cout << "Enter a PO number "
 << "to add: ";
 cin >> po;
 if (st.isfull())
 cout << "stack already full\n";
 else
 st.push(po);
 break;
 case 'P':
 case 'p': if (st isempty())
 cout << "stack already empty\n";
 else {
 st.pop(po);
 cout << "PO #" << po
 << " popped\n";
 }
 break;
 }
 cout << "Please enter A to add a "
 << "purchase order,\n"
 << "P to process a PO, or Q to quit.\n";
 }
 cout << "Bye\n";
 return 0;
}
```

Небольшой цикл `while`, с помощью которого удаляется остаточная часть строки, здесь не нужен, но он удачно вписывается в модификацию этой программы, рассматриваемую в главе 13. Вот результаты выполнения этой программы:

```
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 17885
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
```

```

P
PO #17885 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 17965
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 18002
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #18002 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #17965 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
stack already empty
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
Q
Bye

```

## Резюме

Объектно-ориентированное программирование уделяет особое внимание представлению данных в программе. Первый шаг к решению проблем в программировании с использованием подхода ООП — это описание данных с использованием терминов ее интерфейса с программой, которое дает представление о том, как используются данные. Далее проектируется класс, который реализует этот интерфейс. Как правило, приватные элементы данных хранят информацию, в то время как общедоступные функции-элементы, которые по-другому называются методами, обеспечивают только доступ к данным. Такой класс объединяет данные и методы в один модуль, а статус приватных данных обеспечивает скрытие данных.

Как правило, вы делите объявление класса на две части, которые обычно содержатся в разных файлах. Все, что относится к объявлению класса, помещается в заголовочный файл, при этом методы представлены прототипами функций. Исходные программные коды, которые определяют функции-элементы, помещаются в файл методов. Такой подход отделяет описание интерфейса от деталей реализации. По существу, вам нужно знать только общедоступный интерфейс класса, чтобы пользоваться им. Разумеется, вы можете рассматривать реализацию (если класс не был предоставлен в откомпилированном виде), однако ваша программа не должна опираться на детали реализации, такая информация о некоторой конкретной величине хранится как тип `int`. До тех пор пока

программа и класс взаимодействуют между собой с помощью методов, определяющих интерфейс, вы сохраняете возможность совершенствовать любую из частей по отдельности посредством этих методов, не задумываясь о возможных непредвиденных взаимодействиях.

Класс — это тип, определенный пользователем, а объект — экземпляр класса. Это означает, что объект представляет собой переменную этого типа, т.е. эквивалентен переменной, хранящей, например, значение памяти, выделенной оператором `new` в соответствии со спецификацией класса. Язык C++ стремится сделать типы, определенные пользователем, максимально подобными стандартным типам, следовательно, вы получаете возможность объявлять объекты, указатели на объекты и массивы объектов. Вы можете передавать объекты как аргументы, возвращать их как значения, возвращаемые функциями и присваивать один объект другому объекту того же типа. Если вы предусматриваете использование метода конструктора, можете инициализировать объекты в момент их создания. Если вы предусматриваете использование метода деструктора, программа запускает этот метод в работу сразу после того, как объект прекращает свою деятельность.

Каждый объект хранит собственные копии порции данных, принадлежащих объявлению класса, однако они совместно используют методы класса. Если `mr_object` — имя конкретного объекта и `try_me()` — это функция-элемент, вы вызываете функцию-элемент, используя точечный оператор членства: `mr_object.try_me()`. В терминологии ООП этот вызов функции описывается как посылка сообщения `try_me` объекту `mr_object`. Любая ссылка на элементы данных класса в методе `try_me()` после этого относится к элементам данных объекта `mr_object`. Аналогично в результате вызова функции `i_object.try_me()` обеспечивается доступ к элементам данных объекта `i_object`.

Если вы хотите, чтобы областью действия какой-либо функции-элемента был более чем один объект, можете передать этому методу дополнительные объекты в качестве аргументов. Если нужно, чтобы метод ссылался прямо на объект, который его вызвал, можете воспользоваться для этой цели указателем `this`. Этот указатель настраивается на адрес вызывающего объекта, следовательно, `*this` является псевдонимом самого этого объекта.

Классы с высокой точностью соответствуют абстрактным типам данных (ADT — abstract data types). Интерфейс общедоступной функции-элемента предоставляет услуги, описываемые абстрактными типами данных, а приватный раздел класса и программные коды методов класса представляет собой реализацию, скрытую от клиентов класса.

## Вопросы для повторения

1. Что такое класс?
2. Как класс реализует абстракцию, инкапсуляцию и сокрытие данных?
3. Какими являются отношения между объектом и классом?
4. Чем, кроме своих функциональных свойств, функция-элемент отличается от элементов данных класса?
5. Дайте определение класса, представляющего банковский счет. Элементы данных должны содержать имя вкладчика, номер счета (использовать строку) и сальдо. Функции-элементы должны выполнять такие действия, как:
  - Создание объекта и его инициализация.
  - Назначение начальных значений элементам данных.
  - Отображение имени вкладчика, номера счета и сальдо.
  - Вклад на банковский счет суммы, заданной аргументом.
  - Снятие с банковского счета суммы, заданной аргументом.
- Показать объявление класса, а не реализацию метода (упражнение 1 по программированию предоставляет вам возможность написать реализацию.)
6. Когда вызываются конструкторы класса? Когда вызываются деструкторы класса?
7. Разработать программный код конструктора класса банковского счета, учитывая требования, сформулированные в вопросе 5.
8. Что такое конструктор, определенный по умолчанию, и какую пользу приносит его наличие?
9. Изменить класс **Stock** (его версия содержится в файле **stock2.h**) таким образом, чтобы в нем были функции-элементы, которые возвращают значения отдельных элементов данных. Примечание: элемент, который возвращает имя компании, не должен служить орудием, способным вносить изменения в массив. Иначе говоря, он не может просто возвращать **char \***, он должен возвращать указатель **const** или указатель на копию массива, который был создан с помощью спецификатора **new**.
10. Что такое **this** и **\*this**?

## Упражнения по программированию

1. Сформулируйте определения метода класса, описанного в пункте 5 вопросов для повторения, и написать короткую программу, иллюстрирующую все эти свойства.
2. Выполните упражнение 4 из главы 8, но при этом замените программный код, приведенный там, на соответствующее объявление класса **golf**. Используйте конструктор с соответствующими аргументами для назначения начальных значений.
3. Рассмотрите следующее определение структуры:
 

```
struct customer {
 char fullname[35];
 double payment;
};
```

Напишите программу, которая добавляет и удаляет структуры из стека, описанного в объявлении класса. Каждый раз, когда удаляется запись о заказчике, его платеж прибавляется к текущей сумме, а сама текущая сумма выводится на экран. Примечание: нужно иметь возможность сохранять класс **Stack** неизменным; достаточно изменить объявление **typedef** так, чтобы **Item** имел тип **customer**, а не тип **unsigned long**.
4. Объявление класса имеет вид:
 

```
class Move
{
private:
 double x;
 double y;
public:
 //присваивает x, y значения a, b
 Move(double a = 0, double b = 0);
 //отображает текущие значения x, y
 showmove() const;
 //данная функция присваивает x из т
 //для переменной x из вызывающего
 //объекта, чтобы получить новое x,
 //прибавляет y из т для y вызывающего
 //объекта, чтобы получить новое y,
 //создает новый объект, инициализированный
 //новыми значениями x,y, и возвращает его
 Move add(const Move & m) const;
 // присваивает x,y значения a, b
 reset(double a = 0, double b = 0);
};
```

Дайте определение функции-элемента и напишите программу, которая использует этот класс.
5. Класс **Betelgeusean plorg** обладает следующими свойствами:

- Данные

- У plorg имеется имя, содержащее не более 19 букв.
- plorg характеризуется индексом удовлетворенности (CI— contentment index), который представлен целым числом.

- Операции

- plorg начинается с имени и значения CI, равного 50.
- Значение CI может изменяться.
- plorg может отображать свое имя и CI.
- plorg по умолчанию имеет имя "Plorga".

- Сформулируйте определение класса **Plorg** (элементы данных и прототипы функций-элементов) который представляет plorg. Дайте определения функций-элементов. Напишите короткую программу, которая демонстрирует все свойства класса **Plorg**.

6. Мы можем дать такое описание простого списка:

- Простой список может содержать нуль или большее число элементов некоторого конкретного типа.
- Вы можете создать пустой список.
- Вы можете добавлять элементы в список.
- Вы можете узнать, пуст ли список.
- Вы можете проверить, полон ли список.
- Вы можете посещать каждый элемент списка и выполнить над ним определенные действия.

Нетрудно убедиться, что этот список и в самом деле простой, он, например, не допускает добавлений и удалений элементов. Такой список может быть использован главным образом в простых программных проектах. В этом случае разработайте класс, соответствующий этому описанию. Вы можете реализовать этот список в виде массива или в виде связанных списков, если вы знакомы с таким типом данных. Однако общедоступный интерфейс не должен зависеть от этого вашего выбора. Другими словами, общедоступный интерфейс не должен содержать в себе индексов массива, указателей на вершину и пр. Он должен быть выражен в обобщенных понятиях, таких как создание списка, добавление элемента в список и т.п. Обычный способ посещения каждого элемента и выполнения над ним конкретного действия осуществляется путем использования функции, которая принимает указатель на функцию как аргумент.

```
void visit(void (*pf)(Item &));
```

В данном случае **pf** указывает на функцию (но не на функцию-элемент), которая принимает ссылку на аргумент **Item**, где **Item** — это тип элементов списка. Функция **visit()** применяет эту функцию к каждому элементу списка.

Требуется также написать короткую программу, которая реализует этот проект.

# Работа с классами

**В этой главе рассматривается следующее:**

- Перегрузка операторов
- Дружественные функции
- Перегрузка операции <<, используемой при выводе данных
- Элементы состояния
- Использование функции rand() для генерации случайных чисел
- Автоматическое преобразование и приведение типов для классов
- Функции преобразования классов

Классы в C++ обладают богатым набором сложных и мощных свойств. В главе 9 вы приступили к теме объектно-ориентированного программирования и ознакомились с тем, как давать определения простым классам и как пользоваться ими. Вы узнали, каким образом класс определяет тип данных, предназначенных для представления объекта, а также изучили способы определения операций для функций-элементов. В дополнение к этому вы получили представление о двух функциях-элементах специального вида, конструкторах и деструкторах, которые способны создавать и удалять объекты, построенные в соответствии со спецификацией класса. Эта глава поможет вам больше узнать о свойствах класса, изучить методы проектирования классов. Описание одних свойств может показаться вам несколько упрощенным, описания других — чрезмерно усложненным. В любом случае вы сможете более глубоко понять новые свойства, если выполните примеры, приведенные в этой главе, и поэкспериментируйте с ними. Что случится, например, если использовать обычные аргументы функции вместо аргументов типа ссылок? А что будет, если в каком-либо случае не применить деструктор? Не бойтесь делать ошибки. Поверьте, вы узнаете гораздо больше полезного, выясняя природу ошибки, чем все делая правильно путем механического запоминания. (Только не надо думать, что все обстоятельства, вызванные ошибкой, пробудят в вас исключительную интуицию.) В конечном итоге вы будете вознаграждены более глубоким пониманием функций языка C++.

Данная глава начинается с обсуждения перегрузки операций, которая позволяет вам использовать стандартные операции языка C++, такие как = и +, при работе с объектами класса. Далее в ней рассматриваются так называемые дружественные программные средства, иначе говоря, механизм языка C++, обеспечивающий доступ к приватным данным для функций, не являющихся элементами класса. И наконец, в ней рассказывается о том, как заставить C++ выполнить автоматическое приведение типов применительно к классам. Как только вы освоите эти вопросы, а также материал следующей главы, то сможете убедиться, какую важную роль играют конструкторы и деструкторы классов. Кроме того, вы узнаете, какие стадии вам, возможно, придется пройти в процессе разработки и совершенствования проекта класса.

Одна из трудностей освоения языка C++, по крайней мере на той стадии изучения предмета, на которой вы находитесь в настоящий момент, заключается в том, что необходимо запомнить массу информации. При этом не следует ожидать, что вы все сможете запомнить до того, как накопите достаточно опыта, чтобы знать, на чем следует сконцентрировать свое внимание. В этом отношении изучение языка C++ можно сравнить с изучением многофункционального процессора или программы электронной таблицы. Ни одно из конкретных свойств языка нельзя назвать слишком сложным, но на практике многие пользователи фактически хорошо изучили лишь те свойства, которыми они пользуются ре-

гулярно, например, поиск нужного текста или выделение того или иного фрагмента текста курсивом. Вы, возможно, припомните, что где-то читали, как следует генерировать альтернативные символы или создавать содержание, но все эти навыки невозможно освоить иначе, чем путем ежедневного выполнения этой работы. По-видимому, наиболее подходящий метод усвоения всего богатства материала, представленного в этой главе, состоит в том, чтобы постепенно включать эти новые инструментальные средства в свои собственные программы, написанные на C++. По мере того как с приобретением опыта вам будет все легче и легче работать с этими средствами, переходите к использованию других инструментальных средств. Как сказал Бъярни Страустрап (Bjarne Stroustrup), создатель языка C++, на конференции профессиональных программистов: "Не напрягайтесь в своих отношениях с языками. Не считайте, что вы обязаны использовать все его средства, и ни в коем случае не пытайтесь применять их в первый же день".

## Перегрузка операций

Рассмотрим один из методов, который позволяет придать операциям с объектами более привлекательный вид. *Перегрузка операций* представляет собой еще один пример полиморфизма C++. В главе 8 вы видели, как C++ обеспечивает возможность определять несколько функций, имеющих одно и то же имя при условии, что у них разные сигнатуры (списки аргументов). Это и есть перегрузка функций, или функциональный полиморфизм. Он предназначен для того, чтобы предоставить вам возможность использовать одно и то же имя функции для выполнения разных и тех же базовых операций, даже если вы применяете такую операцию по отношению к различным типам данных. (Представьте себе, каким неудобным для использования окажется английский язык, если будут заданы разные формы глаголов для каждого нового типа объекта — *lift\_lft your left foot, but lift\_sp your spoon* (поднимите левую ногу и поднимите ложку).) Перегрузка операций (операторов) позволяет вам рассматривать операции (операторы) C++ сразу в нескольких смыслах. Фактически многие операции языка C++ (и языка C) уже перегружены изначально. Например, операция \*, будучи примененной к адресу, дает значение, которое хранится по этому адресу. Однако, применяя операцию \* к паре чисел, мы получаем произведение этих значений. C++ использует число и типы операндов, чтобы решить, какое действие предпринять.

C++ позволяет распространить перегрузку операций (операторов) на типы, определенные пользователем.

Благодаря этому появляется возможность, например, использовать символ + для сложения двух объектов. Опять-таки, компилятор воспользуется числом и типом операндов, чтобы определить, каким определением операции сложения воспользоваться. Перегруженные операции часто делают вид программного кода более привычным. Например, сложение двух массивов — это обычная операция при выполнении вычислений. Обычно эта операция приобретает с помощью цикла **for** следующую компактную форму:

```
// поэлементное сложение
for (int i = 0; i < 20; i++)
 evening[i] = sam[i] + janet[i];
```

Но в C++ вы можете определить класс, который представляет массивы и перегружает операцию + таким образом, что можно выполнить следующее действие:

```
// сложить два объекта типа массив
evening = sam + janet;
```

Мы выполним эту операцию в главе 12. (Почему не сейчас? Потому что требуется также перегрузить операцию [], а это несколько сложнее, чем перегрузка операции +.) Такая простая форма записи операции сложения скрывает ее механику и подчеркивает самое существенное, а это еще одна цель объектно-ориентированного программирования (ООП).

Чтобы перегрузить какую-либо операцию, лучше воспользоваться функцией специальной формы, получившей название *операторной*. Операторная функция имеет вид:

```
operator op(список_аргументов)
```

где *op* — это символ операции, подвергаемой перегрузке. Другими словами, *operator+()* перегружает операцию + (здесь *op* — это +), а *operator\*()* перегружает операцию \* (здесь *op* — это \*). Операцией *op* может быть только полноценная операция C++; для этого недостаточно ввести новое обозначение и этим ограничиться. Например, вы не можете пользоваться функцией *operator@()*, поскольку в C++ нет операции @. Но функция *operator[]()* перегрузит операцию [], так как [] — это операция, выполняющая индексацию массива. Предположим, у вас имеется класс **Salesperson**, для которого дается определение функции-элемента *operator+()*, предназначеннной для выполнения перегрузки операции +, чтобы она могла выполнять сложение цифр, отражающих данные о продаже одного объекта класса **Salesperson** (продавец), с цифрами другого объекта этого же класса. Далее, если **district2**, **sid** и **sara** — это объек-

ты класса **Salesperson**, вы можете записать такое уравнение:

```
district2 = sid + sara;
```

Компилятор, убедившись в том, что оба операнда принадлежат одному и тому же классу **Salesperson**, заменят эту операцию соответствующей операторной функцией:

```
district2 = sid.operator+(sara);
```

Эта функция использует объект **sid** неявно (поскольку она вызывает метод) и объект **sara** явно (поскольку он передается как аргумент) для вычисления суммы, которую она возвращает. Разумеется, наиболее привлекательной стороной является то, что вы можете использовать привычную запись операции **+** вместо неуклюжей формой записи функции.

C++ налагает ряд ограничений на перегрузку операций, но их гораздо легче будет понять после того, как вы рассмотрите внимательно процесс перегрузки. Итак,

разберем несколько примеров сначала для того, чтобы изучить сам процесс, а затем обсудим возможные ограничения.

## Время в нашем распоряжении

Если вы работали с отчетом Пригса в течение 2 часов 35 минут до обеда и в течение 2 часов 40 минут после обеда, то возникает вопрос, как долго вы работали над отчетом в течение всего дня? Это один из примеров, в котором понятие сложения имеет смысл, однако величины, которые вы складываете (сочетание часов и минут) не соответствуют встроенному типу данных. В главе 7 подобного рода задача решается путем объявления структуры **travel\_time** и функции **sum()**, выполняющей сложение подобных структур. Сейчас у нас появилась возможность обобщить все это в виде класса **Time**, выполняя для этой цели сложение. Начнем с обычного метода, а затем посмотрим, как вместо него использовать операцию перегрузки. В листинге 10.1 показано определение класса.

### Листинг 10.1 Программа mytime0.h.

```
// mytime0.h - класс Time перед выполнением
// перегрузки операции

#ifndef _MYTIME0_H_
#define _MYTIME0_H_
#include <iostream>
using namespace std;

class Time
{
private:
 int hours;
 int minutes;
}
```

```
public:
 Time();
 Time(int h, int m = 0);
 void AddMin(int m);
 void AddHr(int h);
 void Reset(int h = 0, int m = 0);
 Time Sum(const Time & t) const;
 void Show() const;
};

#endif
```

В рассматриваемом классе имеются методы для преобразования и возврата в исходное положение показаний времени, для отображения временных значений и для сложения двух таких значений. В листинге 10.2 представлены определения этих методов. Обратите внимание на то, как методы **AddMin()** и **Sum()** используют целочисленное деление и операцию деления по модулю для преобразований значений **minutes** и **hours**, когда количество минут превосходит значение 59.

### Листинг 10.2 Программа mytime0.cpp.

```
//mytime0.cpp - реализация методов класса Time
#include "mytime0.h"
Time::Time()
{
 hours = minutes = 0;
}
Time::Time(int h, int m)
{
 hours = h;
 minutes = m;
}
void Time::AddMin(int m)
{
 minutes += m;
 hours += minutes / 60;
 minutes %= 60;
}
void Time::AddHr(int h)
{
 hours += h;
}
void Time::Reset(int h, int m)
{
 hours = h;
 minutes = m;
}
Time Time::Sum(const Time & t) const
{
 Time sum;
 sum.minutes = minutes + t.minutes;
 sum.hours = hours + t.hours + sum.minutes / 60;
 sum.minutes %= 60;
 return sum;
}
void Time::Show() const
{
 cout << hours << " hours, " << minutes
 << " minutes";
 cout << '\n';
}
```

Рассмотрим программный код функции `Sum()`. Обратите внимание на тот факт, что аргумент представляется собой ссылку, однако возвращаемый тип не является ссылкой. Причина выбора ссылки в качестве аргумента заключается в достижении более высокой эффективности. Программный код должен обеспечить получение такого же результата, как если бы объект `Time` был передан по значению. Однако в большинстве случаев передача по ссылке обеспечивает более высокое быстродействие и более эффективное использование памяти, чем при передаче по значению.

Возвращаемое значение, однако, не может быть ссылкой. Это объясняется тем, что данная функция создает новый объект (`sum`) класса `Time`, представляющий собой сумму двух других объектов `Time`. Возврат объекта, как это имеет место в рассматриваемом программном коде, приводит к созданию копии объекта, которой может воспользоваться вызывающая функция. Однако, если возвращаемым типом будет `Time &`, ссылка будет указывать на объект `sum`. Но объект `sum` — это локальная переменная, которая уничтожается, как только функция перестает существовать, следовательно, эта ссылка будет относиться к несуществующему объекту. Вместе с тем использование возвращаемого типа `Time` означает, что программа создает копию объекта `sum`, прежде чем уничтожить его, и вызывающая функция получит эту копию.

#### ПРЕДОСТЕРЕЖЕНИЕ

Не следует возвращать ссылок на локальную переменную или на какой-либо другой временный объект.

И наконец, программа, представленная в листинге 10.3, позволяет выполнить проверку того фрагмента класса, который выполняет суммирование.

#### Листинг 10.3 Программа `usetime0.cpp`.

```
// usetime0.cpp — использует первый вариант
// класса Time
// компилирует файлы usetime0.cpp и
// mytime0.cpp в один программный модуль

#include <iostream>
#include "mytime0.h"
using namespace std;

int main()
{
 Time A;
 Time B(5, 40);
 Time C(2, 55);
 A.Show();
 B.Show();
 C.Show();
 A = B.Sum(C);
 A.Show();
 return 0;
}
```

Результаты выполнения программы:

```
0 hours, 0 minutes
5 hours, 40 minutes
2 hours, 55 minutes
8 hours, 35 minutes
```

#### Добавление операции сложения

Достаточно просто преобразовать класс `Time` таким образом, чтобы можно было пользоваться перегруженной операцией сложения. Для этого достаточно поменять имя `Sum()` на имя `operator+(())`, которое на первый взгляд выглядит несколько необычно. И в самом деле, здесь достаточно только добавить символ операции (в данном случае это `+`) сразу после ключевого слова `operator` и использовать полученный результат в качестве имени метода. Это единственное место в имени идентификатора, где вы можете использовать символ, отличный от буквы, цифры или символа подчеркивания. Листинги 10.4 и 10.5 отражают это небольшое изменение.

#### Листинг 10.4 Программа `mytime1.h`.

```
// mytime1.h — класс Time после выполнения
// перегрузки операции
#ifndef _MYTIME1_H_
#define _MYTIME1_H_
#include <iostream>
using namespace std;

class Time
{
private:
 int hours;
 int minutes;
public:
 Time();
 Time(int h, int m = 0);
 void AddMin(int m);
 void AddHr(int h);
 void Reset(int h = 0, int m = 0);
 Time operator+(const Time & t) const;
 void Show() const;
};

#endif
```

#### Листинг 10.5 Программа `mytime1.cpp`.

```
//mytime1.cpp — реализация методов класса Time
#include "mytime1.h"

Time::Time()
{
 hours = minutes = 0;
}

Time::Time(int h, int m)
{
 hours = h;
 minutes = m;
}
```

```

void Time::AddMin(int m)
{
 minutes += m;
 hours += minutes / 60;
 minutes %= 60;
}

void Time::AddHr(int h)
{
 hours += h;
}

void Time::Reset(int h, int m)
{
 hours = h;
 minutes = m;
}

Time Time::operator+(const Time & t) const
{
 Time sum;
 sum.minutes = minutes + t.minutes;
 sum.hours = hours + t.hours + sum.minutes / 60;
 sum.minutes %= 60;
 return sum;
}

void Time::Show() const
{
 cout << hours << " hours, " << minutes
 << " minutes";
 cout << '\n';
}

```

Подобно функции `Sum()`, `operator+()` вызывается объектом класса `Time`, принимает второй объект `Time` в качестве аргумента и возвращает объект `Time`. Таким образом, можно вызывать метод `operator+()`, используя для этой цели тот же синтаксис, который применялся функцией `Sum()`:

`A = B.operator+(C); // обозначение функции`

Присваивая этому методу имя `operator+()`, вы получаете также возможность пользоваться операторной формой записи:

`A = B + C; // операторная форма записи`

Этот выражение обеспечивает вызов метода `operator+()`. Обратите внимание на тот факт, что объект в левой части операции (в рассматриваемом случае это `B`) является вызывающим, а объект в правой части (в рассматриваемом случае это `C`) — это объект, переданный в качестве аргумента. Листинг 10.6 иллюстрирует этот момент.

#### Листинг 10.6 Программа usetime1.cpp.

```

// usetime1.cpp - использует второй вариант
// класса Time
// компилирует файлы usetime1.cpp и
// mytime1.cpp в один программный модуль
#include <iostream>
#include "mytime1.h"
using namespace std;

```

```

int main()
{
 Time A;
 Time B(5, 40);
 Time C(2, 55);
 A.Show();
 B.Show();
 C.Show();
 A = B.operator+(C); // функциональная
 // форма записи
 A.Show();
 B = A + C; // операторная
 // форма записи
 B.Show();
 return 0;
}

```

Результаты выполнения программы:

```

0 hours, 0 minutes
5 hours, 40 minutes
2 hours, 55 minutes
8 hours, 35 minutes
11 hours, 30 minutes

```

Итак, имя функции `operator+()` обеспечивает ее вызов путем использования как функциональной, так и операторной формы записи. Компилятор использует типы операндов, чтобы знать, что делать:

```

int a, b, c;
Time A, B, C;
c = a + b; // выполнить сложение
 // целочисленных величин
C = A + B; // выполнить сложение, опре-
 // деленное для объектов Time

```

#### Ограничения при выполнении перегрузки

Большая часть операций C++ (табл. 10.1) может быть перегружена таким же образом. Перегруженные операции (за некоторым исключением) не обязательно должны быть функциями-элементами. Однако, по меньшей мере, один из операндов должен иметь тип, определенный пользователем. Рассмотрим ограничения, которые C++ накладывает на перегрузку операций, определенных пользователем, несколько более подробно:

1. Перегруженная операция должна иметь, по меньшей мере, один операнд, имеющий тип, определенный пользователем. Это не позволит вам осуществлять перегрузку операций для стандартных типов данных. Следовательно, вы не сможете переопределить операцию вычитания (-) так, чтобы вместо разности двух значений типа `double` вычислялась их сумма. Это ограничение способствует обеспечению надежной работы программы, но в то же время может повредить творческому подходу к использованию системных ресурсов.
2. Вы не можете использовать операцию таким образом, чтобы она нарушила правила синтаксиса, кото-

рым подчиняется исходная операция. Например, невозможно перегрузить операцию деления по модулю (%) так, чтобы ее можно было использовать с одним операндом:

```
int x;
Time shiva;
% x; // недопустимо для операции
 // деления по модулю
% shiva; // недопустимо для
 // перегруженной операции
```

Аналогично вы не можете изменить приоритеты операций. Следовательно, если перегружается операция сложения в целях получения возможности складывать два класса, новая операция получает тот же приоритет, что и обычное сложение.

3. Невозможно создавать новые символы операций. Например, нельзя сделать так, чтобы функция `operator**()` обозначала возведение в степень.

4. Не подлежат перегрузке следующие операции:

|                               |                                        |
|-------------------------------|----------------------------------------|
| <code>sizeof</code>           | Операция <code>sizeof</code>           |
| <code>.</code>                | Операция принадлежности                |
| <code>*</code>                | Операция указатель на элемент класса   |
| <code>::</code>               | Операция определения диапазона доступа |
| <code>:</code>                | Условная операция                      |
| <code>typeid</code>           | Операция RTTI                          |
| <code>const_cast</code>       | Операция преобразования типа           |
| <code>dynamic_cast</code>     | Операция преобразования типа           |
| <code>reinterpret_cast</code> | Операция преобразования типа           |
| <code>static_cast</code>      | Операция преобразования типа           |

Тем не менее, все операции, приведенные ниже в табл. 10.1, допускают перегрузку.

5. Большая часть операций, представленных в табл. 10.1, может быть перегружена путем использования функций, являющихся и не являющихся элементами классов. Однако вы можете использовать только функции-элементы для перегрузки следующих операторов:

|                    |                                                         |
|--------------------|---------------------------------------------------------|
| <code>=</code>     | Оператора присваивания                                  |
| <code>()</code>    | Оператора вызова функции                                |
| <code>[]</code>    | Оператора индексации                                    |
| <code>-&gt;</code> | Доступ к элементу класса посредством операции указателя |

Таблица 10.1. Операции, которые могут быть перегружены

|                    |                     |                  |                       |                       |                         |
|--------------------|---------------------|------------------|-----------------------|-----------------------|-------------------------|
| <code>+</code>     | <code>-</code>      | <code>*</code>   | <code>/</code>        | <code>%</code>        | <code>^</code>          |
| <code>&amp;</code> | <code> </code>      | <code>~</code>   | <code>!</code>        | <code>=</code>        | <code>&lt;</code>       |
| <code>&gt;</code>  | <code>+=</code>     | <code>=</code>   | <code>*=</code>       | <code>/=</code>       | <code>%=</code>         |
| <code>^=</code>    | <code>&amp;=</code> | <code> =</code>  | <code>&lt;&lt;</code> | <code>&gt;&gt;</code> | <code>&gt;&gt;=</code>  |
| <code>&lt;=</code> | <code>==</code>     | <code>!=</code>  | <code>&lt;=</code>    | <code>&gt;=</code>    | <code>&amp;&amp;</code> |
| <code>  </code>    | <code>++</code>     | <code>--</code>  | <code>,</code>        | <code>-&gt;*</code>   | <code>-&gt;</code>      |
| <code>()</code>    | <code>[]</code>     | <code>new</code> | <code>delete</code>   | <code>new []</code>   | <code>delete []</code>  |

В дополнение к этим формальным ограничениям нужно разумно подходить к проблеме перегрузки операций. Например, не следует перегружать операцию `*` таким образом, чтобы она производила обмен элементами данных между двумя объектами `Time`. Ничего в этой записи не говорит о том, что выполняется такая операция, так что будет лучше, если конкретный метод класса будет определен с именем, несущим в себе информацию, например, `Swap()`.

## Другие перегруженные операции

Имеет смысл ввести некоторые другие операции для класса `Time`. Например, вам может понадобиться вычесть одно временное значение из другого или умножить его на некоторый множитель. Для этого требуется перегрузка операций вычитания и умножения. Метод тот же, что и примененный выше для операции сложения, — создаются методы `operator-( )` и `operator*( )`. Иначе говоря, в объявление класса следует добавить следующие прототипы:

```
Time operator-(const Time & t) const;
Time operator*(double n) const;
```

Далее в файл реализации добавьте следующие определения методов:

```
Time Time::operator-(const Time & t) const
{
 Time diff;
 int tot1, tot2;
 tot1 = t.minutes + 60 * t.hours;
 tot2 = minutes + 60 * hours;
 diff.minutes = (tot2 - tot1) % 60;
 diff.hours = (tot2 - tot1) / 60;
 return diff;
}

Time Time::operator*(double mult) const
{
 Time result;
 long totalminutes = hours * mult * 60
 + minutes * mult;
 result.hours = totalminutes / 60;
 result.minutes = totalminutes % 60;
 return result;
}
```

### ПРИМЕЧАНИЕ

Мы не рассмотрели и даже не будем рассматривать каждую операцию, указанную в списке ограничений или в таблице 10.1. Однако Приложение Е содержит все операции, которые не рассматриваются в основном тексте.

После того как эти изменения будут выполнены, можно проверить новые определения с помощью программного кода, представленного в листинге 10.7. (Здесь предполагается, что файлы модифицированных классов изменились и вместо версии `mytime1` появилась версия `mytime2`.)

#### Листинг 10.7 Программа `usetime2.cpp`.

```
// usetime2.cpp - используется третий вариант
// класса Time
// файлы usetime2.cpp и mytime2.cpp
// компилируются в единый программный модуль

#include <iostream>
#include "mytime2.h"
using namespace std;

int main()
{
 Time A;
 Time B(5, 40);
 Time C(2, 55);

 A.Show();
 B.Show();
 C.Show();

 A = B + C; // operator+()
 A.Show();
 A = B - C; // operator-()
 A.Show();
 A = B * 2.75; // operator*()
 A.Show();
 return 0;
}
```

Результаты выполнения программы:

```
0 hours, 0 minutes
5 hours, 40 minutes
2 hours, 55 minutes
8 hours, 35 minutes
2 hours, 45 minutes
15 hours, 35 minutes
```

## Использование дружественных структур

Как вы уже могли убедиться, C++ управляет доступом к приватным разделам объекта класса. Обычно общедоступные методы класса служат единственным средством доступа, однако иногда такое ограничение оказывается чрезмерно жестким и не позволяет решать некоторые конкретные проблемы в процессе создания программ. Для таких случаев C++ предусматривает другую форму доступа — *дружественные структуры*. Дружественные структуры принимают формы:

- Дружественных функций
- Дружественных классов
- Дружественных функций-элементов

Делая функцию дружественной по отношению к какому-либо классу, вы наделяете ее такими же привилегиями доступа, какими обладает функция-элемент этого класса. Мы сейчас рассмотрим дружественные функции, а другие дружественные структуры будут описаны в главе 14.

Прежде чем изучать, как создаются дружественные структуры, выясним, в силу каких причин возникает необходимость в этих средствах. Довольно часто перегрузка бинарной операции (операции, имеющей два аргумента) класса порождает необходимость в дружественных структурах. Умножение объекта `Time` на вещественное число может служить примером подобного рода ситуации, так что перейдем к ее изучению.

В примере с классом `Time` перегруженная операция умножения отличается от остальных двух перегруженных операций тем, что она работает с двумя различными типами. Иначе говоря, каждая из операций сложения и вычитания выполняется над двумя величинами типа `Time`, в то время как операция умножения использует сочетание значения `Time` со значением типа `double`. Это обстоятельство ограничивает область применения этой операции. Напомним, что левый операнд представляет собой вызывающий объект. Иначе говоря,

`A = B * 2.75;`

преобразуется в следующее обращение к функции-элементу:

`A = B.operator*(2.75);`

Что можно сказать о таком операторе?

`A = 2.75 * B;` // не соответствует
 // функции-элементу

В принципе, для `2.75 * B` результат должен быть таким же, как и для `B * 2.75`, однако первое выражение не может соответствовать функции-элементу, поскольку `2.75` не является объектом типа `Time`. Напомним, что левый операнд — это вызывающий объект, но `2.75` не является объектом. Следовательно, компилятор не может заменить это выражение обращением к функции-элементу.

Один из способов, позволяющих обойти эту трудность, заключается в том, чтобы уведомить каждого (и помнить об этом самому), что следует использовать только `B * 2.75` и ни в коем случае не `2.75 * B`. Это и есть пример дружественного отношения к пользователю, что не всегда характерно для ООП.

Существует и другая возможность — функция, не являющаяся элементом класса. (Напомним, что большая часть операций может быть перегружена с помощью функции-элемента либо посредством функции, не являющейся элементом какого-либо класса.) Функция, не

являющаяся элементом какого-либо класса, не вызывается объектом. Причина состоит в том, что любые значения, которые она использует, в том числе и объекты, являются явными аргументами. Следовательно, компилятор может сопоставить выражение

```
A = 2.75 * B; // не может соответствовать
// функцией-элементу
```

со следующим вызовом функции, не являющейся элементом какого-либо класса:

```
A = operator*(2.75, B);
```

Эта функция будет иметь такой прототип:

```
Time operator*(double m, const Time & t);
```

В функции перегруженной операции, не являющейся элементом класса, левый операнд выражения операции соответствует первому аргументу операторной функции, а правый операнд соответствует второму аргументу.

Использование функции, не являющейся элементом класса, решает проблему расстановки операндов в нужном порядке (сначала `double`, а затем `Time`), но при этом возникает новая проблема: функции, не являющиеся элементами класса, не могут осуществлять прямой доступ к приватным данным этого класса. Итак, по меньшей мере обычные функции, не являющиеся элементами соответствующего класса, не получают доступа к данным этого класса. Однако имеется специальная категория функций, не являющихся членами класса, получивших название *дружественных*, которые способны обеспечить доступ к приватным элементам класса.

## Создание дружественных конструкций

Первым шагом к созданию дружественной функции является размещение прототипа в объявлении класса, при этом объявлению предпосыпается ключевое слово `friend`:

```
friend Time operator*(double m,
const Time & t); // размещается в
// объявлении класса
```

Что касается этого прототипа, то следует отметить две его особенности:

- Несмотря на то что функция `operator*()` объявлена в объявлении класса, она не является функцией-элементом.
- Хотя функция `operator*()` не является функцией-элементом, она получает те же права доступа, что и функция-элемент.

Вторым шагом является формулировка определения функции. Поскольку такая функция не является функцией-элементом, не следует прибегать к помощи специ-

фикатора `Time::`. Кроме того, нельзя пользоваться в определении ключевым словом `friend`:

```
Time operator*(double m, const Time & t)
{
 Time result;
 long totalminutes = t.hours * mult *
 60 + t.minutes * mult;
 result.hours = totalminutes / 60;
 result.minutes = totalminutes % 60;
 return result;
}
```

При наличии такого объявления оператор

```
A = 2.75 * B;
```

преобразуется в оператор

```
A = operator*(2.75, B);
```

и при этом вызывается дружественная функция, не являющаяся функцией-элементом, которую мы только что определили.

Одним словом, функция, дружественная по отношению к какому-либо классу, не являясь его элементом, имеет те же права доступа, что и функция-элемент.

### НЕ ЯВЛЯЮТСЯ ЛИ ДРУЖЕСТВЕННЫЕ КОНСТРУКЦИИ ПРИМЕРОМ ОТХОДА ОТ ПРИНЦИПОВ ООП?

На первый взгляд может показаться, что дружественные конструкции нарушают принцип сокрытия данных ООП, поскольку механизм дружественных конструкций позволяет функциям, не являющимся элементами конкретного класса, осуществлять доступ к приватным данным этого класса. Тем не менее, такая точка зрения поверхностна. Вместо этого, считайте дружественные функции частью расширенного интерфейса класса. Например, с концептуальной точки зрения значение `Time`, умноженное на `double`, во многом совпадает со значением `double`, умноженным на `Time`. Тот факт, что в первом случае требуется дружественная функция, в то время как во втором случае умножение может быть выполнено при участии функции-элемента, является следствием особенностей синтаксиса языка C++. Пользуясь как дружественной функцией, так и тем или иным методом класса, вы можете задействовать и ту, и другую операцию с одним и тем же интерфейсом пользователя. При этом следует иметь в виду, что только определение класса может отличить дружественные функции от прочих, следовательно, за объявлением класса сохраняется право определять, каким функциям можно разрешить доступ к приватным данным. Итак, методы класса и дружественные средства – это два различных механизма, выражающие интерфейс класса.

По существу, эта конкретная дружественная функция может быть записана не как дружественная конструкция путем внесения в определение функции следующих изменений:

```
Time operator*(double m, const Time & t)
{
 return t * m; // используется
// функция-элемент
}
```

Первоначальная версия этой функции получает прямой доступ к `t.minutes` и `t.hours`, следовательно, она должна быть дружественной конструкцией. Эта версия использует только объект `t` класса `Time` как единое целое, оставляя функции-элементу манипулирование приватными значениями, так что эта версия не обязательно должна быть дружественным средством. Тем не менее, идея превратить эту версию в дружественную функцию также вполне оправдана. Самое главное, она связывает функцию таким образом, что та становится частью официального интерфейса класса. Если в дальнейшем у вас появится необходимость в том, чтобы такая функция осуществляла прямой доступ к приватным данным, достаточно изменить только определение функции, но не прототип класса.



### СОВЕТ

Если вы хотите перегрузить какую-либо операцию для того, чтобы она могла работать с классами, и использовать эту операцию в выражении, в котором первый операнд не является классом, то для изменения последовательности операндов можно воспользоваться дружественной функцией.

## Общий вид дружественной конструкции: перегрузка операции <<

Одним из наиболее полезных свойств класса является то, что можно перегрузить операцию `<<` таким образом, чтобы затем использовать ее совместно с `cout` для отображения содержимого конкретного объекта. В некотором смысле такая перегрузка является более сложной, чем рассмотренные выше примеры, в связи с этим мы выполним ее в два этапа, а не в один, как обычно.

Предположим, что `trip` — это объект класса `Time`. Чтобы отобразить значения `Time`, мы использовали функцию `Show()`. Однако, не будет ли лучше, если мы поступим таким образом:

```
cout << trip; // можно ли сделать так,
// чтобы cout распознавала класс Time?
```

Вы можете так сделать, поскольку `<<` — это одна из операций C++, которая допускает перегрузку. На самом деле, она уже и так чрезмерно перегружена. В своем основном воплощении операция `<<` в C и C++ предназначена для работы с битами; в выражении она сдвигает биты влево (приложение E). Однако класс `ostream` перегружает эту операцию, превращая ее в инструментальное средство для обработки выходных данных. Напомним, что `cout` — это объект `ostream` и что он достаточно "разумен", чтобы распознать все базовые типы данных C++. Это объясняется тем, что в объявление класса `ostream` включено определение перегруженной операции `operator<<()` для каждого базового типа. Иначе говоря, одно определение использует аргумент типа `int`, другое

— аргумент типа `double` и т.д. Следовательно, один из способов обучить `cout` распознавать объект `Time` состоит в том, чтобы добавить определение новой операции с функциями в объявление класса `ostream`. Однако изменять содержимое файла `iostream` и вмешиваться в работу стандартного интерфейса не следует. Гораздо целесообразнее использовать объявление класса `Time`, чтобы обучить класс `Time` правильно использовать `cout`.

### Первая версия перегрузки операции <<

Чтобы обучить класс `Time` правильно использовать `cout`, вам придется воспользоваться дружественной функцией. Почему? Да потому, что оператор, подобный

```
cout << trip;
```

использует два объекта, при этом объект (`cout`) класса `ostream` идет первым. Если для перегрузки операции `<<` вы используете функцию, дружественную по отношению к классу `Time`, объект `Time` следует первым, как это имело место, когда мы перегружали операцию `*` посредством функции-элемента. Это значит, чтобы вы обязались использовать операцию `<<` таким способом:

```
trip << cout; //если бы функция operator<<()
//была бы функцией-элементом класса Time
```

Это может сбить с толку. В то же время, воспользовавшись дружественной функцией, вы можете перегрузить эту операцию таким образом:

```
void operator<<(ostream & os, const Time & t)
{
 os << t.hours << " hours "
 << t.minutes << " minutes";
}
```

Это позволяет вам использовать выражение

```
cout << trip;
```

чтобы осуществлять вывод в следующем формате:

```
4 hours, 23 minutes
```

### ОПРЕДЕЛЕНИЕ ДРУЖЕСТВЕННОСТИ КОНСТРУКЦИЙ

При объявлении нового класса `Time` функцию `operator<<()` превращается в функцию, дружественную по отношению к классу `Time`. Однако эта функция, не будучи враждебной к классу `ostream`, в то же время не является дружественной по отношению к нему. Функция `operator<<()` принимает аргумент `ostream` и аргумент `Time`, так что может показаться, что эта функция должна быть дружественной по отношению к обоим классам. Но если вы просмотрите программный код этой функции, то обнаружите, что она осуществляет доступ к отдельным элементам объекта `Time`, в то же время она использует объект `ostream` как единое целое. Поскольку функция `operator<<()` осуществляет прямой доступ к элементам объекта `Time`, она должна быть дружественной по отношению к классу `Time`. Но так как рассматриваемая функция не получает прямого доступа к приватным элементам

данных объекта **ostream**, она не обязана быть дружественной по отношению к классу **ostream**. И это хорошо, поскольку вы не должны вносить поспешные изменения в определение класса **ostream**.

Обратите внимание на тот факт, что новое определение функции **operator<<()** использует ссылку **os** на класс **ostream** в качестве ее первого аргумента. Обычно **os** ссылается на объект **cout**, как это имеет место в выражении **cout << trip**. Но вы могли использовать эту операцию с другими объектами класса **ostream**, и в этом случае **os** послужила бы ссылкой на эти объекты. (Вы ничего не слышали о других объектах класса **ostream**? Напомним вам об объекте **cerr**, впервые рассмотренном в главе 9. А в главе 16, вы узнаете, как создаются новые объекты для управления выводом данных в файлы, и эти объекты могут использовать методы класса **ostream**. Затем вы можете воспользоваться определением функции **operator<<()**, чтобы записать данные класса **Time** в файлы и вывести их на экран.) Более того, при вызове **cout << trip** используется сам объект **cout**, а не его копия, так что рассматриваемая функция передает объект как ссылку, а не по значению. Таким образом, выражение **cout << trip** превращает **os** в псевдоним **cout**, а выражение **cerr << trip** превращает **os** в псевдоним **cerr**. Объект **Time** может быть передан по значению или по ссылке, поскольку и та, и другая форма делает значения этого объекта доступными для рассматриваемой функции. Кроме того, для передачи по ссылке требуется меньше памяти и времени, чем при передаче по значению.

### Вторая версия перегрузки операции <<

Реализации, подобные только что рассмотренной, порождают одну проблему. Такие операторы, как

```
cout << trip;
```

работают хорошо, однако подобная реализация не позволяет образовывать сочетание переопределенной операции << с операциями, которые обычно использует **cout**:

```
cout << "Triptime: " << trip
 << " (Tuesday)\n"; //невыполнимо
```

Чтобы понять, почему эта конструкция не работает и что нужно предпринять, чтобы заставить ее работать, нужно сначала немного больше узнать о том, как функционирует конструкция **cout**. Рассмотрим следующие операторы:

```
int x = 5;
int y = 8;
cout << x << y;
```

C++ считывает оператор вывода слева направо, воспринимая его как эквивалент следующего выражения:

```
(cout << x) << y;
```

Для операции <<, согласно ее определению в классе **iostream**, объект **ostream** является левым операндом. Ясно, что выражение **cout << x** удовлетворяет этому требованию, поскольку **cout** — это объект класса **ostream**. Но оператор вывода также требует, чтобы все выражение (**cout << x**) принимало значение типа объекта класса **ostream**, так как это выражение находится слева от << **y**. Поэтому класс **ostream** реализует функцию **operator<<()** таким образом, что она возвращает объект **ostream**. В частности, она возвращает вызывающий объект, в рассматриваемом случае — **cout**. Следовательно, выражение (**cout << x**) само является объектом **ostream** и может быть левым операндом для операции <<.

Вы можете применить такой же подход и в отношении к дружественной функции. Для этого всего лишь повторно вызовите функцию **operator<<()**, чтобы она возвращала ссылку на объект **ostream**:

```
ostream & operator<<(ostream & os,
 const Time & t)
{
 os << t.hours << " hours, "
 << t.minutes << " minutes";
 return os;
}
```

Обратите внимание на то, что возвращаемым типом будет **ostream &**. Напомним, это значит, что данная функция возвращает ссылку на объект **ostream**. Поскольку для начала программа передает этой функции ссылку на объект, конечный результат состоит в том, что значение, возвращаемое функцией, является именно тем объектом, который был ей передан. Иначе говоря, оператор

```
cout << trip;
```

становится следующим функциональным вызовом:

```
operator<<(cout, trip);
```

И в результате этого вызова возвращается объект **cout**. Таким образом, выполняется следующий оператор:

```
cout << "Trip time: " << trip
 << " (Tuesday)\n"; // так можно
```

Разобьем все это на отдельные этапы, чтобы иметь возможность подробно рассмотреть весь процесс. Сначала

```
cout << "Trip time: "
```

вызывает для конкретного **ostream** определение операции <<, которая отображает строку и возвращает объект **cout**, следовательно, выражение **cout << "Trip time: "** отображает строку, после чего оно заменяется возвращаемым значением, т.е. значением **cout**. Это позволяет получить более компактную форму исходного оператора:

```
cout << trip << " (Tuesday)\n";
```

Далее, программа использует объявление операции `<<` в классе `Time`, чтобы отобразить время поездки и еще раз возвратить объект `cout`. Это позволяет получить более компактную форму рассматриваемого оператора:

```
cout << " (Tuesday)\n";
```

В завершение программы используется определение операции `<<` для строк в классе `ostream` в целях отображения заключительной строки.

### СОВЕТ

В общем случае, чтобы выполнить перегрузку операции `<<` для отображения объекта класса `c_name`, используйте дружественную функцию с определением, данным в такой форме:

```
ostream & operator<<(ostream & os,
 const c_name & obj)
{
 os << ... ; // отображение
 // содержимого объекта
 return os;
}
```

В листинге 10.8 представлено определение класса, модифицированное таким образом, чтобы были включены две дружественные функции, `operator*()` и `operator<<()`. Оно реализует первую из них как встроенную функцию, поскольку она обладает компактным программным кодом. (Когда определением служит прототип, как в данном случае, следует употреблять префикс `friend`.)

### ПОМНИТЕ

Ключевое слово `friend` используется только в прототипе, содержащемся в объявлении класса. Оно не используется в определении функции, если только определение не является прототипом.

### Листинг 10.8 Программа mytime3.h.

```
// mytime3.h - класс Time с дружественными
// конструкциями
#ifndef _MYTIME3_H_
#define _MYTIME3_H_
#include <iostream>
using namespace std;
class Time
{
private:
 int hours;
 int minutes;
public:
 Time();
 Time(int h, int m = 0);
 void AddMin(int m);
 void AddHr(int h);
 void Reset(int h = 0, int m = 0);
 Time operator+(const Time & t) const;
 Time operator-(const Time & t) const;
 Time operator*(double n) const;
```

```
friend Time operator*(double m,
 const Time & t)
{
 return t * m; } // встроенно
// определение
friend ostream & operator<<(ostream & os,
 const Time & t);
};
```

Листинг 10.9 демонстрирует пересмотренный набор определений. Еще раз обратите внимание на тот факт, что методы используют спецификатор `Time::`, в то время как дружественные функции этого не делают.

### Листинг 10.9 Программа mytime3.cpp.

```
//mytime3.cpp - реализация методов класса Time
#include "mytime3.h"
Time::Time()
{
 hours = minutes = 0;
}
Time::Time(int h, int m)
{
 hours = h;
 minutes = m;
}
void Time::AddMin(int m)
{
 minutes += m;
 hours += minutes / 60;
 minutes %= 60;
}
void Time::AddHr(int h)
{
 hours += h;
}
void Time::Reset(int h, int m)
{
 hours = h;
 minutes = m;
}
Time Time::operator+(const Time & t) const
{
 Time sum;
 sum.minutes = minutes + t.minutes;
 sum.hours = hours + t.hours + sum.minutes / 60;
 sum.minutes %= 60;
 return sum;
}
Time Time::operator-(const Time & t) const
{
 Time diff;
 int tot1, tot2;
 tot1 = t.minutes + 60 * t.hours;
 tot2 = minutes + 60 * hours;
 diff.minutes = (tot2 - tot1) % 60;
 diff.hours = (tot2 - tot1) / 60;
 return diff;
}
```

```

Time Time::operator*(double mult) const
{
 Time result;
 long totalminutes = hours * mult * 60
 + minutes * mult;
 result.hours = totalminutes / 60;
 result.minutes = totalminutes % 60;
 return result;
}

ostream & operator<<(ostream & os, const Time & t)
{
 os << t.hours << " hours,"
 << t.minutes << " minutes";
 return os;
}

```

Листинг 10.10 представляет еще один образец программы.

#### Листинг 10.10 Программа usetime3.cpp.

```

// usetime3.cpp – использует четвертую версию
// класса Time
// Совместная компиляция usetime3.cpp и
// mytime3.cpp

#include <iostream>
#include "mytime3.h"
using namespace std;

int main()
{
 Time A;
 Time B(5, 40);
 Time C(2, 55);
 cout << A <<"; " << B << ":" "
 << C << endl;
 A = B + C; // функция operator+()
 cout << A << endl;
 A = B * 2.75; // функция operator*()
 cout << A << endl;
 cout << 10 * B << endl;
 return 0;
}

```

Результаты выполнения программы:

```

0 hours, 0 minutes; 5 hours, 40 minutes:
2 hours, 55 minutes
8 hours, 35 minutes
15 hours, 35 minutes
56 hours, 40 minutes

```

## Перегруженные операции: дружественные и обычные функции

При выполнении перегрузки операций во многих случаях вы будете поставлены перед выбором: использовать для этой цели функцию-элемент или функцию, не являющуюся таковой. Обычно вариант функции, не являющейся элементом класса, может быть представлен дру-

жественной функцией, которая осуществляет прямой доступ к приватным данным соответствующего класса. В качестве примера рассмотрим операцию сложения для класса Time. У нее есть прототип в объявлении класса Time:

```
// вариант с использованием функции-элемента
Time operator+(const Time & t) const;
```

Вместо этого, класс мог бы воспользоваться следующим прототипом:

```
// вариант с использованием функции,
// не являющейся членом класса
friend Time operator+(const Time & t1,
 const Time & t2);
```

Операция сложения требует двух operandов. В случае использования функции-элемента один operand передается неявно с помощью указателя **this**, а второй — явно как аргумент функции. Если речь идет о дружественной функции, оба опаранда передаются как аргументы.

#### ПОМНИТЕ

Вариант перегрузки операторной функции, не являющейся элементом класса, требует на один аргумент больше, чем вариант с использованием для той же операции функции-элемента.

Любой из этих двух прототипов соответствует выражению **B + C**, в котором **B** и **C** являются объектами класса Time. Иначе говоря, компилятор может преобразовать оператор

```
A = B + C;
```

в какой-либо из двух следующих:

```
A = B.operator+(C); // функция-элемент
A = operator+(B, C); // не является
 // функцией-элементом
```

Имейте в виду, что обязательно следует выбрать какую-либо одну из форм при определении заданной операции, но не обе сразу. Поскольку обе формы соответствуют одному и тому же выражению, определение одновременно обеих форм рассматривается как неопределенность.

Тогда какой из этих форм следует отдать предпочтение? Как уже отмечалось, для некоторых операций функции-элементы являются единственным правильным выбором. Во всех остальных случаях большой разницы в выборе форм нет. В некоторых случаях для определенных конструкций класса функция, не являющаяся элементом этого класса, может оказаться предпочтительнее, особенно если для данного класса даны определения преобразований типов. Далее в этой главе будет рассмотрен соответствующий пример.

## Перегрузка: класс Vector

Рассмотрим еще один класс, в котором используется перегрузка операций и дружественные инструментальные средства, в частности, класс, представляющий векторы. На примере этого класса будут показаны дальнейшие аспекты проектирования классов, такие как внедрение двух различных способов описания одной и той же в виде объекта. Если вас николько не интересуют векторы, можете использовать многие из новых методов в других целях. Вектор в инженерном деле и в физике — это величина, имеющая значение (модуль) и направление. Например, если вы толкнете какой-нибудь предмет, то результат этого действия будет зависеть от того, насколько сильным был толчок (величина модуля) и в каком направлении он был выполнен. Например, если толкнуть падающую вазу в одном направлении, то можно предотвратить ее падение, в то время как толчок в противоположном направлении ускорит ее печальный конец. Чтобы как можно более точно описать передвижение автомобиля, нужно указать скорость (величина модуля) и направление. Ваши оправдания в конфликте с дорожным патрулем, сводящиеся к тому, что вы не превышали допустимой скорости, не убедят его в вашей правоте, если вы нарушили правила дорожного движения. (Специалисты в области иммунобиологии и вычислительной техники могут употреблять понятие вектора в разных смыслах, однако пока мы не будем останавливаться на этом. В главе 15 рассматривается версия этого понятия, используемая в вычислительной технике.) Приводимое ниже замечание пополнит ваш багаж знаний о векторах, однако для понимания всех аспектов C++, раскрывающихся в последующих примерах, во всестороннем изучении векторов нет необходимости.

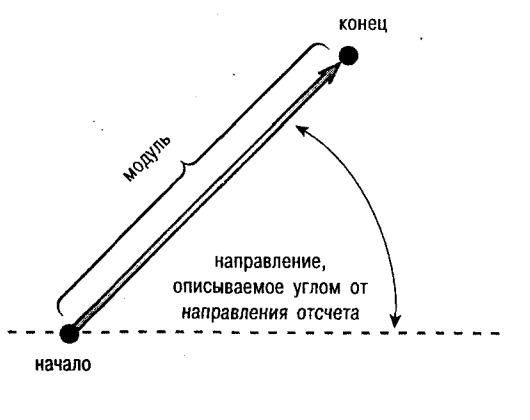


РИСУНОК 10.1 Описание перемещений с помощью вектора.

## ВЕКТОРЫ

Многие физические величины характеризуются модулем и направлением. Результат толчка, например, зависит как от его силы, так и от направления: Для перемещения объекта на экране компьютера также нужно знать расстояние и направление. Вы можете описывать подобного рода явления с помощью векторов. Например, можно описать движение (перемещение) объекта на экране посредством вектора, представленного в виде стрелки, проведенной из исходной позиции в нужную точку. Длина вектора представляет его модуль, она показывает, как далеко была перемещена точка. Ориентация стрелки показывает направление (рис. 10.1). Вектор, который показывает подобное изменение положения точки, называется вектором смещения.

Сложение двух векторов получило простую интерпретацию. Сначала проведите один вектор. Затем проведите второй вектор, исходящий из конца стрелки первого вектора. И наконец, начертите вектор из исходной точки первого вектора в конечную точку второго вектора. Этот третий вектор является суммой первых двух (рис. 10.2). Обратите внимание на то, что длина суммарного вектора меньше суммы длин векторов-слагаемых.

Векторы представляют собой естественный выбор для выполнения перегрузки операций. Во-первых, вы не можете представить вектор только одним числом, так что есть смысл создать класс, представляющий векторы. Во-вторых, в векторном исчислении существуют аналоги обычным арифметическим операциям, таким как сложение и вычитание. Эта параллель позволяет найти анализ перегрузки соответствующих операций, так чтобы вы смогли применить перегруженные операции к векторам.

Чтобы упростить дело, мы будем рассматривать двумерные векторы, такие как перемещения по экрану, вместо трехмерных векторов, которые способны представить перемещение вертолета или гимнаста. Для описания двумерного вектора достаточно всего лишь двух чисел, но вы сами должны определить, какие наборы из двух чисел выбрать:

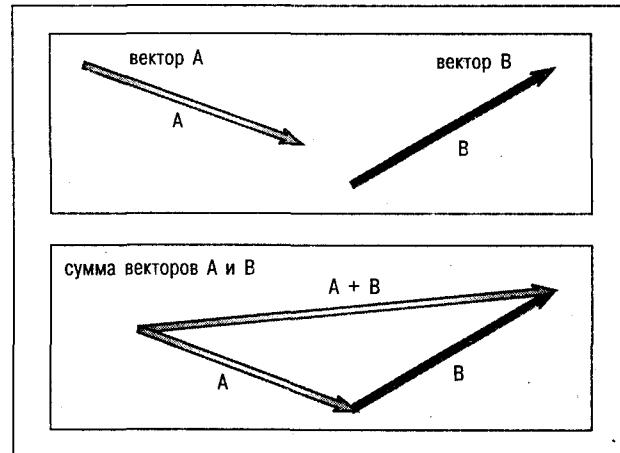


РИСУНОК 10.2 Сложение двух векторов.

- Можно описать вектор с помощью его модуля (длины) и направления (угла).
- Можно представить вектор его компонентами  $x$  и  $y$ .

Эти компоненты представляют собой горизонтальный вектор (компонент  $x$ ) и вертикальный вектор (компонент  $y$ ), сумма которых и составляет искомый вектор. Например, вы можете описать движение как перемещение точки на 30 единиц вправо и на 40 единиц вверх (рис. 10.3). При таком перемещении точка будет помещена в то же место, что и при перемещении на 50 единиц под углом  $53,1^\circ$  к горизонту. Поэтому вектор с модулем 50 и с углом  $53,1^\circ$  эквивалентен вектору, имеющему горизонтальный компонент, равный 30, и вертикальный компонент, равный 40 единицам. В случае перемещения векторов имеет значение только то, откуда начинается и где кончается это перемещение, а не маршрут, выбранный для перемещения из исходной в конечную точку. Выбор представления в основном аналогичен рассматриваемому в главе 7 в программе, которая выполняла преобразование прямоугольных координат в полярные и наоборот.

В одних случаях удобно пользоваться координатной формой, в других — полярной, так что мы встроим обе формы представления векторов в описание класса. (примечание "Множественные представления и классы" далее в этой главе.) Кроме того, мы спроектируем класс таким образом, что, если вы измените одно представление вектора, объект автоматически изменит другое представление. Способность встраивать в объект подобного рода элементы интеллекта является еще одним достоинством классов C++. В листинге 10.11 представлено определение класса.

Обратите внимание на тот факт, что четыре функции, которые отображают значения компонентов, определены в объявлении классов. Это автоматически превращает их во встроенные функции. Тот факт, что программные коды этих функций такие короткие, делает их превосходными кандидатами на роль встроенных функций. Ни одна из них не должна подвергать

изменениям данные объекта, поэтому они были объявлены с использованием спецификатора **const**. Как было отмечено в главе 9, таковы требования синтаксиса к объявлению функции, которая не будет вносить изменения в объект, к которому она имеет явный доступ.

В программе, представленной в листинге 10.12, показаны все методы и дружественные функции, объявленные в программе, представленной в листинге 10.11. Обратите внимание на то, как функции конструктора и функция **set()** обеспечивают прямоугольное и полярное представление конкретного вектора. Следовательно, и тот, и другой набор значений немедленно оказывается в вашем распоряжении без дальнейших вычислений, как только в этом возникает необходимость. Кроме того, как указано в главах 4 и 7, математические функции, встроенные в C++, используют значения углов в радианах, в силу этого обстоятельства указанные функции предусматривают преобразования углов из радианов в градусы и наоборот в виде соответствующих методов. Подобная реализация скрывает от пользователя такие операции, как преобразование полярных координат в прямоугольные или перевод радиан в градусы. Все, что нужно знать пользователю, — это то, что класс использует углы в градусах, и это дает возможность использовать векторы в двух эквивалентных представлениях.

Такие проектные решения следуют традициям ООП, согласно которым интерфейс класса сосредоточивается на самом существенном (абстрактная модель), в то время как детали остаются в тени. Таким образом, когда вы используете класс **Vector**, вы можете думать только об основных свойствах векторов, таких как их способность представлять перемещения и возможность сложения двух векторов. Вопрос о том, в какой форме представить вектор, в записи с помощью компонентов или с помощью модуля и направления, отодвигается на задний план, так как вы можете задать вектор и представить его в том формате, который больше всего подходит на текущий момент.

Далее мы рассмотрим некоторые свойства этого класса более подробно.

 **ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**  
Некоторые системы все еще используют файл **math.h**, а не файл **cmath**. Кроме того, некоторые системы C++ не выполняют автоматического просмотра библиотеки математических функций. Например, некоторые системы UNIX требуют от вас выполнения следующих действий:

```
$ CC source_file(s) -lm
```

Опция **-lm** дает редактору связей команду искать библиотеку математических функций. Следовательно, когда в конце концов вы приступите к компиляции программ, использующих класс **Vector**, и получите сообщение о том, что не определены внешние файлы, попытайтесь использовать опцию **-lm** либо проверьте, требуется ли для вашей системы что-либо подобное.

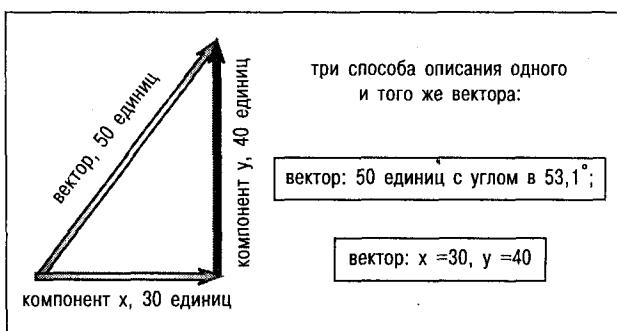


РИСУНОК 10.3 Компоненты  $x$  и  $y$  вектора.

**Листинг 10.11 Класс vector.h.**

```
// vector.h - класс векторов с операцией <<, выбор режима
#ifndef _VECTOR_H_
#define _VECTOR_H_

class Vector
{
private:
 double x; // значение по горизонтали
 double y; // значение по вертикали
 double mag; // длина вектора
 double ang; // направление вектора
 char mode; // 'r' = прямоугольная, 'p' = полярная

 // приватные методы установки значений
 void set_mag();
 void set_ang();
 void set_x();
 void set_y();
public:
 Vector();
 Vector(double n1, double n2, char form = 'r');
 void set(double n1, double n2, char form = 'r');
 ~Vector();
 double xval() const { return x; } // отображение значения x
 double yval() const { return y; } // отображение значения y
 double magval() const { return mag; } // отображение модуля
 double angval() const { return ang; } // отображение угла
 void polar_mode(); // установить mode в 'p'
 void rect_mode(); // установить mode в 'r'

 // перегрузка операций
 Vector operator+(const Vector & b) const;
 Vector operator-(const Vector & b) const;
 Vector operator-() const;
 Vector operator*(double n) const;
 // дружественные конструкции
 friend Vector operator*(double n, const Vector & a);
 friend ostream & operator<<(ostream & os, const Vector & v);
};
#endif
```

**Листинг 10.12 Программа vector.cpp.**

```
// vector.cpp - методы класса Vector
#include <iostream>
#include <cmath>
using namespace std;
#include "vector.h"
const double Rad_to_deg = 57.2957795130823;

// приватные методы
// вычисляет модуль по значениям x и y
void Vector::set_mag()
{
 mag = sqrt(x * x + y * y);
}

void Vector::set_ang()
{
 if (x == 0.0 && y == 0.0)
 ang = 0.0;
 else
 ang = atan2(y, x);
}
```

```

// вычисляет x по полярным координатам
void Vector::set_x()
{
 x = mag * cos(ang);
}

// вычисляет y по полярным координатам
void Vector::set_y()
{
 y = mag * sin(ang);
}

// общедоступные методы
Vector::Vector() // конструктор, заданный по умолчанию
{
 x = y = mag = ang = 0.0;
 mode = 'r';
}

// создает вектор на базе прямоугольных координат, если form является r
// (по умолчанию) или на базе полярных координат, если form является p
Vector::Vector(double n1, double n2, char form)
{
 mode = form;
 if (form == 'r')
 {
 x = n1;
 y = n2;
 set_mag();
 set_ang();
 }
 else if (form == 'p')
 {
 mag = n1;
 ang = n2 / Rad_to_deg;
 set_x();
 set_y();
 }
 else
 {
 cout << "Incorrect 3rd argument to Vector() - ";
 cout << "vector set to 0\n";
 x = y = mag = ang = 0.0;
 mode = 'r';
 }
}

// создается вектор на базе прямоугольных координат, если form является r
// (по умолчанию) или на базе полярных координат, если form является p
void Vector:: set(double n1, double n2, char form)
{
 mode = form;
 if (form == 'r')
 {
 x = n1;
 y = n2;
 set_mag();
 set_ang();
 }
 else if (form == 'p')
 {
 mag = n1;
 ang = n2 / Rad_to_deg;
 set_x();
 set_y();
 }
 else

```

```

{
 cout << "Incorrect 3rd argument to Vector() - ";
 cout << "vector set to 0\n";
 x = y = mag = ang = 0.0;
 mode = 'r';
}
}

Vector::~Vector() // деструктор
{
}

void Vector::polar_mode() // устанавливает режим полярных координат
{
 mode = 'p';
}

void Vector::rect_mode() // устанавливает режим прямоугольных координат
{
 mode = 'r';
}

// перегрузка операции
// сложение двух объектов Vector
Vector Vector::operator+(const Vector & b) const
{
 return Vector(x + b.x, y + b.y);
}

// вычитание объекта b класса Vector из объекта a
Vector Vector::operator-(const Vector & b) const
{
 return Vector(x - b.x, y - b.y);
}

// изменяет знак объекта Vector на противоположный
Vector Vector::operator-() const
{
 return Vector(-x, -y);
}

// умножение вектора на n
Vector Vector::operator*(double n) const
{
 return Vector(n * x, n * y);
}

// дружественные методы
// умножает n на объект a класса Vector a
Vector operator*(double n, const Vector & a)
{
 return a * n;
}

// отображает прямоугольные координаты, если mode является r,
// отображает полярные координаты, если mode является p
ostream & operator<<(ostream & os, const Vector & v)
{
 if (v.mode == 'r')
 os << "(x,y) = (" << v.x << ", " << v.y << ")";
 else if (v.mode == 'p')
 {
 os << "(m,a) = (" << v.mag << ", "
 << v.ang * Rad_to_deg << ")";
 }
 else
 os << "Vector object mode is invalid";
 return os;
}

```

## Использование элементов состояния

В классе хранятся как прямоугольные, так и полярные координаты вектора. Этот класс использует элемент данных с именем `mode`, чтобы иметь возможность управлять тем, какую форму используют конструктор, метод `set()` и перегруженная операция `operator<<()`. При этом '`r`' представляет режим прямоугольных координат (по умолчанию), а '`p`' — полярные координаты. Подобный элемент получил название *элемента состояния*, поскольку он описывает состояние, в котором находится объект. Чтобы получить представление о том, что это означает, рассмотрим программный код конструктора:

```
Vector::Vector(double n1, double n2, char form)
{
 mode = form;
 if (form == 'r')
 {
 x = n1;
 y = n2;
 set_mag();
 set_ang();
 }
 else if (form == 'p')
 {
 mag = n1;
 ang = n2 / Rad_to_deg;
 set_x();
 set_y();
 }
 else
 {
 cout << "Incorrect 3rd argument
 ^to Vector() -- ";
 cout << "vector set to 0\n";
 x = y = mag = ang = 0.0;
 mode = 'r';
 }
}
```

Если третий аргумент равен '`r`' или если он не указан (по умолчанию прототип присваивает значение '`r`'), входные данные интерпретируются как прямоугольные координаты, в то время как значение '`p`' обеспечивает интерпретацию входных данных как полярных координат:

```
// Установка значений x = 3, y = 4
Vector folly(3.0, 4.0);

// Установка значений mag = 20, ang = 30
Vector foolery(20.0, 30.0);
```

Обратите внимание на то, что конструктор использует приватные методы `set_mag()` и `set_ang()`, чтобы установить значения модуля и угла, если даны значения `x` и `y`, а приватные методы `set_x()` и `set_y()` — чтобы установить значения `x` и `y`, если даны значения модуля и угла. Заметьте также, что конструктор выдает предупреждающее сообщение и устанавливает состояние в '`r`', если указано значение, отличное от '`r`' или '`p`'.

Аналогично функция `operator<<()` использует режим для выбора способа отображения значений:

```
// отображаются прямоугольные координаты,
// если режим соответствует 'r',
// если режим соответствует 'p', то
// отображаются полярные координаты
ostream & operator<<(ostream & os,
 const Vector & v)
{
 if (v.mode == 'r')
 os << "(x,y) = (" << v.x
 << ", " << v.y << ")";
 else if (v.mode == 'p')
 {
 os << "(m,a) = (" << v.mag << ", "
 << v.ang * Rad_to_deg << ")";
 }
 else
 os << "Vector object mode is invalid";
 return os;
}
```

Различные методы, посредством которых можно установить режим, спроектированы таким образом, что воспринимают как правильные только значения '`r`' и '`p`', следовательно, заключительная конструкция `else` в таких функциях не выполняется. Тем не менее, никогда не вредно предпринять защитные меры; такие меры позволяют обнаружить программные ошибки, которые другими способами трудно обнаружить.

## МНОЖЕСТВЕННЫЕ ПРЕДСТАВЛЕНИЯ И КЛАССЫ

Широко используются количественные величины, имеющие, хотя и различные, но в то же время эквивалентные значения. Например, вы можете оценить потребление бензина в галлонах на милю, как это делают в Соединенных Штатах, или в литрах на 100 км, как это принято в Европе. Можно представить число в виде строки символов или в числовой форме, а интеллект — в виде значения IQ (Intelligence Quotient — Коэффициент умственного развития). Классы сами по себе прекрасно приспособлены для того, чтобы объединять различные аспекты и представления того или иного явления в единый объект. Во-первых, можно хранить различные представления в одном объекте. Во-вторых, можно создавать функции класса таким образом, чтобы отображение значений в одном представлении автоматически приводило к отображению значений в другом представлении (представлениях). Например, метод `set_by_polar()` класса `Vector` присваивает элементам `mag` и `ang` значения аргументов функции, но он также устанавливает значения переменных `x` и `y`. То, что все преобразования выполняются внутри классов, позволяет воспринимать ту или иную количественную величину в ее естественном виде, а не в виде ее абстрактного представления.

## Еще немного о перегрузке

Сложение двух векторов — очень простая задача, если речь идет о координатах `x`, `y`. Достаточно сложить два компонента `x`, чтобы получить результирующий компонент `x`, и два компонента `y`, чтобы получить результи-

рующий компонент *y*. При ознакомлении с этим описанием у вас, возможно, возникнет желание воспользоваться следующим программным кодом:

```
Vector Vector::operator+(const Vector & b)
{
 Vector sum;
 sum.x = x + b.x;
 sum.y = y + b.y;
 return sum;
}
```

И это вполне оправдано, если объект хранит только компоненты *x* и *y*. К сожалению, этот вариант программного кода не способен установить значения полярных координат. Эта проблема решается путем добавления дополнительного программного кода:

```
Vector Vector::operator+(const Vector & b)
{
 Vector sum;
 sum.x = x + b.x;
 sum.y = y + b.y;
 sum.set_ang(sum.x, sum.y);
 sum.set_mag(sum.x, sum.y);
 return sum;
}
```

Однако гораздо проще и надежнее поручить эту работу конструктору:

```
Vector Vector::operator+(const Vector & b)
{
 // возвращает вновь построенный объект
 // класса Vector
 return Vector(x + b.x, y + b.y);
}
```

В рассматриваемом случае программный код передает конструктору объекта **Vector** два новых значения компонентов *x* и *y*. Затем конструктор создает новый безымянный объект, использующий эти значения, а функция возвращает копию этого объекта. Таким образом, вы гарантируете, что новый объект класса **Vector** создан в соответствии со стандартизованными правилами, которые были заложены в конструктор.

### СОВЕТ

Если метод должен вычислить новый объект класса, выясните, можно ли воспользоваться услугами конструктора, чтобы выполнить эту работу. В этом случае вы не только избавитесь от неприятностей — такой подход гарантирует, что новый объект будет построен должным образом.

## Умножение

В наглядном представлении при умножении вектора на скалярное число вектор удлиняется или укорачивается во столько раз, сколько определяет величина множите-

ля. Таким образом, умножение некоторого вектора на 3 приводит к появлению вектора, который в 3 раза пре-восходит по длине исходный, но сохраняет то же направление. Нетрудно осуществить преобразование полярных координат (операцию умножения). В полярных координатах умножается на коэффициент только модуль, а угол остается прежним. В прямоугольных координатах умножение на число состоит в том, что компоненты *x* и *y* у вектора умножаются на это число по отдельности. Иначе говоря, если компонентами вектора являются 5 и 12, то при умножении на 3 получаются значения компонентов, равные 15 и 36. Именно эти действия и выполняет перегруженная операция умножения:

```
Vector Vector::operator*(double n) const
{
 return Vector(n * x, n * y);
}
```

Что касается перегруженной операции сложения, то этот код дает конструктору возможность создавать корректный класс **Vector** по новым значениям компонентов *x* и *y*. Это позволяет выполнять умножение величины типа **double** на значение **Vector**. Как и в примере с **Time**, мы можем воспользоваться дружественной встроенной функцией для получения вектора **Vector**, на значение **double**, превосходящее исходное:

```
Vector operator*(double n,
 const Vector & a) //дружественная функция
{
 return a * n; //умножает вектор Vector
 //на коэффициент double
}
```

### Некоторые уточнения: перегрузка и перегруженная операция

Уже в обычном языке С операция вычитания имеет два смысла. Во-первых, если она используется с двумя операндами, то это операция вычитания. Операция вычитания относится к категории **бинарных**, поскольку в ней участвуют два и только два операнда. Во-вторых, когда она используется с одним операндом, как, например, **-x**, то это является операцией умножения на **-1**. Такая операция относится к категории **унарных**, это значит, что для ее выполнения требуется один операнд. Обе операции — вычитание и перемена знака — применимы и к векторам, так что в классе **Vector** они реализованы обе.

Чтобы вычесть вектор *B* из вектора *A*, достаточно просто выполнить вычитание соответствующих компонентов, так что определение перегрузки вычитания во многом аналогично определению перегрузки сложения:

```
Vector operator-(const Vector & b) const;
 // прототип
Vector Vector::operator-(const Vector & b)
{
 const // определение
```

```
{
 // возвращает созданный Vector
 return Vector(x - b.x, y - b.y);
}
```

В данном случае важно сохранить правильный порядок. Рассмотрим следующий оператор:

```
diff = v1 - v2;
```

Он преобразуется в вызов функции-элемента:

```
diff = v1.operator-(v2);
```

Это означает, что вектор, переданный как явный аргумент, вычитается из неявного векторного аргумента, следовательно, нужно воспользоваться `x - b.x`, но не `b.x - x`.

Далее рассмотрим унарную операцию отрицания (`-1`), для которой требуется один операнд. Применение этой операции к обычному числу, например, к `-x`, приводит к изменению знака этого числа. Следовательно, применение этой операции к вектору приводит к изменению знака каждой его компоненты. Точнее, функция должна возвратить новый вектор, обратный по отношению к исходному. (В полярных координатах замена знака на обратный означает, что модуль вектора не меняется, зато меняется направление — на противоположное. В реальной жизни многие политики, мало понимающие или совсем не разбирающиеся в математике, тем не менее, основываясь на собственной интуиции, мастерски владеют этой операцией.) Предлагаем прототип и определение перегруженного отрицания:

```
Vector operator-() const;
Vector Vector::operator-() const
{
 return Vector(-x, -y);
}
```

Теперь обратите внимание на то, что здесь мы имеем два отдельных определения функции `operator-()`. Все правильно, поскольку эти два определения имеют различные сигнатуры. Вы можете начать с определения как для бинарной, так и для унарной версии операции отрицания, поскольку в C++ реализована и та, и другая версия этой операции. Операция, имеющая только бинарную форму, такая как, например, деление (`/`), может быть перегружена только как бинарная операция.

### ПОМНИТЕ

Поскольку перегрузка операций реализуется с помощью функций, можно перегружать одну и ту же операцию многократно, пока каждая операторная функция имеет собственную, отличную от других сигнатуру, и пока каждая операторная функция имеет то же число operandов, что и соответствующая встроенная операция языка C++.

## Комментарий к реализации

Рассматриваемая реализация сохраняет как прямоугольные, так и полярные координаты вектора для объекта. Тем не менее, общедоступный интерфейс не зависит от этого факта. Все, что должен обеспечить интерфейс, — это возможность отображения обоих представлений вектора и возвращение индивидуальных значений. Внутреннее представление векторов может существенно отличаться. Например, объект может сохранять только компоненты `x` и `y`. Допустим, что метод `magval()`, который возвращает величину модуля вектора, может вычислять модуль по значениям `x` и `y`, а не просто производить поиск соответствующего значения, которое хранится в объекте. Подобный подход изменяет реализацию, в то время как пользовательский интерфейс остается неизменным. Такое отделение интерфейса от реализации представляет собой одну из целей ООП. Он позволяет вам производить точную настройку реализации, не изменяя программного кода, используемого этим классом.

У каждой из этих реализаций есть свои преимущества и недостатки. Сохранение данных означает, что объекту отводится большее пространство памяти и что всякий раз, когда объект `Vector` подвергается изменениям, следует соблюдать особую осторожность при внесении изменений как в прямоугольное, так и в полярное представление. Но просмотр данных выполняется быстрее. Если тому или иному приложению требуется доступ к обоим представлениям вектора, то предпочтительнее реализация, используемая в данном примере. Если же представление в полярных координатах используется не часто, то более подходящей будет другая реализация. Вы можете воспользоваться одной реализацией в одной программе, другой реализацией — во второй программе, но сохранять один и тот же пользовательский интерфейс в обоих случаях.

## Применение класса Vector к решению задачи случайного блуждания

В листинге 10.13 представлена короткая программа, использующая усовершенствованные классы. Она моделирует известную задачу блуждания пьяницы (Drunkard's Walk). По существу, в наше время, когда пьяница считается человеком, которого одолевают проблемы со здоровьем, а не источником увеселения окружающих, эта задача обычно называется задачей случайного блуждания. Идея заключается в том, что вы ставите кого-то у фонарного столба. Субъект начинает ходить туда-сюда, но при этом на каждом шагу изменяет направление. Одна из формулировок этой задачи может звучать так: сколько шагов потребуется субъекту, чтобы удалиться от

столба, скажем, на 50 футов? В векторной форме эта задача может быть сформулирована так: сколько нужно сложить случайно ориентированных векторов, чтобы их сумма превысила 50 футов?

Программа, представленная в листинге 10.13, позволяет задать расстояние, которое нужно пройти в таком режиме, и длину шага не совсем трезвого путника. Она определяет его текущее положение после каждого шага (представленное соответствующим вектором) и сообщает, какое число шагов потребуется, чтобы преодолеть заданное расстояние, а также конечное положение путника (в обоих форматах представления). Как вы сможете убедиться сами, достижения путника не впечатляют. Сделав тысячу шагов в два фута каждый, он удалился от исходной точки всего лишь на 50 футов. Программа производит деление значения абсолютного удаления от точки старта (50 футов в рассматриваемом случае) на количество шагов, чтобы показать, насколько эффективен этот вид путешествий. Чтобы задавать случайные направления, программа использует стандартные функции `rand()`, `srand()` и `time()`, описание которых приводится в следующем разделе. Напомним, что листинг 10.12 компилируется вместе с листингом 10.13.

### Листинг 10.13 Программа randwalk.cpp.

```
// randwalk.cpp - используется класс Vector
// компилировать вместе с vector.cpp file
#include <iostream>
#include <cstdlib> // прототипы функций
 // rand(), srand()
#include <ctime> // прототип функции time()
using namespace std;
#include "vector.h"

int main()
{
 srand(time(0)); //генератор случайных чисел
 //с начальным значением
 double direction;
 Vector step;
 Vector result(0.0, 0.0);
 unsigned long steps = 0;
 double target;
 double dstep;
 cout << Enter target distance (q to quit): ";
 while (cin >> target)
 {
 cout << "Enter step length: ";
 if (!(cin >> dstep))
 break;
 while (result.magval() < target)
 {
 direction = rand() % 360;
 step.set(dstep, direction, 'p');
 result = result + step;
 steps++;
 }
 }
}
```

```
cout << "After " << steps
 << " steps, the subject "
 "has the following location:\n";
cout << result << "\n";
result.polar_mode();
cout << " или\n" << result << "\n";
cout << "Average outward distance per step = "
 << result.magval()/steps << "\n";
steps = 0;
result.set(0.0, 0.0);
cout << "Enter target distance (q to quit): ";
}
cout << "Bye!\n";
return 0;
}
```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Вам придется воспользоваться файлом `stdlib.h` вместо `cstdlib` и файлом `time.h` вместо `ctime`.

Результаты выполнения программы:

```
Enter target distance (q to quit): 50
Enter step length: 2
After 253 steps, the subject has the
following location:
(x,y) = (46.1512, 20.4902)
or
(m,a) = (50.495, 23.9402)
Average outward distance per step = 0.199587
Enter target distance (q to quit): 50
Enter step length: 2
After 951 steps, the subject has the
following location:
(x,y) = (-21.9577, 45.3019)
or
(m,a) = (50.3429, 115.8593)
Average outward distance per step = 0.0529362
Enter target distance (q to quit): 50
Enter step length: 1
After 1716 steps, the subject has the
following location:
(x,y) = (40.0164, 31.1244)
or
(m,a) = (50.6956, 37.8755)
Average outward distance per step = 0.0295429
Enter target distance (q to quit): q
Bye!
```

Случайная природа этого процесса обуславливает значительные отличия между попытками, даже если начальные условия не меняются. В результате уменьшения шага наполовину увеличивается число шагов, необходимых для прохождения заданного расстояния, в среднем в 4 раза. Согласно результатам теории вероятности, число шагов ( $N$ ) длины  $s$ , необходимых для удаления от начальной точки на расстояние  $D$ , задается формулой:

$$N = (D/s)^2$$

Но это всего лишь среднее значение, возможны значительные расхождения результатов разных попыток.

Например, тысяча попыток преодолеть расстояние 50 футов шагами длиной по 2 фута дают среднее значение, равное 636 шагам (достаточно близкое к значению 625, полученному на основании теоретических расчетов), но при этом разброс составляет от 91 до 3951 шага. Аналогично тысяча попыток преодолеть расстояние 50 футов шагами длиной по 1 футу дают среднее значение, равное 2557 шагам (близкое к значению 2500 шагов, полученному на основании теоретических расчетов) при разбросе от 345 до 10882. Так что, если вы попадете в режим случайного блуждания, не теряйте присутствия духа и удлините свой шаг. Вы, естественно, не сможете выбрать нужное направление, зато, по крайней мере, куда-нибудь да придет.

### Примечания к программе

Сначала поговорим о случайных числах. Библиотека С стандарта ANSI, которая также поставляется вместе с C++, содержит функцию `rand()`, которая возвращает случайное целое число в диапазоне от 0 до значения, которое зависит от реализации. Наша программа использует операцию вычисления модуля, чтобы получить значение угла в диапазоне от 0 до 359. Функция `rand()` работает по принципу применения соответствующего алгоритма к начальному числу для получения случайного числа. Полученное случайное число служит начальным числом при следующем вызове этой же функции и т.д. Фактически полученные числа являются *псевдослучайными*, так как десять последовательных обращений обычно дают тот же набор случайных чисел (точное значение зависит от реализации). Однако функция `srand()` предоставляет вам возможность отказаться от начального числа, заданного по умолчанию, и инициализировать другую последовательность случайных чисел. Рассматриваемая программа использует значение, возвращаемое функцией `time(0)` для задания начального числа. Функция `time(0)` возвращает текущее календарное время, которое часто представлено в виде числа секунд, прошедших с момента некоторой конкретной даты. (Собственно говоря, функция `time()` берет адрес переменной типа `time_t`, присваивает значение времени этой переменной и возвращает ее. Использование значения 0 в качестве адресного аргумента позволяет избежать использования переменной `time_t`, бесполезной во всех других отношениях.) Таким образом, оператор

```
srand(time(0));
```

каждый раз, когда вы запускаете программу, использует новое начальное значение, благодаря чему случайное генерируемое число приобретает еще более случайный характер. Заголовочный файл `cstdlib` (раньше он носил имя `stdlib.h`) содержит прототипы функций `srand()` и

`rand()`, в то время как `ctime` (ранее `time.h`) содержит прототип функции `time()`.

Рассматриваемая программа использует вектор `result` для хранения сведений о достижениях путника. На каждой итерации внутреннего цикла программа устанавливает для вектора `step` новое направление и прибавляет его к текущему значению вектора `result`. Как только модуль вектора `result` превысит заданное значение расстояния, цикл прерывается.

Установив векторный режим, программа отображает заключительное положение в прямоугольных и полярных координатах.

Между прочим, выполнение оператора

```
result = result + step;
```

приводит к тому, что представление '`r`' устанавливается независимо от исходных представлений векторов `result` и `step`. И вот почему. Функция, реализующая операцию сложения, создает и возвращает новый вектор, представляющий собой сумму двух этих аргументов. Эта функция создает такой вектор, используя конструктор, заданный по умолчанию, а тот, в свою очередь, строит векторы в представлении '`r`'. Следовательно, результирующий вектор представлен в режиме '`r`'. По умолчанию присвоение значений производится каждому элементу отдельно, поэтому '`r`' присваивается переменной `result.mode`. Если вы предпочитаете другую линию поведения, например, чтобы `result` сохранял первоначальное представление, можете перекрыть назначения, заданные по умолчанию, дав соответствующее определение операции присваивания для рассматриваемого класса. В следующей главе приводятся примеры такого рода.

## Автоматические преобразования и приведение типов для классов

При изучении свойств классов невозможно обойти вниманием тему преобразования типов. Мы рассмотрим, как C++ осуществляет преобразование стандартных типов в типы, определенные пользователем, и наоборот. Но сначала выясним, как C++ решает проблему преобразования его встроенных типов. Когда вы употребляете в программе оператор, присваивающий значение одного стандартного типа переменной другого стандартного типа, C++ автоматически преобразует значение в тип принимающей переменной при условии, что оба типа совместимы. Например, все приводимые ниже операции присваивания производят преобразования числовых типов:

```
//значение 8 типа int преобразуется в тип long
long count = 8;
```

```
// значение 11 типа int преобразуется в тип double
double time = 11;

// значение 3.33 типа double преобразуется
// в величину 3 типа int
int side = 3.33;
```

Эти операции присваивания работают успешно, поскольку C++ признает, что различные числовые типы представляют один и тот же объект, а именно число, и поскольку в C++ реализованы встроенные правила, обеспечивающие выполнение таких преобразований. Однако напомним, что в результате таких преобразований может быть утрачена заданная точность (см. главу 3). Например, присваивание значения 3.33 переменной типа `int` приводит к тому, что мы получим значение 3, а дробная часть (0.33) будет потеряна.

Язык C++ не выполняет автоматического преобразования несовместимых типов. Например, оператор

```
int * p = 10; // тип clash
```

не может быть выполнен, так как в его левой части фигурирует тип указатель, в то время как в правой части указано число. И несмотря на то что в памяти компьютера адрес представлен в виде целого числа, целые числа и указатели в концептуальном плане существенно различаются. Например, невозможно возвести указатель в квадрат. Тем не менее, там, где не выполняется автоматическое преобразование типов, можно воспользоваться приведением типов:

```
int * p = (int *) 10; // правильно, p и
// (int *) 10 являются указателями
```

Этот оператор устанавливает указатель к адресу 10 путем приведения типа 10 к типу указатель на `int` (то есть, к типу `int *`).

Вы можете определить класс, который сравнительно близок к базовому типу, или другой класс, благодаря которому становится целесообразным преобразование одной формы в другую. В этом случае можно дать C++ указания, как выполнять такие преобразования — автоматически или посредством приведения типов. Чтобы показать, как все это осуществляется на практике, переделаем программу, переводящую фунты в стоуны из главы 3, введя в употребление классы. Прежде всего подберем подходящий тип. По существу, мы представляем одну и ту же физическую величину (вес) двумя способами (в фунтах и стоунах). Класс представляет собой прекрасное средство, позволяющее включить два представления в один объект. В связи с этим имеет смысл встроить оба представления в один и тот же класс, а затем разработать соответствующие методы для представления веса в разных формах. В листинге 10.14 представлен заголовок такого класса.

#### Листинг 10.14 Заголовок stonewt.h.

---

```
// stonewt.h - определение класса Stonewt
#ifndef _STONEWT_H_
#define _STONEWT_H_
class Stonewt
{
private:
 enum {lbs_per_stn = 14}; //число фунтов в стоуне
 int stone; //всего стоунов
 double pds_left; //дробное число фунтов
 double pounds; //общий вес в фунтах
public:
 // конструктор для фунтов double
 Stonewt(double lbs);

 // конструктор для стоунов, lbs
 Stonewt(int stn, double lbs);

 // конструктор, заданный по умолчанию
 Stonewt();
 ~Stonewt();

 // представить вес в фунтах
 void show_lbs() const;

 // представить вес в стоунах
 void show_stn() const;
};

#endif
```

---

Обратите внимание на то обстоятельство, что в классе имеется три конструктора. Они позволяют выполнить инициализацию объекта `Stonewt` числом фунтов, представленным в форме с плавающей точкой, или сочетанием стоунов и фунтов. Вы можете также создать объект `Stonewt`, не инициализируя его.

Класс включает две функции отображения. Одна из них отображает вес в фунтах, а другая — в стоунах и фунтах. В листинге 10.15 показана реализация методов класса. Обратите внимание на то, что каждый конструктор присваивает значения всем трем приватным элементам. Следовательно, при создании объекта `Stonewt` автоматически устанавливаются оба представления веса.

Как было показано в главе 9, `enum` представляет собой удобный способ определения специфичных для конкретного класса констант при условии, что это целые числа.

#### Листинг 10.15 Программа stonewt.cpp.

---

```
#include <iostream>
using namespace std;
#include "stonewt.h"
// создает объект Stonewt на базе значения
double
Stonewt::Stonewt(double lbs)
{
 stone = int (lbs) / lbs_per_stn;
 // целочисленное деление
 pds_left = int (lbs) % lbs_per_stn
 + lbs - int(lbs);
 pounds = lbs;
}
```

---

```

// создает объект Stonewt на базе
// значений stone и double
Stonewt::Stonewt(int stn, double lbs)
{
 stone = stn;
 pds_left = lbs;
 pounds = stn * Lbs_per_stn + lbs;
}

Stonewt::Stonewt() // конструктор по
 // умолчанию, wt = 0
{
 stone = pounds = pds_left = 0;
}

Stonewt::~Stonewt() // деструктор
{
}

// показывает вес в стоунах
void Stonewt::show_stn() const
{
 cout << stone << " stone, "
 << pds_left << " pounds\n";
}

// показывает вес в фунтах
void Stonewt::show_lbs() const
{
 cout << pounds << " pounds\n";
}

```

Поскольку объект **Stonewt** представляет только одно значение веса, имеет смысл предусмотреть способы преобразования целого значения или значения с плавающей точкой в объект **Stonewt**. Но мы уже это делали! В C++ любой конструктор, принимая одиночный аргумент, действует как шаблон преобразования значения, имеющего тип этого аргумента, в тип класса. Следовательно, конструктор

```

// шаблон преобразования значения
// double в Stonewt
Stonewt(double lbs);

```

служит правилом, согласно которому производится преобразование значения типа **double** в значение типа **Stonewt**. Иначе говоря, вы можете создавать программный код, подобный следующему:

```

// создание объекта Stonewt
Stonewt myCat;

// использование Stonewt(double) для
// преобразования 19.6 в Stonewt
myCat = 19.6;

```

Рассматриваемая программа использует конструктор **Stonewt(double)**, чтобы создать временный объект **Stonewt**, воспользовавшись значением 19.6 для инициализации. Затем процедура поэлементного присваивания скопирует содержимое этого временного объекта в **myCat**. Этот процесс получил название неявного преобразования, поскольку он производится автоматически, без необходимости явного приведения типов.

Только конструктор, который может быть употреблен с одним-единственным аргументом, может быть использован как функция преобразования. У конструктора

```
Stonewt(int stn, double lbs);
```

два аргумента, следовательно, он не может быть использован для преобразования типов.

Возможность использования конструктора в качестве функции автоматического преобразования типов представляется довольно-таки удобным средством. Однако по мере приобретения опыта программисты, работающие на C++, обнаруживают, что автоматический режим не всегда желателен, так как он может привести к непредсказуемым преобразованиям. В связи с этим в последних версиях C++ появилось новое ключевое слово **explicit**, позволяющее отключать автоматический режим. Иначе говоря, вы можете объявить конструктор таким способом:

```
explicit Stonewt(double lbs); // неявное
 // преобразование не допускается
```

Эта конструкция блокирует неявное преобразование, аналогичное рассмотренному в предыдущем примере, и в то же время не препятствует выполнению явных преобразований, т.е. преобразований, при которых используется явное приведение типов:

```

// создать объект Stonewt
Stonewt myCat;

// не допускается, если Stonewt(double)
// объявлен как явный
myCat = 19.6;

// правильно, это явное преобразование
myCat = Stonewt(19.6);

// правильно, старая форма явного
// приведения типов
myCat = (Stonewt) 19.6;

```

### ПОМНИТЕ

Конструктор C++ с одним аргументом определяет преобразование типа этого аргумента в тип класса. Если конструктор уточняется посредством ключевого слова **explicit**, то он используется только для явных преобразований; в противном случае он применяется и для неявных преобразований.

Когда компилятор может использовать функцию **Stonewt(double)**? Если ключевое слово **explicit** используется в объявлении функции **Stonewt(double)**, эта функция может быть использована только для явного приведения типов; в противном случае она будет использована для осуществления следующих неявных преобразований:

- Когда инициализируется объект **Stonewt** значением типа **double**.

- Когда присваивается значение типа **double** объекту **Stonewt**.
- Когда передается значение типа **double** функции, в качестве аргумента которой выступает объект **Stonewt**.
- Когда функция, объявление которой предусматривает возвращение значения типа **Stonewt**, делает попытку возвратить значение **double**.
- Когда в любой из перечисленных выше ситуаций используется встроенный тип, который однозначно может быть преобразован в тип **double**.

Рассмотрим последний случай подробнее. Процесс сопоставления аргументов, обеспечиваемый прототипированием функций, приводит к тому, что конструктор **Stonewt(double)** действует как преобразователь других числовых типов. Другими словами, оба приводимых ниже оператора сначала преобразуют **int** в **double**, а затем используют конструктор **Stonewt(double)**:

```
// использует Stonewt(double),
// преобразуя int в double
Stonewt Jumbo(7000);

// использует Stonewt(double),
// преобразуя int в double
Jumbo = 7300;
```

В то же время этот двухступенчатый процесс преобразования работает только в тех случаях, когда имеется однозначный выбор. Так, если рассматриваемый класс объявил конструктор **Stonewt(long)**, компилятор отвергнет эти операторы и при этом укажет, что **int** можно преобразовать как в **long**, так и в **double**, следовательно, обращение неоднозначно.

В листинге 10.16 используются конструкторы рассматриваемого класса для инициализации некоторых объектов **Stonewt** и для выполнения преобразований типов. Не забывайте, что листинг 10.15 следует компилировать вместе с листингом 10.16.

#### Листинг 10.16 Программа stone.cpp.

```
// stone.cpp - преобразования, заданные
// пользователем
// Компиляция вместе с stonewt.cpp

#include <iostream>
using namespace std;
#include "stonewt.h"
void display(Stonewt st, int n);
int main()
{
 // использует конструктор для инициализации
 Stonewt pavarotti = 260;
 // то же, что и Stonewt wolfe = 285.7;
 Stonewt wolfe((double)285.7);
 Stonewt taft(21, 8);
```

```
cout << "The tenor weighed ";
pavarotti.show_stn();
cout << "The detective weighed";
wolfe.show_stn();
cout << "The President weighed ";
taft.show_lbs();

// использует конструктор для инициализации
pavarotti = double(265.8);

taft = 325;//то же, что и taft=Stonewt(325);
cout << "After dinner the tenor weighed ";
pavarotti.show_stn();
cout << "After dinner the President weighed ";
taft.show_lbs();
display(taft, 2);
cout << "The wrestler weighed even more.\n";
display(422, 2);
cout << "No stones left unearned\n";
return 0;
}

void display(Stonewt st, int n)
{
 for (int i = 0; i < n; i++)
 {
 cout << "Wow! ";
 st.show_stn();
 }
}
```

Результаты выполнения программы:

```
The tenor weighed 18 stone, 8 pounds
The detective weighed 20 stone, 5.7 pounds
The President weighed 302 pounds
After dinner, the tenor weighed 18 stone,
 13.8 pounds
After dinner, the President weighed 325 pounds
Wow! 23 stone, 3 pounds
Wow! 23 stone, 3 pounds
The wrestler weighed even more.
Wow! 30 stone, 2 pounds
Wow! 30 stone, 2 pounds
No stone left unearned
```

#### Примечания к программе

Прежде всего обратите внимание на тот факт, что, когда конструктор имеет единственный аргумент, при инициализации объекта класса можно воспользоваться такой формой:

```
// синтаксис для инициализации объекта
// класса при использовании конструктора
// с одним аргументом
Stonewt pavarotti = 260;
```

Это эквивалентно двум другим формам, которые использовались раньше:

```
// стандартные синтаксические формы для
// инициализации объектов класса
Stonewt pavarotti(260);
Stonewt pavarotti = Stonewt(260);
```

Тем не менее, две последние формы могут быть также использованы в случаях, когда конструкторы включают несколько аргументов.

Обратите внимание на следующие два оператора присваивания из листинга 10.16:

```
pavarotti = 265.8;
taft = 325;
```

В первом операторе присваивания для приведения величины 265.8 к значению типа **Stonewt** используется конструктор с аргументом типа **double**. Этот оператор устанавливает значение элемента **pounds** объекта **pavarotti** равным 265.8. Поскольку он использует конструктор, при этом присваивании устанавливаются значения элементов **stone** и **pds\_left** рассматриваемого класса. Аналогично второй оператор присваивания преобразует значение типа **int** в величины типа **double** и использует **Stonewt(double)** для установки значений всех трех указанных элементов.

И наконец, рассмотрите на следующее обращение к функции:

```
// преобразует 422 сначала в double,
// а затем в Stonewt
display(422, 2);
```

Прототип функции **display()** показывает, что первым ее аргументом должен быть объект **Stonewt**. Столкнувшись с аргументом типа **int**, компилятор осуществляет поиск конструктора **Stonewt(int)**, чтобы преобразовать величину типа **int** в желанный тип **Stonewt**. Не найдя такого конструктора, компилятор ищет конструктор с некоторым другим встроенным типом, в который можно было бы преобразовать тип **int**. Конструктор **Stonewt(double)** вполне подходит для этой цели. Таким образом, компилятор преобразует **int** в **double**, а затем использует **Stonewt(double)** для преобразования результата в объект **Stonewt**.

## Функции преобразования

Программа, представленная в листинге 10.16, преобразует число в объект **Stonewt**. А можете ли вы решить обратную задачу? Другими словами, можете ли вы преобразовать объект **Stonewt** в значение типа **double**, как это сделано в следующем фрагменте программы?

```
Stonewt wolfe(285.7);
double host = wolfe; //?? возможно ли это??
```

Да, вы сможете это сделать, но без использования конструкторов. Конструкторы обеспечивают преобразование различных типов данных только в тип класса. Чтобы сделать обратное, необходимо воспользоваться в C++ специальной формой операторной функции, получившей название *функции преобразования*.

Функции преобразования во многом сходны с приведением типов, определяемым пользователем, и нужно пользоваться ими так, как вы используете приведение типов. Например, если вы определяете функцию, выполняющую преобразование объекта **Stonewt** в значение типа **double**, можете осуществить следующее преобразование:

```
Stonewt wolfe(285.7);
// синтаксис #1
double host = double (wolfe);
// синтаксис #2
double thinker = (double) wolfe;
```

Другой вариант — можно предоставить компилятору решать самому, что делать:

```
Stonewt wells(20, 3);
//неявное использование функции преобразования
double star = wells;
```

Компьютер, обнаружив, что правая часть оператора представляет собой тип **Stonewt**, а левая часть — тип **double**, пытается выяснить, не объявляли ли вы функцию преобразования, соответствующую этому описанию.

Итак, как вы создаете функцию преобразования? Чтобы преобразовать тип **имяТипа**, воспользуйтесь такой формой функции преобразования:

```
operator имяТипа();
```

Обратите внимание, каким требованиям должна отвечать функция преобразования:

- Быть методом некоторого класса.
- Не задавать возвращаемый тип.
- Не иметь аргументов.

Например, функция преобразования типа **double** должна иметь такой прототип:

```
operator double();
```

Часть **имяТипа()** указывает тип возвращаемого значения. Тот факт, что функция является методом класса, означает, что она должна быть вызвана конкретным объектом класса, который и укажет функции, каким образом нужно преобразовывать тип. Следовательно, такой функции вовсе не нужны аргументы.

Чтобы получить функцию, выполняющую преобразование объектов **stone\_wt** в тип **int** и в тип **double**, требуется добавление в объявление классов следующих прототипов:

```
operator int();
operator double();
```

В листинге 10.17 представлено скорректированное объявление класса.

**Листинг 10.17 Класс stonewt1.h.**

```
// stonewt1.h - скорректированное определение класса Stonewt
#ifndef _STONEWT1_H_
#define _STONEWT1_H_
class Stonewt
{
private:
 enum { Lbs_per_stn = 14}; // число фунтов в стоуне
int stone; // всего стоунов
 double pds_left; // дробное число фунтов
 double pounds; // общий вес в фунтах
public:
 Stonewt(double lbs); // построение на базе double pounds
 Stonewt(int stn, double lbs); // построение на базе stone, lbs
 Stonewt(); // конструктор, заданный по умолчанию
 ~Stonewt();
 void show_lbs() const; // показывает вес в фунтах
 void show_stn() const; // показывает вес в стоунах
// функции преобразования
 operator int() const;
 operator double() const;
} ;
#endif
```

В листинге 10.18 представлены определения двух функций преобразования; эти определения должны быть добавлены в файл функции-элемента рассматриваемого класса. Обратите внимание на то обстоятельство, что каждая функция возвращает требуемое значение, даже если возвращаемый тип не объявлен. Заметьте также, что определение преобразования типа `int` предусматривает округление до ближайшего целого, а не отбрасывание дробной части. Например, если значение переменной `pounds` равно 114.4, то `pounds + 0.5` равно 114.9, а `int (114.9)` равно 114. Но если `pounds` равно 114.6, то `pounds + 0.5` равно 115.1, а `int (115.1)` равно 115.

**Листинг 10.18 Программа stonewt1.cpp.**

```
// stonewt1.cpp - методы класса Stonewt +
// функции преобразования
#include <iostream>
using namespace std;
#include "stonewt1.h"

// Прежние определения помещаются в этом месте

// Функции преобразования
Stonewt::operator int() const
{
 return int (pounds + 0.5);
}

Stonewt::operator double() const
{
 return pounds;
}
```

Программа, представленная в листинге 10.19, производит проверку новых функций преобразования. Оператор присваивания в программе использует неявное пре-

образование, в то время как завершающий оператор `cout` использует явное приведение типов. Напомним, что листинг 10.18 компилируется вместе с листингом 10.19.

**Листинг 10.19 Программа stonewt1.cpp.**

```
// stonewt1.cpp - функции преобразования,
// определенные пользователем
// компилировать вместе с stonewt1.cpp
#include <iostream>
using namespace std;
#include "stonewt1.h"

int main()
{
 Stonewt poppins(9.2, 8); // 9 стоунов 2,8 фунта
 double p_wt = poppins; // неявное преобразование
 cout << "Convert to double => ";
 cout << "Poppins: " << p_wt << " pounds.\n";
 cout << "Convert to int => ";
 cout << "Poppins: " << int (poppins)
 << " pounds.\n";
 return 0;
}
```

Приводим результаты выполнения этой программы; они демонстрируют преобразование объекта типа `Stonewt` в тип `double` и тип `int`:

```
Convert to double => Poppins: 128.8 pounds.
Convert to int => Poppins: 129 pounds.
```

**Автоматическое выполнение преобразования типов**

В последнем примере `int (poppins)` используется совместно с `cout`. Предположим, что вместо этого пропущено явное приведение типов:

```
cout << "Poppins: " << poppins << " pounds.\n";
```

Будет ли теперь программа выполнять неявное преобразование типов, как в следующем операторе?

```
double p_wt = poppins;
```

Нет. В примере с `p_wt` из контекста следует, что `poppins` должна быть преобразована в тип `double`. Однако в примере с `cout` нет никаких указаний, позволяющих определить, следует ли проводить преобразование в тип `int` или в тип `double`. Сталкиваясь с подобного рода нехваткой информации, компилятор предупредит, что вы пытаетесь выполнить неоднозначное преобразование. Ничего в рассматриваемом операторе не указывает на то, какой тип использовать.

Интересен тот факт, что, если класс определяет только функцию преобразования `double`, компилятор признает наш оператор правильным. Это объясняется тем, что возможно только одно преобразование, т.е. неопределенности в этом случае нет места.

Такая же ситуация имеет место и в случае присваивания. В условиях имеющихся объявлений классов компилятор отвергнет приводимый ниже оператор как неоднозначный:

```
long gone = poppins; //неоднозначный оператор
```

В C++ вы можете присвоить переменной типа `long` как значение `int`, так и значение `double`, так что компилятор на законных основаниях может выполнить любое из этих преобразований. Компилятор не хочет брать на себя ответственность за выбор преобразования. Однако если вы удалите одну из двух рассматриваемых функций преобразования, компилятор признает этот оператор правильным. Например, предположим, что не было указано определение `double`. В этом случае компилятор выполнит преобразование `int`, чтобы перевести `poppins` в значение типа `int`. Затем он преобразует значение `int` в значение типа `long` во время его присваивания переменной `gone`.

Когда класс определяет два и большее число преобразований, вы все еще можете выполнять явное приведение типов, чтобы указать, какую функцию преобразования следует применять. Вы можете пользоваться любой из приводимых записей оператора приведения типов:

```
// выполняется преобразование double
long gone = (double) poppins;
// выполняется преобразование int
long gone = int (poppins);
```

Первый оператор преобразует вес `poppins` в значение типа `double`, и затем оператор присваивания преобразует значение типа `double` в значение типа `long`. Аналогично второй оператор преобразует `poppins` сначала в тип `int`, а затем в тип `long`.

Подобно конструкторам преобразований, функции преобразований могут оказаться недостаточно эффективными. Проблема, возникающая при использовании функций, которые выполняют автоматические неявные преобразования, заключается в том, что они могут осуществлять такие преобразования тогда, когда вы их не ждете. Предположим, например, что вы написали такой программный код:

```
int ar[20];
...
Stonewt temp(14, 4);
...
int Temp = 1;
...
cout << ar[temp] << "\n"; //использован
//temp вместо Temp
```

Обычно вы рассчитываете на то, что компилятор сам обнаружит такие ошибки, как использование объекта вместо целого числа, служащего в качестве индекса массива. Однако класс `Stonewt` даст определение `operator int()`, следовательно, объект `temp` класса `Stonewt` будет преобразован в `int` со значением 200 и будет использован как индекс массива. Из этого можно сделать следующий вывод: часто имеет смысл выполнять явное преобразование и исключить возможность выполнения неявного преобразования. Ключевое слово `explicit` не работает с функциями преобразования, однако все, что от вас требуется, — это заменить функции преобразования на другую функцию, которая выполняет ту же задачу, но только в том случае, когда вызывается явно. Таким образом, вы можете заменить

```
Stonewt::operator int()
{ return int (pounds + 0.5); }
```

на

```
int Stonewt::Stone_to_Int()
{ return int (pounds + 0.5); }
```

В этом случае недопустим оператор

```
int plb = poppins;
```

однако, если вам действительно требуется выполнить преобразование, допускается следующее:

```
int plb = poppins.Stone_to_Int();
```

Но неявные функции преобразования следует использовать осторожно. Зачастую функция, допускающая только явный вызов, более предпочтительна.



#### ПРЕДОСТЕРЕЖЕНИЕ

Проявляйте осторожность при работе с функциями преобразования. Во многих случаях следует отдавать предпочтение функциям, которые допускают только явный вызов.

Подводя некоторые итоги, отметим следующие типы преобразований для классов, которые могут быть использованы в C++:

- Конструктор класса с одним аргументом служит инструкцией для выполнения преобразования значения, имеющего тип аргумента, в тип класса. Например, конструктор класса **Stonewt** с аргументом типа **int** вызывается автоматически, когда вы присваиваете значение типа **int** объекту класса **Stonewt**. Однако употребление ключевого слова **explicit** в объявлении конструктора позволяет устраниТЬ неявные преобразования, допуская только явные преобразования.
- Специальная функция-элемент класса, называемая функцией преобразования, служит в качестве инструкции для выполнения преобразования объектов классов в некоторые другие типы. Функция преобразования является элементом класса, она не имеет объявленных возвращаемых значений, не имеет аргументов и называется **operator имяТипа()**, где **имяТипа** является типом, в который преобразуется объект. Эта функция преобразования вызывается автоматически, когда вы присваиваете объект класса переменной этого типа либо применяете операцию приведения типа к этому типу.

## Преобразования и дружественные конструкции

Внесем дополнение в класс **Stonewt**. Как мы уже отмечали при рассмотрении класса **Time**, вы можете использовать любую функцию-элемент или дружественную функцию для выполнения перегрузки сложения. (Чтобы упростить дело, предположим, что ни одной функции преобразования, подобной **operator double()**, не было объявлено.) Вы можете реализовать сложение посредством следующих функций-элементов.:

```
Stonewt Stonewt::operator+(const Stonewt
 & st) const
{
 double pds = pounds + st.pounds;
 Stonewt sum(pds);
 return sum;
}
```

Можно также реализовать сложение в виде дружественной функции следующим образом:

```
Stonewt operator+(const Stonewt & st1,
 const Stonewt & st2)
{
 double pds = st1.pounds + st2.pounds;
 Stonewt sum(pds);
 return sum;
}
```

Любая из указанных форм позволяет выполнять следующее:

```
Stonewt jennySt(9, 12);
Stonewt bennySt(12, 8);
Stonewt total;
total = jennySt + bennySt;
```

Кроме того, при наличии конструктора **Stonewt(double)** каждая такая форма позволяет выполнять следующее:

```
Stonewt jennySt(9, 12);
double kennyD = 176.0;
Stonewt total;
total = jennySt + kennyD;
```

Но только дружественная функция дает возможность выполнять следующее:

```
Stonewt jennySt(9, 12);
double pennyD = 146.0;
Stonewt total;
total = pennyD + jennySt;
```

Чтобы знать, почему так происходит, преобразуйте каждую операцию сложения в соответствующий вызов функции. Прежде всего,

```
total = jennySt + bennySt;
```

становится

```
total = jennySt.
operator+(bennySt); // функция-элемент
```

либо

```
// дружественная функция
total = operator+(jennySt, bennySt);
```

В каждом из этих случаев фактические типы аргументов соответствуют формальным аргументам. Кроме того, функция-элемент вызывается так, как требуется, — с помощью объекта **Stonewt**.

Далее,

```
total = jennySt + kennyD;
```

становится

```
// функция-элемент
total = jennySt.operator+(kennyD);
```

или иначе

```
// дружественная функция
total = operator+(jennySt, kennyD);
```

Опять-таки, вызывается функция-элемент, как и требуется, с помощью объекта **Stonewt**. На этот раз в каждом случае один аргумент имеет тип **double**, этот факт служит причиной вызова конструктора **Stonewt(double)**, осуществляющего преобразование этого аргумента в объект **Stonewt**.

Между прочим, наличие определения функции-элемента **operator double()** создает в этом случае некоторую путаницу, так как появляется еще одна возможность для интерпретации. Вместо того, чтобы преобразовывать **kennyD** в **double** и выполнять сложение объектов

**Stonewt**, компилятор преобразует **jennySt** в **double** и выполняет сложение величин типа **double**. Слишком большое число функций преобразования создает неопределенность. И наконец,

```
total = pennyD + jennySt;
```

становится

```
// дружественная функция
total = operator+(pennyD, jennySt);
```

В данном случае оба аргумента имеют тип **double**, благодаря чему вызывается конструктор **Stonewt(double)** для их преобразования в объекты **Stonewt**. Однако функция-элемент не может быть вызвана.

```
total = pennyD.operator+(jennySt);
// не имеет смысла
```

Причина в этом случае состоит в том, что только объект класса может вызвать функцию-элемент. C++ не будет делать попыток преобразовать **pennyD** в объект **Stonewt**. Преобразования производятся над аргументами функции-элемента, а не над объектами, осуществляющими вызов функций-элементов.

Из всего сказанного выше следует вывод: определение сложения как дружественной конструкции облегчает программе возможность приспособиться к автоматическому преобразованию типов. Причина заключается в том, что оба операнда становятся аргументами функции, т.е. речь теперь пойдет о прототипировании функции для обоих операндов.

## Осуществление выбора

Когда вы хотите сложить величины типа **double** с величинами типа **Stonewt**, в вашем распоряжении имеется две возможности. Первая, которую мы только что описали в общих чертах, предусматривает определение **operator+(const Stonewt &, const Stonewt &)** как дружественной функции и использование конструктора **Stonewt(double)** для преобразования аргументов типа **double** в аргументы типа **Stonewt**.

Вторая возможность заключается в дальнейшей перегрузке операции сложения функциями, которые явно используют один аргумент типа **double**:

```
Stonewt operator+(double x); //функция-элемент
friend Stonewt operator+(double x, Stonewt & s);
```

Таким образом, оператор

```
total = jennySt + pennyD; //Stonewt + double
точно соответствует функции-элементу operator+(double
x), а оператор
```

```
total = pennyD + jennySt; //double + Stonewt
точно соответствует дружественной функции
operator+(double x, Stonewt &s). Ранее мы делали нечто
подобное при умножении объектов Vector.
```

Каждый вариант имеет свои преимущества. В случае применения первого варианта размеры программы меньше, поскольку вы определяете меньшее число функций. Отсюда также следует, что разработка программы потребует от вас меньше усилий, к тому же снижается вероятность возникновения ошибок. Недостаток этого варианта заключается в больших затратах времени и памяти на вызов конструктора преобразования, когда в этом возникнет необходимость. Второй вариант — это зеркальное отображение первого. Он требует от вас написания большего количества ваших собственных программных кодов, что приводит к увеличению размера программ, зато их быстродействие выше.

Если ваша программа часто использует операцию сложения значений типа **double** с объектами **Stonewt**, то, возможно, имеет смысл перегрузить сложение, чтобы действовать в таких случаях напрямую. Если программа использует такой вид сложения от случая к случаю, то проще положиться на автоматическое преобразование или, если вы соблюдаете осторожность, на явное преобразование.

## Резюме

В данной главе обсуждаются многие важные аспекты определения и использования классов. Некоторые вопросы могут показаться вам сложными, пока вы не приобретете опыт, достаточный для их понимания. Итак, подведем итоги изучения материала, представленного в настоящей главе.

Обычно единственной возможностью доступа к приватным элементам класса является использование соответствующего метода класса. C++ делает это ограничение менее жестким благодаря применению дружественных функций. Чтобы сделать функцию дружественной, объягите функцию в разделе объявлений и предложите этому объявлению ключевое слово **friend**.

C++ расширяет понятие перегрузки на операции, предоставляя пользователю возможность давать определения специальных операторных функций, которые описывают, какое отношение имеют конкретные операции к конкретным классам. Операторная функция может быть функцией-элементом класса или дружественной функцией. (Только небольшое число операций могут быть функциями-элементами того или иного класса.) C++ предоставляет вам возможность вызывать операторную функцию либо путем обращения к этой функции, либо путем использования перегруженной операции с сохранением ее обычного синтаксиса. Операторная функция для операции *op* принимает вид:

**operator*op*(список аргументов)**

*Список аргументов* представляет операнды заданной операции. Если операторная функция является функ-

цией-элементом некоторого класса, то ее первый операнд — это вызывающий объект, который не является частью списка аргументов. Например, мы перегрузили сложение, дав определение функции-элемента `operator+()` для класса `Vector`. Если `up`, `right` и `result` — три вектора, то вы можете использовать любой из приведенных ниже операторов для сложения векторов:

```
result = up.operator+(right);
result = up + right;
```

При рассмотрении второго варианта тот факт, что операнды `up` и `right` принадлежат типу `Vector`, подсказывает C++, что надо использовать определение операции сложения, данное для класса `Vector`.

Когда операторная функция принадлежит к числу функций-элементов, то первый оператор является объектом, вызывающим эту функцию. В предыдущих операторах, например, объект `up` является вызывающим объектом. Если вы хотите определить какую-либо операторную функцию, такую, чтобы первый операнд не был объектом какого-либо класса, вы должны прибегнуть к использованию дружественной функции. Затем вы сможете передать операнды в определение функции в любом устраивающем вас порядке.

Одна из самых обычных задач перегрузки операций — это определение операции `<<`, которое допускало бы ее использование в сочетании с объектом `cout` для отображения содержимого объекта. Когда требуется, чтобы объект `ostream` был первым операндом, следует определить операторную функцию как дружественную. Чтобы переопределенную таким образом операцию можно было конкатенировать с самой собой, нужно создать возвращаемый тип `ostream &`. Приводим обобщенную форму, удовлетворяющую этим требованиям:

```
ostream & operator<<(ostream & os,
 const c_name & obj)
{
 os << ... ; // отобразить
 // содержимое объекта
 return os;
}
```

Однако если в классе имеются методы, возвращающие значения элементов данных, которые вы намерены отобразить, можно применить эти методы, а не прибегать к прямому доступу посредством `operator<<()`. В таком случае эта функция может не относиться (и не должна относиться) к разряду дружественных.

C++ позволяет вам установить преобразования в тип класса и из типа класса. В первом случае конструктор любого класса с одним аргументом действует как функция преобразования, превращающая значения типа аргумента в значение типа класса. C++ вызывает конструктор автоматически, когда вы присваиваете некото-

рому объекту значение типа аргумента. Например, предположим у вас имеется класс `String` с конструктором, который принимает значение типа `char *` в качестве единственного аргумента. Если `bean` представляет собой объект `String`, вы можете воспользоваться таким оператором:

```
// преобразует тип char * в тип String
bean = "pinto";
```

Если вы предпошлете объявлению конструктора ключевое слово `explicit`, то этот конструктор может быть использован только в случае явных преобразований:

```
// явное преобразование типа char * в тип
// String
bean = String("pinto");
```

Чтобы привести тип класса к другому типу, нужно дать определение функции преобразования, сопроводив его инструкциями о том, как выполнять эти преобразования. Функция преобразования должна быть функцией-элементом. Если речь идет о преобразовании в тип `имяТипа`, оно должно иметь следующий прототип:

```
operator имяТипа();
```

Обратите внимание на тот факт, что этот прототип не должен иметь никакого объявленного возвращаемого типа, никаких аргументов и должен (несмотря на то, что в данном случае нет никакого объявленного возвращаемого типа) возвращать преобразованное значение. Например, функция, преобразующая тип `Vector` в тип `double`, будет иметь такой вид:

```
Vector::operator double()
{
 ...
 return a_double_value;
}
```

Опыт показывает, что лучше не злоупотреблять подобными функциями неявного преобразования.

Как вы уже, должно быть, заметили, классы требуют проявления гораздо большей осторожности и внимания к деталям, чем простые C-структуры. Но за это вы будете вознаграждены.

## Вопросы для повторения

- Используйте функцию-элемент в целях выполнения перегрузки операции умножения для класса `Stonewt`; обеспечьте возможность умножения элементов данных этого класса на значение типа `double`. Обратите внимание на то обстоятельство, что для этого требуется перейти на единицы измерения, выраженные в стонах-фунтах. Другими словами, умножение 10 стонов 8 фунтов на 2 равно 21 стону 2 фунтам.

2. В чем заключается различие между дружественной функцией и функцией-элементом?
3. Может ли функция, не являющаяся членом класса, иметь доступ к элементам класса?
4. Используйте дружественную функцию для перегрузки операции умножения для класса **Stonewt**; обеспечьте возможность умножения значения **double** на значение **Stone**.
5. Какие операции не могут быть перегруженными?
6. Какие ограничения накладываются на перегрузку следующих операций: = () [] ->?
7. Дайте определение функции преобразования для класса **Vector**, которая преобразует тип **Vector** в значение типа **double**, представляющее соответствующий модуль вектора.

## Упражнения по программированию

1. Внести изменения в листинг 10.13, чтобы представленная в нем программа, вместо того чтобы сообщать результаты одной попытки конкретной комбинации цель-итерация, сообщала о максимальном, минимальном и среднем числе итераций для  $N$  попыток, где  $N$  — это целое число, задаваемое в программе пользователем.
2. Внесите такие изменения в класс **Stonewt**, чтобы в нем содержался элемент состояния, посредством которого можно было бы установить представление объекта в стоунах, в целых или дробных значениях фунтов. Перегрузите операцию << для замены методов **show\_stn()** и **show\_lbs()**. Перегрузите операции сложения, вычитания и умножения таким образом, чтобы можно было складывать, вычитать и умножать значения **Stonewt**. Выполните тестирование полученного класса с помощью короткой программы.
3. Перепишите класс **Stonewt** таким образом, чтобы он мог перегружать операции сравнения. Напишите программу, которая объявляет массив из шести объектов класса **Stonewt** и выполняет инициализацию трех первых объектов в объявлении массива. Воспользуйтесь циклом, чтобы осуществить считывание значений, используемых для присваивания остальным трем элементам массива. Далее необходимо отобразить максимальный и минимальный элементы, а также количество элементов, превышающих или равных 11 стоунам.
4. Комплексное число состоит из двух частей: вещественной и мнимой. Одним из способов записи мнимого числа является такой: (3.0, 4.0i). Здесь 3.0 — вещественная часть, а 4.0 — мнимая. Предположим,

что  $a = (A, Bi)$  и  $c = (C, Di)$ . Вот некоторые операции, выполняемые над комплексными числами:

- Сложение:  $a + c = (A + C, (B + D)i)$
- Вычитание:  $a - c = (A - C, (B - D)i)$
- Умножение:  $a * c = (A * C - B * D, (A * D + B * C)i)$
- Умножение: ( $x$  — вещественное число):  $x * c = (x * C, x * D)i$
- Поиск сопряженного числа:  $\bar{a} = (A, -Bi)$

Определите класс, оперирующий комплексными числами, чтобы приводимая ниже программа могла воспользоваться им и получать правильные результаты. Обратите внимание на то, что должна выполняться перегрузка операций << и >>. Многие системы уже включают поддержку комплексных чисел в заголовочном файле **complex.h**, поэтому воспользуйтесь файлом **complex0.h**, чтобы избежать проблем. Используйте ключевое слово **const** там, где это оправдано.

```
#include <iostream>
using namespace std;
#include "complex0.h" // чтобы избежать
 // путаницы с файлом complex.h
int main()
{
 // инициализация посредством (3,4i)
 complex a(3.0, 4.0);
 complex c;
 cout << "Enter a complex number (q to quit):\n";
 while (cin >> c)
 {
 cout << "c is " << c << '\n';
 cout << "complex conjugate is ";
 << ~c << '\n';
 cout << "a + c is " << a + c << '\n';
 cout << "a - c is " << a - c << '\n';
 cout << "a * c is " << a * c << '\n';
 cout << "2 * c is " << 2 * c << '\n';
 cout << "Enter a complex number (q to quit):\n";
 }
 cout << "Done!\n";
 return 0;
}
```

Результаты выполнения программы. (Обратите внимание на то, что оператор **cin >>** с благодаря перегрузке теперь требует ввода вещественной части и мнимой части.)

```
Enter a complex number (q to quit):
real: 10
imaginary: 12
c is (10,12i)
complex conjugate is (10,-12i)
a + c is (13,16i)
a - c is (13,16i)
a * c is (-18,76i)
2 * c is (20,24i)
Enter a complex number (q to quit):
real: q
Done!
```

# Классы и динамическое распределение памяти

**В этой главе рассматривается следующее:**

- Применение метода динамического распределения памяти для элементов класса
- Неявные и явные конструкторы копирования
- Неявные и явные перегруженные операторы присвоения
- Последовательность действий при использовании конструктора `new`
- Использование элементов класса `static`
- Использование указателей на объекты
- Реализация ADT типа "очередь"

**В** этой главе рассматривается порядок использования операторов `new` и `delete` при работе с классами. Анализируются также некоторые тонкие моменты, иногда возникающие при динамическом распределении памяти. Круг обсуждаемых вопросов включает проектирование конструкторов, деструкторов, а также перегрузку операторов.

Давайте рассмотрим специальный пример, который иллюстрирует, каким образом C++ может увеличивать степень загрузки памяти. Предположим, что требуется создать класс с элементом, представляющим фамилию какого-либо человека. Простейшим путем реализации этого замысла является применение элемента символьного массива, хранящего данное имя. Однако этот метод имеет некоторые недостатки. Предположим, что наш массив включает 14-символьные элементы, а человека зовут Бартоломью Смидсбери-Графтховингем (Bartholomew Smeadsbury-Crafthovingham). Понятно, что в этом случае для размещения имени выделенного места будет явно недостаточно. Для предотвращения возникновения подобных ситуаций можно использовать 40-символьные элементы массивов. Но если ваш массив будет включать 2 тыс. подобных объектов, возрастет объем памяти, занимаемой частично заполненными символьными массивами. (Нетрудно заметить, что в этом случае возрастает загрузка компьютерной памяти.) Однако в подобной ситуации имеется альтернативный вариант.

Многие вопросы, например, такие как вычисление объема используемой памяти, лучше решать при выполнении программы, чем во время ее компиляции. Обычный подход для хранения объекта (имени объекта), применяемый при работе с языком C++, состоит в использовании оператора `new` в конструкторе класса для выделения требуемого объема памяти. Но добавление оператора `new` в конструктор класса приведет к появлению некоторых новых проблем, если вы не примете дополнительные меры. В частности, речь идет о расширении деструктора класса, вызове всех конструкторов совместно с деструктором `new`, а также создании дополнительных методов класса для облегчения корректной инициализации и присвоения. (В этой главе рассматриваются все эти процессы.) Если вы только приступаете к изучению языка C++, то, возможно, предпочтете использовать символьные массивы. Затем при освоении проектирования классов можно вернуться к концепции ООП и расширить объявление класса путем применения оператора `new`. Одним словом, придерживайтесь методики постепенного прогресса при изучении C++.

## Динамическая память и классы

Что бы вы хотели получить на завтрак, обед и ужин в следующем месяце? Сколько унций молока вам перепадет на ужин в третий день? Сколько изюма будет положено в вашу кашу на пятнадцатый день? Если вы отно-

ситесь к подавляющему большинству людей, мыслящих стандартно, то решение некоторых из указанных вопросов будет отложено до момента фактического приема пищи. В языке C++ применяется аналогичная стратегия по отношению к распределению памяти. Память выделяется на этапе выполнения программы, а не на этапе компиляции. Вследствие этого применение памяти зависит от потребностей программы, а не определяется набором жестко установленных правил классов памяти. Для осуществления динамического управления памятью, как вы помните, C++ использует операторы `new` и `delete`. К сожалению, использование этих операторов при работе с классами может привести к возникновению новых проблем при программировании. Как будет показано дальше, применение деструкторов в этом случае может стать необходимым, а не формальным. При этом иногда будет перегружен оператор присвоения для того, чтобы программы вели себя должным образом. Теперь приступим к подробному изучению этих вопросов.

## Обзорный пример и элементы статических классов

На протяжении некоторого времени мы не использовали операторы `new` и `delete`. Сейчас рассмотрим их функционирование в составе небольшой программы. При этом используется новый класс памяти: элемент статического класса. В роли базового выступает класс `String`. (В комплект поставки C++ входит библиотека класса `string`, которая поддерживается заголовочным файлом `string.h`. Этот класс описывается в главе 15. Между прочим, скромный класс `String`, который рассматривается в этой главе, позволяет исследовать некоторые базовые принципы, лежащие в основе класса.) Объект класса `String` содержит указатель на строку и значение, представляющее длину строки. Класс `String` будет применяться прежде всего для контроля за тем, каким образом функционируют элементы классов `new`, `delete` и `static`. По этой причине конструкторы и деструкторы отображают сообщения при вызове, благодаря чему вы можете следовать действиям. При этом не используются некоторые полезные элементы и дружественные функции, такие как перегруженные операции `++` и `>>`, а также функции преобразования, применяемые в целях упрощения интерфейса класса. (Но не радуйтесь раньше времени! Вопросы для повторения, относящиеся к этой главе, включают полезные функции поддержки. Так что вам придется немного потрудиться.) Листинг 11.1 содержит объявление класса. В этом случае вызывается файл `strng1.h` вместо `string.h` для избежания конфликта с файлом стандартной библиотеки `string.h`. (Последние реализации C++ включают заголовочный файл `string.h`, поддерживающий функции С, подобные `strcpy()`, которые

работают со строками в стиле C, заголовочный файл `cstring`, основанный на файле `string.h`, и заголовочный файл `string`, который поддерживает класс C++ под названием `string`.)

### Листинг 11.1 Заголовочный файл `strng1.h`.

```
// strng1.h -- определение класса string
#include <iostream>
using namespace std;
#ifndef _STRNG1_H_
#define _STRNG1_H_
class String
{
private:
 char * str; // указатель на строку
 int len; // длина строки
 static int num_strings; // количество
 // объектов
public:
 String(const char * s); // конструктор
 String(); // конструктор, заданный по
 // умолчанию
 ~String(); // деструктор
// дружественная функция
 friend ostream & operator<<(ostream & os,
 const String & st);
};
#endif
```

Отметим две особенности, связанные с приведенным объявлением. Во-первых, для представления имени используется указатель-на-`char` вместо массива `char`. Это означает, что при объявлении класса не выделяется пространство для хранения самой строки. Вместо этого в конструкторах при выделении пространства для строки будет применяться оператор `new`. Такой подход позволяет избежать эффекта "смирительной рубашки" при объявлении класса с предопределенным предельным значением для размера строки.

Во-вторых, определение объявляет элемент `num_strings` в качестве элемента класса памяти `static`. Элемент статического класса имеет специальное свойство: программа создает только одну копию переменной статического класса независимо от количества созданных объектов. Это означает, что статический элемент распределяется среди всех объектов данного класса подобно тому, как телефонный номер может совместно использоваться всеми членами семьи. Если, например, создается 10 объектов типа `String`, в таком случае они могут включать 10 элементов `str` и 10 элементов `len`, но только один общий элемент `num_strings` (рис. 11.1). Это удобно при работе с данными, которые являются приватными по отношению к классу, но при этом должно поддерживаться одно и то же значение для всех объектов класса. Например, элемент `num_strings` предназначен для отслеживания количества созданных объектов.

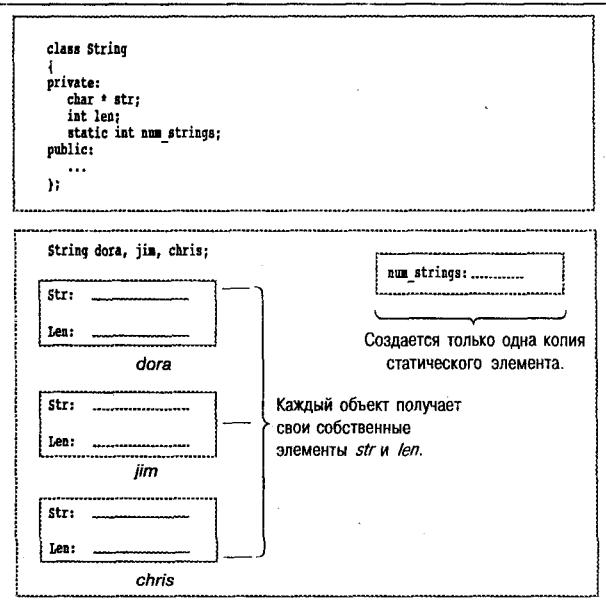


РИСУНОК 11.1 Элемент статических данных.

Между прочим, элемент `num_strings` применяется в качестве удобного средства для иллюстрирования элементов статических данных, а также как устройство для рассмотрения потенциальных проблем, связанных с программированием. Как правило, класс `String` не использует подобный элемент. Если вы нуждаетесь в подобном средстве, удобнее всего определить общий класс `String`,

### Листинг 11.2 Программа string1.cpp.

```

// string1.cpp -- методы класса String
#include <iostream>
#include <cstring> // string.h для некоторых случаев применения
#include "string1.h"
using namespace std;

// инициализация элемента статического класса
int String::num_strings = 0;

// методы класса
String::String(const char * s) // создание String на базе строки С
{
 len = strlen(s); // установка размера
 str = new char[len + 1]; // выделение памяти
 strcpy(str, s); // инициализация указателя
 num_strings++; // установка объекта cout
 cout << num_strings << ":" << str
 << "\n" object created\n"; // информация для вас
}
String::String() // конструктор, заданный по умолчанию
{
 len = 4;
 str = new char[4]; // строка, определенная по умолчанию
 strcpy(str, "C++");
 num_strings++;
 cout << num_strings << ":" << str
 << "\n" default object created\n"; // комментарий
}

```

а затем для добавления данного свойства в производный класс воспользоваться методом наследования класса (см. главу 12). (Наследование классов позволяет создавать новый класс, который расширяет существующий.)

Рассмотрим реализацию методов класса на примере листинга 11.2. Вы увидите, каким образом обрабатываются эти два объекта (иллюстрируется применение указателей и статических элементов).

Прежде всего обратите внимание на следующий оператор из листинга 11.2:

```
int String::num_strings = 0;
```

Этот оператор инициализирует статический элемент `num_strings`, присваивая ему значение нуль. Заметьте, что нельзя инициализировать переменную статического элемента в пределах объявления класса. Причина в том, что объявление представляет собой описание способа распределения памяти, но при этом не осуществляется само распределение памяти. Распределение и инициализация памяти осуществляются путем создания объекта с помощью данного формата. В случае со статическим элементом класса его инициализация осуществляется независимо, с помощью отдельного оператора, не входящего в объявление класса. Причина заключается в том, что элемент статического класса хранится отдельно, а не как часть объекта. Обратите внимание, что оператор инициализации предоставляет тип и использует операцию определения диапазона доступа:

```
int String::num_strings = 0;
```

```

String::~String()
{
 cout << "\"" << str << "\" object deleted, ";
 --num_strings;
 cout << num_strings << " left\n";
 delete [] str;
}

ostream & operator<<(ostream & os, const String & st)
{
 os << st.str;
 return os;
}

```

Рассматриваемая инициализация включается в файл методов, но не в файл объявления. Причина этого состоит в том, что объявление класса находится в заголовочном файле, а программа может включать заголовочный файл в несколько других файлов. Это приведет к появлению нескольких копий оператора инициализации, что является ошибкой.

Иключение (см. главу 9), состоящее в отсутствии инициализации элемента статических данных внутри объявления класса, проявляется в том случае, если элементом статических данных будет `const`, имеющий интегральный или перечислимый тип.

### ПОМНИТЕ

Элементы статических данных определяются в объявлении класса и инициализируются в файле, содержащем методы класса. Оператор определения диапазона доступа применяется при инициализации для индикации классов, которым принадлежат статические элементы. Однако, если статический элемент является `const`, имеющей интегральный или перечислимый тип, он может быть инициализирован в самом объявлении класса.

Заметим, что каждый конструктор содержит выражение `num_strings++`. Тогда каждый раз, когда программа создает новый объект, общая переменная `num_strings` увеличивается на единицу, отслеживая общее количество объектов `String`. Деструктор также включает выражение `--num_strings`. Таким образом, класс `String` также отслеживает удаленные объекты, поддерживая текущее значение элемента `num_strings`.

### ФУНКЦИИ СТАТИЧЕСКИХ КЛАССОВ ПАМЯТИ

Функцию-элемент также можно объявить статической. (Ключевое слово `static` должно содержаться в объявлении функции, но не в определении функции, если эти понятия существуют раздельно.) Существуют две важные последовательности. Во-первых, статическая функция-элемент не вызывается объектом; фактически с ней не может взаимодействовать даже указатель `this`. Если статическая функция-элемент объявляется в общедоступном разделе, к ней можно обратиться с помощью наименования класса и оператора определения диапазона доступа. Например, пред-

```

// необходимый деструктор
// комментарий
// обязательная часть
// комментарий
// обязательная часть

```

положим, что класс `String` включает статическую функцию-элемент под названием `HowMany()` со следующим прототипом/определением в объявлении класса:

```

static int HowMany() { return num_strings; }

```

К функции-элементу можно обратиться следующим образом:

```

int count = String::HowMany(); // вызывается
 // статическая функция-элемент

```

Во-вторых, поскольку статическая функция-элемент не ассоциируется с определенным объектом, из элементов данных она может применить только статические элементы данных. Например, статический метод `HowMany()` может получить доступ к статическому элементу `num_strings`, но не к `str` или `len`. Аналогично статическая функция-элемент может применяться для установки флага в масштабе класса. Этот флаг контролирует формат, который применяется для хранения данных в объектах отображаемого класса.

Рассмотрим первый конструктор, который инициализирует объект `String` вместе со стандартной строкой C:

```

String::String(const char * s)
 // сконструируйте String из строки C
{
 len = strlen(s); // установите размер
 str = new char[len + 1]; // распределение
 // памяти
 strcpy(str, s); // инициализация указателя
 num_strings++; // установка счетчика объекта
 cout << num_strings << ":" << " " << str
 << "\\" object created\n"; // комментарий
}

```

Вспомним, что элемент класса `str` является только указателем, поэтому конструктор должен выделить память, которая позволит хранить строку. При инициализации объекта можно передать строку указателя конструктору:

```

String boston("Boston");

```

Затем конструктор должен распределить достаточный объем памяти для хранения строки и скопировать строку в это место. Рассмотрим этот процесс более подробно.

Прежде всего функция инициализирует элемент `len`, применяя при вычислении длины строки функцию `strlen()`. Затем в целях распределения достаточного объема пространства для хранения строки применяется оператор `new`, после чего элементу `str` присваивается адрес, по которому находится новая память. (Как вы помните, функция `strlen()` отображает длину строки, не учитывая заключительный нулевой символ. Поэтому конструктор добавляет 1 к `len`, что позволит включать в пространство для строки нулевой символ.)

Затем для копирования передаваемой строки в новую область памяти конструктор использует функцию `strcpy()`. После этого обновляется объектный счетчик. Наконец, для мониторинга предстоящих изменений конструктор отображает текущее количество объектов и строку, которая хранится в объекте. Подобная возможность пригодится позднее, если при функционировании класса `String` случится сбой.

Для понимания этого подхода нужно добиться, чтобы строка не была включена в объект. Стока сохраняется отдельно, в области динамической памяти, а объект сохраняет информацию лишь о ее местонахождении.

Обратите внимание на тот факт, что нельзя использовать следующий оператор:

```
str = s; // не тот путь, которым
 // следует воспользоваться
```

Здесь просто хранится адрес, но не создается копия строки. Конструктор, заданный по умолчанию, ведет себя аналогичным образом. Различие лишь в том, что этот конструктор поддерживает заданную по умолчанию строку в стиле C++. Деструктор содержит самое важное дополнение к рассматриваемому примеру, которое поможет при обработке классов:

```
String::~String() //необходимый деструктор
{
```

### Листинг 11.3 Программа vegnews.cpp.

```
// vegnews.cpp -- применение операторов new и delete при работе с классами
// Компиляция с помощью strng1.cpp
#include <iostream>
using namespace std;
#include "strng1.h"

String sports("Spinach Leaves Bowl for Dollars");
void callme1();
String * callme2();

int main()
{
 cout << "Top of main()\n";
 String headlines[2] = // массив локальных объектов
 {
 String("Celery Stalks at Midnight"),
 String("Lettuce Prey")
 };
 // разновидность внешнего объекта
 // создает локальный объект
 // создает динамический объект
}
```

```
cout << "\" " << str
 << "\" object deleted, ";
--num_strings;
cout << num_strings << " left\n";
delete [] str;
}
```

Деструктор начинается с извещения о том, когда именно получен вызов. Эта часть носит информативный характер, но несущественна. Однако оператор `delete` играет важную роль. Обратите внимание, что элемент `str` указывает на память, распределенную с помощью `new`. Если объект `String` перестает существовать, указатель `str` также теряет силу. Но указатель `str` указывал на остаток памяти, который распределяется в том случае, если для очистки не применяется оператор `delete`. При удалении объекта происходит очистка памяти, которую он занимал ранее. Однако не происходит автоматической очистки памяти, которая определялась указателями-элементами объекта. С этой целью необходимо применить деструктор. Если разместить в деструкторе оператор `delete`, можно с уверенностью заключить, что при разрушении объекта произойдет очистка памяти, которая была распределена конструктором с помощью оператора `new`.

### ПОМНИТЕ

Всякий раз, когда в конструкторе для распределения памяти применяется оператор `new`, необходимо для очистки этой памяти использовать `delete` в соответствующем деструкторе. Если применяется функция `new[ ]` (со скобками), необходимо воспользоваться функцией `delete [ ]` (со скобками).

Листинг 11.3, который позаимствован из программы, разработанной при создании *The Daily Vegetable*, иллюстрирует методы работы с конструкторами и деструктором `String`. Не забудьте скомпилировать листинг 11.2 вместе с листингом 11.3.

```

cout << headlines[0] << "\n";
cout << headlines[1] << "\n";
callme1();
cout << "Middle of main()\n";
String *pr = callme2(); // устанавливается указатель для объекта
cout << sports << "\n";
cout << *pr << "\n"; // вызывается метод класса
delete pr; // удаляется объект
cout << "End of main()\n";
return 0;
}

void callme1()
{
 cout << "Top of callme1()\n";
 String grub; // локальный объект
 cout << grub << "\n";
 cout << "End of callme1()\n";
}

String * callme2()
{
 cout << "Top of callme2()\n";
 String *pveg = new String("Cabbage Heads Home");
 // динамический объект применяет конструктор
 cout << *pveg << "\n";
 cout << "End of callme2()\n";
 return pveg; // pveg перестает существовать, объект "живет"
}

```



### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Этот первый набросок проекта для **String** имеет несколько явных недочетов. Однако многие компиляторы не реагируют на них на данном этапе, хотя некоторые из них испытывают явные затруднения при попытках генерирования программного вывода. В следующем разделе, который посвящен рассмотрению класса **String**, обсуждаются проблемы, связанные с классами, и методы по их устранению. В некоторых реализациях языка завершающий деструктор также может не вызываться до тех пор, пока не прервется вывод. Поэтому сообщение от деструкторов может и не отображаться.

Результаты выполнения программы:

```

1: "Spinach Leaves Bowl for Dollars" object
 created
Top of main()
2: "Celery Stalks at Midnight" object
 created
3: "Lettuce Prey" object created
Celery Stalks at Midnight
Lettuce Prey
Top of callme1()
4: "C++" default object created
C++
End of callme1()
"C++" object deleted, 3 left
Middle of main()
Top of callme2()
4: "Cabbage Heads Home" object created
Cabbage Heads Home
End of callme2()
Spinach Leaves Bowl for Dollars
Cabbage Heads Home

```

```

"Cabbage Heads Home" object deleted, 3 left
End of main()
"Lettuce Prey" object deleted, 2 left
"Celery Stalks at Midnight" object deleted,
 1 left
"Spinach Leaves Bowl for Dollars" object
 deleted, 0 left

```

### Примечания к программе

Убедитесь, что вам понятна последовательность событий в этом образце программы; рассмотрите их снова. Объект **sports** представляет собой внешнюю переменную, поэтому ее создают еще до того, как начинает выполняться **main()**. Следующие созданные объекты представляют собой два элемента массива **headlines**. Программа вызывает конструктор дважды для инициализации каждого элемента массива. Каждый элемент является объектом класса, поэтому при вызове

```

cout << headlines[0] << "\n";
cout << headlines[1] << "\n";

```

для двух объектов, **headlines [0]** и **headlines [1]**, вызывается дружественный метод **operator <<()**. Затем программа вызывает функцию **callme1()**, которая использует заданный по умолчанию конструктор для создания локального объекта под названием **grub**. Этот конструктор инициализирует элемент **str** значением "**C++**". Локальный объект перестанет существовать, если функция **callme1()** завершит свое выполнение, как показано в следующих строках вывода:

```

Top of callme1()
4: "C++" default object created
C++
End of callme1()
"C++" object deleted, 3 left

```

Деструктор не только выводит на печать прощальное сообщение, но и очищает память, которая содержит строку "C++".

Рассмотрим некоторые особенности примера. Программа вызывает функцию `callme2()`, а для создания и инициализации объекта `String` применяется оператор `new`:

```

String *pveg = new String("Cabbage Heads
 ↪Home");

```

Функция присваивает указателю `pveg` адрес этого нового объекта. Поскольку функция с помощью аргумента строки поддерживает `new String`, программа в целях инициализации объекта вызывает соответствующий конструктор. На рис. 11.2 проиллюстрировано действие оператора.

Заметим, что, поскольку `pveg` является указателем на объект, `*pveg` представляет собой объект. Это значит, что `*pveg` можно использовать так же, как объявленный объект:

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

Это приводит к тому, что функция отображает строку "Cabbage Heads Home". Затем функция прерывает выполнение, автоматически освобождая память, которая используется ее переменными. Это значит, что память применяется для хранения освобожденного указателя `pveg`. Но поскольку `callme2()` не применяет `delete pveg`, остается распределенной память, содержащая объект, на который указывает `pveg`. Обратите внимание на отсутствие сообщения деструктора о выполнении прерывания

со стороны `callme2()`; это свидетельствует о том, что объект по-прежнему присутствует — строка "Cabbage Heads Home" продолжает существовать! Но в связи с тем что действие `pveg` завершается, программа больше не может осуществлять доступ к этому объекту с помощью `pveg`. Однако программа передает значение `pveg` назад, вызывающей программе, и присваивает его указателю `pr`. Одним словом, сначала `pveg` указывает на объект `String`, затем действие `pveg` завершается. Тем временем программа устанавливает `pr` как указатель на объект `String`. Таким образом, программа для доступа к динамическому объекту может использовать `pr`. И она выполняет это после первого отображения объекта `sports`:

```

Top of callme2()
4: "Cabbage Heads Home" object created
Cabbage Heads Home используется указатель
pveg в функции callme2()
End of callme2()
Spinach Leaves Bowl for Dollars
Cabbage Heads Home используется указатель pr
в функции main()

```

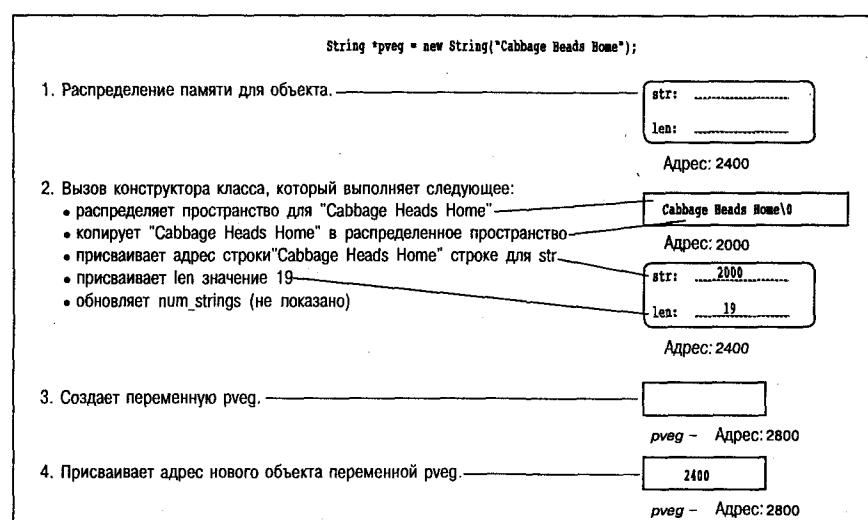
В настоящий момент программа приступает к выполнению довольно заурядных заданий по удалению оставшихся объектов. Поскольку программа создает объект `cabbage` с помощью оператора `new`, объект может быть удален с помощью оператора `delete`:

```
delete pr;
```

Напомним, что подобный подход позволяет освободить память, на которую указывает указатель `pr`, но не сам `pr`. Одним словом, этот оператор удаляет объект. При удалении объекта активизируется деструктор класса, который затем освобождает память, занимаемую строкой "Cabbage Heads Home":

```
"Cabbage Heads Home" object deleted, 3 left
```

**РИСУНОК 11.2**  
Создание объекта с помощью  
оператора `new`.



Правила распространения контролируют существование оставшихся объектов. Два массива элементов являются автоматическими переменными. Поэтому они прекращают свое существование, если процесс выполнения выходит за пределы блока, где они определены. В этом случае блок представляет собой тело функции `main()`, поэтому, если `main()` завершает свое выполнение, эти два объекта освобождаются. В конце концов, внешний объект прекращает свое существование, если программа завершает выполнение:

```
End of main()
"Celery Stalks at Midnight" object deleted,
 2 left
"Lettuice Frey" object deleted, 1 left
"Spinach Leaves Bowl for Dollars" object
 deleted, 0 left
```

(Как уже упоминалось, некоторые компиляторы не вызывают конструкторы для внешних объектов, пока не будет реализован вывод прерывания. Поэтому они не отображают заключительную строку.)

### **Новый подход к использованию операторов new и delete**

Обратите внимание, что программа использует операторы `new` и `delete` на двух уровнях. Во-первых, оператор `new` применяется в целях распределения памяти для строк наименований для каждого создаваемого объекта. Так происходит с функциями конструктора, поэтому функция деструктора использует оператор `delete` со скобками. Таким образом, память, используемая для хранения содержимого строк, освобождается автоматически при уничтожении объекта. Во-вторых, программа использует оператор `new` для распределения памяти для всего объекта при вызове функции `callme2()`. Это способствует выделению пространства не для строки наименований, а для объекта, т.е. в пространстве размещается указатель `str`, который содержит адрес строки и адрес элемента `len`. (Пространство для элемента `num_strings` не выделяется, поскольку этот статистический элемент хранится отдельно от объектов.) При создании объекта, в свою очередь, вызывается конструктор, который выделяет место для хранения строки и присваивает указателю `str` адрес строки. Затем программа применяет оператор `delete` для удаления объекта, когда работа с ним завершается. Этот объект является отдельным объектом, поэтому в программе применяется оператор `delete` без скобок. Таким образом, опять освобождается только пространство, которое используется указателем `str` и элементом `len`. Память, которая используется для хранения строки, на которую указывает `str`, не освобождается, но деструктор принимает на себя выполнение этой завершающей задачи.

Следует снова подчеркнуть, когда именно вызываются деструкторы (рис. 11.3).

- Если объект является автоматической переменной, вызов деструктора объекта реализуется в том случае, когда выполнение программы передается за пределы блока, в котором определяется объект. Таким образом, деструктор вызывается для `headlines[0]` и `headlines[1]`, если выполнение программы передается за пределы функции `main()`, а деструктор для `grub` вызывается, если выполнение программы передается за пределы функции `callme1()`.
- Если объект представляет собой статическую переменную (внешнюю, статическую, статическую внешнюю или переменную из пространства имен), при прекращении выполнения программы вызывается ее деструктор. Это справедливо для объекта `sports`.
- Если объект создается с помощью оператора `new`, соответствующий деструктор вызывается только в том случае, если происходит удаление объекта, как и в случае, когда объект создается в блоке `callme2()` и удаляется в блоке `main()`.

Метод, с помощью которого для создания объекта в одной функции и удаления объекта в другой функции используется оператор `new`, может стать причиной потенциальных затруднений. Именно программист должен помнить о необходимости удалить объект. Например, рассмотрите следующее скрытое изменение:

```
String * ps;
for (int i = 0; i < 100; i++)
{
 ps = callme2();
 cout << *ps << "\n";
}
delete ps;
```

```
class Act { ... };
...
Act nice; // external object
...
int main()
{
 Act *pt = new Act; // dynamic object
 {
 Act up; // automatic object
 ...
 }
 delete pt;
 ...
}
```

деструктор для автоматического объекта вызывается в том случае, если выполнение достигает конца данного блока

деструктор для динамического объекта \*`pt` вызывается в том случае, если оператор `delete` применяется к указателю `pt`

деструктор для статического объекта `nice` вызывается в том случае, если выполнение достигает конца программы

поскольку этот статистический элемент хранится отдельно от объектов.

РИСУНОК 11.3 Вызов деструкторов.

С помощью этого кода создается 100 определенных объектов. Каждая итерация цикла устанавливает `ps` для указания самого последнего объекта, причем не указывается размещение предшествующего объекта. Наконец, программный код удаляет только тот объект, который создавался в последнюю очередь. Остальные 99 объектов занимают память, к которой программа не имеет доступа. Подобный тип программирования приводит к так называемой *утечке памяти*. Если деструкторы сконструированы должным образом, то утечка внутренней памяти происходит только при разрушении объекта. Но предположим, что требуется явным образом удалять объекты, которые создавались с помощью оператора `new`. В этом случае оператор `delete` должен размещаться в пределах цикла `for`.

## Устранение проблем, связанных с классом String

Описанное выше определение класса `String` является неполным. Конечно, ради краткости изложения пришлось пожертвовать реализацией многих полезных методов типа перегруженных операций `<`, `==` и `>`. Эти операции позволяют выполнять сравнения строк. Однако имеются еще более существенные недостатки, обусловленные природой классов. Для проверки этого утверждения обратите внимание на следующую простую программу (листинг 11.4), где применяется текущая реализация `String`:

### Листинг 11.4 Программа problem1.cpp.

```
// problem1.cpp -- применяет функцию с
// аргументом типа String
// Компиляция с помощью strng1.cpp
#include <iostream>
using namespace std;
#include "strng1.h"

void showit(String s, int n);
int main()
{
 String motto("Home Sweet Home");
 showit(motto, 3);
 return 0;
}

void showit(String s, int n) // отображение
 // объекта s типа String n раз
{
 for (int i = 0; i < n; i++)
 cout << s << "\n";
}
```



### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Поскольку данная и следующие две программы демонстрируют недостатки проектирования, сам вывод будет ме-

няться от компилятора к компилятору. Приведенные здесь примеры получены с помощью компилятора Borland C++ 3.1.

В этом примере объект `String` передается функции `showit()`, которая затем отображает строку заданное количество раз. Ниже приводится пример вывода для одной системы:

```
1: "Home Sweet Home" object created
Home Sweet Home
Home Sweet Home
Home Sweet Home
"Home Sweet Home" object deleted, 0 left
"Home Sweet Home" object deleted, -1 left
Null pointer assignment
```

Обратите внимание на следующие две особенности. В программе создается только один объект, а разрушаются два объекта. При этом в памяти остается общее значение для объектов, равное `-1`. (В конкретном случае реализации вместо `"Home Sweet Home"` в качестве заключительного выводимого результата может отображаться какая-то несуразица.) Затем отображается сообщение о присваивании нулевого указателя. (Пользователь может получить подобное сообщение. Это связано со свойствами конкретного компилятора. Или же в данном случае будет выводиться какое-либо иное сообщение. Возможно, вы и вовсе не получите никакого сообщения. Однако эту проблему не следует упускать из внимания.)

Ниже рассматривается другая простая программа (листинг 11.5), которая вызывает сомнения:

### Листинг 11.5 Программа problem2.cpp.

```
// problem2.cpp -- инициализирует одну строку
// присваиванием другой строки
// Компиляция с помощью strng1.cpp
#include <iostream>
using namespace std;
#include "strng1.h"

int main()
{
 String motto("Home Sweet Home");
 String ditto(motto); // инициализирует
 // ditto значениями motto

 cout << motto << "\n";
 cout << ditto << "\n";
 return 0;
}
```

Предпринимается попытка добиться того, чтобы объявление класса не обрабатывалось в явном виде: один объект `String` инициализирует другой. Однако, как показывает следующий вывод, эта попытка оказалась успешной. Программа также представляет несколько нео-

бычный подход, который иллюстрируется следующим примером:

```
1: "Home Sweet Home" object created
Home Sweet Home
Home Sweet Home
"Home Sweet Home" object deleted, 0 left
"Home Sweet Home" object deleted, -1 left
Null pointer assignment
```

## Неявные функции-элементы

Общим для этих двух примеров является тот факт, что в обоих случаях вызываются неявные функции-элементы. Эти функции определяются автоматически, и их поведение не соответствует данному особому классу разработки. В частности, C++ автоматически поддерживает следующие функции-элементы:

- Заданный по умолчанию конструктор, если конструкторы не определены
- Конструктор копирования, если он не определен
- Оператор присваивания, если он не определен
- Заданный по умолчанию деструктор, если он не определен
- Оператор адресации, если он не определен

Неявный оператор адресации возвращает адрес вызываемого объекта (т.е. величину указателя `this`). Это удобно для наших целей, в дальнейшем эта функция-элемент не рассматривается. Деструктор, заданный по умолчанию, не выполняет никаких действий. Поэтому этот деструктор не рассматривается далее. К этой теме следует обращаться в том случае, если речь идет о классе, заменяющем этот деструктор. Все эти вопросы могут стать предметом обширных дискуссий.

### Конструктор, заданный по умолчанию

Если при разработке каких-либо конструкторов у вас возникают трудности, C++ предлагает воспользоваться конструктором, заданным по умолчанию. Например, предположим, что определен класс `Klunk` и пропущены все конструкторы. Затем компилятор будет поддерживать следующий заданный по умолчанию конструктор:

```
Klunk::Klunk() { } // неявный конструктор,
// заданный по умолчанию
```

Таким образом поддерживается конструктор, который не имеет аргументов и не выполняет никаких действий. Он необходим, поскольку при создании объекта всегда вызывается конструктор:

```
Klunk lunk; // вызывает конструктор,
// заданный по умолчанию
```

Конструктор, заданный по умолчанию, превращает `Klunk` в подобие обычной автоматической переменной;

таким образом, в период инициализации значение этой переменной неизвестно.

После определения любого конструктора C++ не заботится об определении конструктора, заданного по умолчанию. Если необходимо создать объекты, которые не инициализируются в явном виде, или же массив объектов, требуется в явном виде определить конструктор, заданный по умолчанию. Этот конструктор не имеет аргументов, но можно воспользоваться им для установки определенных значений:

```
Klunk::Klunk() // явный вид конструктора,
// заданного по умолчанию
{
 klunk_ct = 0;
```

Конструктор с аргументами по-прежнему может быть конструктором, заданным по умолчанию, если все его аргументы располагают заданными по умолчанию значениями. Например, класс `Klunk` может иметь следующий встроенный конструктор:

```
Klunk(int n = 0) { klunk_ct = n; }
```

Однако можно иметь только один конструктор, заданный по умолчанию. Поэтому вы не сможете выполнить следующее действие:

```
Klunk() { klunk_ct = 0 }
Klunk(int n = 0) { klunk_ct = n; }
// двусмысленный
```

### Конструктор копирования

Конструктор копирования применяется для копирования объекта в заново создаваемый объект. Поэтому он используется в процессе инициализации, а не во время обычного присваивания. Конструктор копирования для класса имеет следующий прототип:

```
Class_name(const Class_name &);
```

Заметим, что здесь содержится постоянная ссылка на класс объекта в качестве соответствующего аргумента. Например, конструктор копирования для класса `String` будет иметь следующий прототип:

```
String(const String &);
```

По поводу конструктора копирования следует уточнить два вопроса: когда его использовать и каковы его функции.

### Условия применения конструктора копирования

Конструктор копирования вызывается всякий раз, когда создается новый объект. Этот конструктор аналогичным образом инициализируется для существующего объекта. Такая ситуация может быть реализована в следующих случаях. Чаще всего приходится явным образом инициализировать новый объект для существующего

объекта. Например, предположим, что `motto` является объектом `String` и следующие четыре определяющих объявления вызывают конструктор копирования:

```
String ditto(motto); // вызывает
 // String(const String &)
String metoo = motto; // вызывает
 // String(const String &)
String also = String(motto); // вызывает
 // String(const String &)
String * pstring = new String(motto);
 // вызывает String(const String &)
```

В зависимости от реализации два средних объявления могут применять конструктор копирования непосредственно для создания `metoo` и `also`. Кроме того, эти объявления могут использовать конструктор копирования для образования временных объектов, содержимое которых затем присваивается `metoo` и `also`. В последнем примере анонимный объект инициализируется для `motto` и указателю `pstring` присваивается адрес нового объекта.

Менее очевиден тот факт, что компилятор применяет конструктор копирования, когда программа генерирует копии объекта. В частности, он применяется, если функция передает объект по значению или возвращает объект. Помните, передача по значению означает создание копии исходной переменной. Компилятор также применяет конструктор копирования, когда он генерирует временные объекты. Например, компилятор может генерировать временный объект `Vector`, в котором при добавлении трех объектов `Vector` будет содержаться промежуточный результат. Компиляторы будут варыроваться всякий раз, когда генерируются временные объекты, но все они вызывают конструктор копирования при передаче объектов по значению или при возвращении объектов. В частности, в листинге 11.4 функция вызывает конструктор копирования:

```
showit(motto, 3); // создает и передает
 // копию объекта motto
```

Программа использует конструктор копирования для инициализации `st`, формального параметра типа `String` для функции `showit()`.

Кстати, тот факт, что передача объекта по значению приводит к вызову конструктора копирования, является удачным поводом для реализации передачи по ссылке. Такой подход позволяет сберечь время, затрачиваемое на вызов конструктора, а также способствует экономии пространства для хранения нового объекта.

### Функции конструктора копирования

Конструктор копирования, заданный по умолчанию, выполняет поэлементное копирование нестатических

элементов. Каждый элемент копируется по значению. В листинге 11.5 подобный подход иллюстрируется следующим образом:

```
ditto.str = motto.str;
ditto.len = motto.len;
```

Если элемент является объектом класса, конструктор копирования для этого класса применяется для копирования одного элемента объекта в другой. Статические элементы типа `num_strings` не изменяются, они принадлежат классу как целому, а не отдельным объектам. На рис. 11.4 проиллюстрировано действие неявного конструктора копирования.

### К чему могут привести возможные ошибки

Сейчас пришло время рассмотреть три важных новшества, которые содержатся в листингах 11.4 и 11.5. Первое новшество состоит в том, что в выводе программы показан один сконструированный объект и два разрушенных. Дело в том, что каждая программа создает два объекта, причем второй объект создается с помощью конструктора копирования. Конструктор копирования, заданный по умолчанию, не сообщает никакой информации о своей деятельности. Поэтому создание объектов не анонсируется. Это новшество при коммуникации имеет косметический характер и не оказывает влияния на надежность программы.

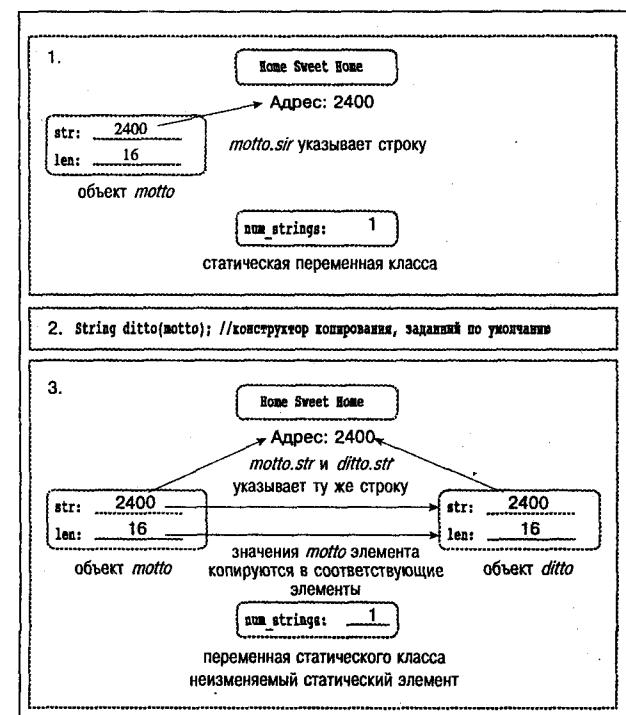


РИСУНОК 11.4 Поэлементное копирование.

Второе новшество заключается в том, что каждая программа сообщала, что осталось -1 объектов. Дело в том, что конструктор, заданный по умолчанию, не оказывает влияния на статические элементы. Поэтому конструктор копирования не обновляет счетчик `num_strings`. Однако деструктор обновляет счетчик и вызывается для отказа от всех объектов, независимо от способа их образования. Это новшество представляет некоторые трудности, поскольку программа не сохраняет счетчик объекта в точности. Для разрешения этого затруднения следует поддерживать явную копию конструктора, который выполняет обновление счетчика:

```
String::String(const String & s)
{
 num_strings++;
 ... // важный материал для перехода в это
 // место
}
```



### СОВЕТ

Если ваш класс содержит статический элемент данных, значение которого изменяется при создании новых объектов, поддерживайте явную копию конструктора, занятого выполнением учета.

Третье новшество не столь заметно, но представляется серьезную опасность. Симптомом этого явления при работе Borland 3.1 является следующее сообщение:

### Null pointer assignment

Подобное сообщение отображается после того, как прерывается программа.

Microsoft Visual C++ 5.0 (режим отладки) отображает окно с сообщением об ошибке. В этом окне утверждается, что допущена ошибка Debug Assertion `_CtrlValidHeapPointer(pUserData)`. Другие системы могут выводить другие сообщения или даже вовсе не выводить никаких сообщений. Нужно учитывать, что в программах могут быть скрыты особенности, которые описаны выше.

Причина этого явления заключается в том, что неявный конструктор копирования выполняет копирование по значению. Обратите внимание, например, на листинг 11.5. Здесь эффект проявляется в следующем:

```
ditto.str = motto.str;
```

Это не приводит к копированию строки; копируется указатель на строку. Поэтому после инициализации `ditto` значением `motto` вы располагаете двумя указателями на одну строку. Не вызывает особых затруднений тот случай, когда функция `operator<<()` применяет указатель для отображения строки. Трудности возникают, если вызывается деструктор. Деструктор `String` освобождает

память, которая отмечена указателем `str`. Ниже показан эффект от разрушения `ditto`:

```
delete [] ditto.str; // удаление строки, на
// которую указывает ditto.str
```

Это приводит к высвобождению памяти, занятой строкой "Home Sweet Home". Затем получаем следующий эффект от разрушения `motto`:

```
delete [] motto.str; // эффект не
// определен
```

Здесь `motto.str` указывает на местоположение памяти, которая только что была высвобождена. И это приводит к неопределенности, а возможно, и к сбоям в работе. В данном случае программа выдает предупреждение в виде нулевого указателя. Это предостережение обычно представляет собой указание на ошибку в распределении памяти.

Для устранения этой особенности применяют глубокую копию. Вместо того чтобы просто копировать адрес строки, конструктор копирования должен дублировать строку и присваивать адрес дубликата элементу `str`. Таким образом, каждый объект получает свою собственную строку, а не ссылку на строку другого объекта. В результате при каждом вызове деструктора освобождается другая строка, вместо того чтобы имеющийся дубликат предпринимал попытки освобождения той же самой строки. Ниже рассматривается способ кодирования конструктора копирования `String`:

```
String::String(const String & st)
{
 num_strings++; // выполнение обновления
 // статического элемента
 len = st.len; // та же длина
 str = new char [len +1]; //распределение
 //пространства
 strcpy(str, st.str); //копирование строки в
 //новое место
 cout << num_strings << ":" << str
 << "\\" object created\n"; // для
 // информации
}
```

Определение конструктора копирования обязательно, поскольку некоторые элементы класса были инициализированы оператором `new` с помощью указателей на данные, что предпочтительнее, чем в случае самостоятельной инициализации. На рис. 11.5 иллюстрируется глубокое копирование.



### ПРЕДОСТЕРЕЖЕНИЕ

Если класс содержит элементы, которые являются указателями, инициализированными оператором `new`, следует определить конструктор копирования. Этот конструктор копирует сведения, отмеченные указателями, в данные, а не сами указатели. Такой подход ограничивает глубокое копирование.

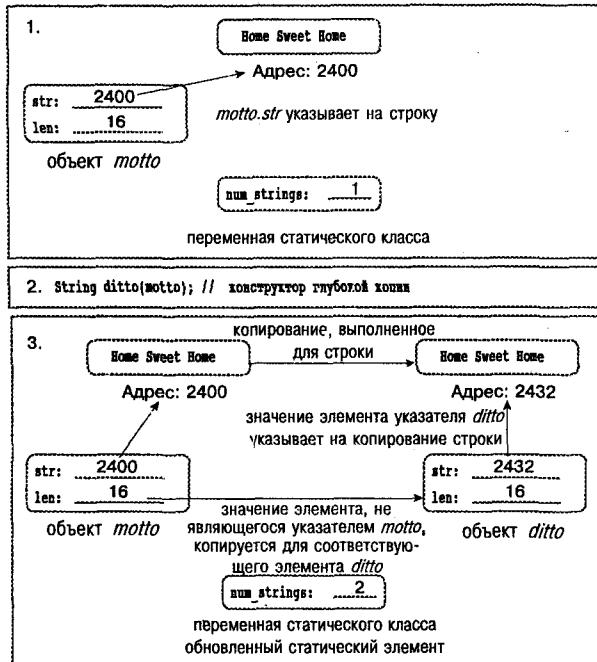


РИСУНОК 11.5 Глубокое копирование.

Вскоре будет проведено тестирование нового конструктора копирования. Но сначала рассмотрим другую программу, которая демонстрирует определенные недостатки (листинг 11.6). В программе реализуется присваивание одного объекта другому.

#### Листинг 11.6 Программа problem3. cpp.

```
// problem3.cpp -- присваивает один объект
// другому
// Компиляция с помощью strng1.cpp
#include <iostream>
using namespace std;
#include "strng1.h"

int main()
{
 String motto("Home Sweet Home");
 String ditto; // конструктор,
 // заданный по умолчанию
 ditto = motto; // присваивание объекта
 cout << motto << "\n";
 cout << ditto << "\n";
 return 0;
}
```

Ниже показано, что же получится, если выполнить эту программу (результаты могут быть разными в зависимости от компилятора):

```
1: "Home Sweet Home" object created
2: "C++" object created
Home Sweet Home
Home Sweet Home
```

```
"Home Sweet Home" object deleted, 1 left
Sweet Home" object deleted, 0 left
Null pointer assignment
```

Здесь, по крайней мере, процесс разрушения уравновешивает процесс созидания. Однако складывается впечатление, что один процесс разрушения оказывает влияние на строку другого объекта. Поэтому при разрушении поступает сообщение о проблемах с другой строкой. И снова большее количество сообщений о нулевом указателе поразит наше воображение. Результат выполнения программы опять-таки зависит от компилятора, но, даже если полученный вывод будет отличаться корректным видом, могут остаться скрытые проблемы, которые таят в себе потенциальную опасность.

#### Оператор присваивания

Подобно тому как ANSI C позволяет выполнять присваивание структуры, C++ дает возможность осуществлять присваивание объекта класса. Этот процесс выполняется в автоматическом режиме путем перегрузки оператора присваивания для класса. Он имеет следующий прототип:

```
Class_name & Class_name::operator=(const
Class_name &);
```

Таким образом реализуется получение и возвращение ссылки на объект класса. Например, ниже указан прототип для класса String:

```
String & String::operator=(const String &);
```

#### Применение оператора присваивания

Оператор присваивания, для которого выполнена перегрузка, применяется в том случае, если один объект присваивается другому:

```
string motto("Home Sweet Home");
string ditto;
ditto = motto; // применение оператора
 // присваивания, для которого
 // выполнена перегрузка
```

Этот оператор не обязательно использовать при инициализации объекта:

```
string metoo = ditto; // применение
 // конструктора копирования
```

Здесь **metoo** представляет заново созданный объект, который инициализирован значениями **ditto**; следовательно, применяется конструктор копирования. Однако, как упоминалось ранее, реализации характеризуются возможностью двухэтапной обработки этого оператора. Это значит, что для создания временного объекта применяется конструктор копирования, а затем при копировании значений для нового объекта выполняется присваивание. Другими словами, при инициализации всегда

вызывается конструктор копирования, кроме того, формы с помощью оператора = тоже могут вызывать оператор присваивания.

### Функции оператора присваивания

Подобно оператору копирования, при неявной реализации оператора присваивания выполняется поэлементное копирование. Если сам элемент является объектом некоторого класса, программа применяет оператор присваивания, который определен для этого класса. Таким образом, выполняется копирование для этого определенного элемента. Статические элементы данных остаются без изменений.

### К чему могут привести возможные ошибки

При рассмотрении листинга 11.6 можно заметить две особенности. Во-первых, при вторичном вызове деструктора отображается строка, которая изменена до неузнаваемости. Во-вторых, при прерывании программы генерирует сообщение о присваивании нулевого указателя. Обе эти особенности указывают на ошибки в процессе распределения памяти. Здесь проявляется проблема, аналогичная той, которая имела место при рассмотрении конструктора копирования: при поэлементном копировании происходит копирование значений указателей вместо ссылок на данные. Таким образом, если деструктор вызывается для `ditto`, происходит удаление строки "Home Sweet Home". Если же деструктор вызывается для `motto`, предпринимается попытка удаления той строки, которая уже была удалена на предварительном этапе. Как упоминалось ранее, при попытке удалить данные, которые были уже удалены на предварительном этапе, возможны непредсказуемые последствия. Кроме того, при очистке памяти может измениться ее содержимое. В листингах 11.4 и 11.5 при первоначальном применении оператора `delete` строка на самом деле остается неизменной, при повторном обращении к деструктору строка отображается корректно. Однако в листинге 11.6 при первоначальном удалении строка изменяется. Как можно отметить, если в результате выполнения некоторой операции проявляется неопределенный эффект, ваш компилятор может выполнять любые предложенные ему действия. Таким образом можно отображать текст Декларации о независимости (Declaration of Independence) или удалять с жесткого диска нежелательные файлы.

### Фиксированное присваивание

Для разрешения тех затруднений, которые возникают при использовании неподходящего оператора присваивания, заданного по умолчанию, следует обратиться к определению собственного оператора присваивания. Этот оператор создает глубокую копию. Реализация ана-

логична конструктору копирования, однако следует обратить внимание на некоторые отличия.

1. Поскольку целевой объект может иметь ссылку на данные, которые распределены на предварительном этапе, для "освобождения от ранее взятых обязательств" функция должна применять оператор `delete []`.
2. Функция должна реализовать защиту от присваивания объекта самого себе. В противном случае в процессе освобождения памяти, который описан выше, может произойти удаление содержимого объекта еще до выполнения повторного присваивания.
3. Функция возвращает ссылку на вызываемый объект.

При возвращении объекта функция может эмулировать способ обычного присваивания для встроенных типов данных, которые могут соединяться в цепь. Иначе говоря, если A, B и C являются объектами `String`, можно записать следующее:

```
A = B = C;
```

В обозначении для функции это выражение принимает следующий вид:

```
A.operator=(B.operator=(C));
```

Таким образом, возвращаемое значение `B.operator=(C)` становится аргументом функции `A.operator=()`. Поскольку возвращаемое значение является ссылкой на объект `String`, она принадлежит корректному типу аргумента.

Ниже показано, каким образом можно записать оператор присваивания для класса `String`:

```
String & String::operator=(const String & st)
{
 if (this == &st) // объект, присвоенный
 // самому себе
 return *this; // все сделано
 delete [] str; // освобождение старой
 // строки
 len = st.len;
 str = new char [len + 1]; // получение
 // пространства для новой строки
 strcpy(str, st.str); // копирование строки
 return *this; // возвращение ссылки
 // вызываемому объекту
}
```

Прежде всего код проверяется на самоприсваивание. Этот процесс протекает следующим образом. Если адрес, который находится в правой части оператора присваивания (&s), аналогичен адресу объекта-получателя (`this`), значит, имеет место самоприсваивание. Если это так, программа возвращает `*this` и прерывает свое выполнение. Обратившись к главе 10, вы увидите, что оператор присваивания является одним из операторов, которые

могут перегружаться только с помощью функции-элемента класса.

С другой стороны, функция приступает к освобождению памяти, на которую указывает `str`. Дело в том, что довольно скоро `str` будет присвоен адрес новой строки. Если оператор `delete` не применяется впервые, предыдущая строка останется в памяти. Поскольку программа больше не располагает указателем для старой строки, память будет не нужна.

После этого программа функционирует как конструктор копирования. Происходит распределение пространства, которое необходимо для новой строки, затем в новое место копируется строка из правостороннего объекта.

После завершения этой процедуры программа возвращает указатель `*this` и прерывает свое выполнение.

В процессе присваивания не создается новый объект, поэтому нет необходимости в настройке значения элемента статических данных `num_strings`.

## Новый, усовершенствованный класс `String`

Теперь, когда у читателя уже сложилось определенное представление о классах, внимательно рассмотрим класс `String`. Обратите внимание, что добавлены конструктор копирования и оператор присваивания, о которых речь шла ранее. Поэтому класс корректно управляет памятью, применяемой объектами класса. После рассмотрения способов конструирования и разрушения объектов можно обратиться к конструкторам и деструкторам классов. Заметьте, их не следует анонсировать всякий раз, когда они применяются. Поскольку до сих пор не приходилось наблюдать конструкторы в работе, нужно упростить заданный по умолчанию конструктор. Тогда вместо "C++" конструируется пустая строка. Теперь после рассмотрения принципа функционирования элементов статических данных можно устраниТЬ возможность выполнения пересчета параметров объекта.

Затем добавим к классу несколько возможностей. Чрезвычайно полезный класс `String` должен включать все функциональные возможности стандартной библиотеки `cstring`, куда входят строковые функции. Мы рассмотрим только основные функции, которые необходимы для иллюстрации метода. (Не забывайте, что класс `String` является лишь примером для иллюстрации, и стандартный строковый класс C++ значительно более обширен.) В частности, будут добавлены следующие методы:

```
int length () const { return len; }
friend bool operator>(const String &st1,
 const String &st2);
friend bool operator<(const String &st,
 const String &st2);
```

```
friend bool operator==(const String &st,
 const String &st2);
friend operator>>(istream & is,
 String & st);
```

Первый из новых методов возвращает длину сохраняемой строки, а три последующие позволяют сравнивать строки. Функция `operator>()`, например, возвращает значение `true` в том случае, если первая строка поступает после второй строки в алфавитном порядке (или, более точно, в последовательности, которая упорядочивается компьютером). Самым простым способом реализации функций сравнения строк является применение стандартной функции `strcmp()`. Эта функция возвращает отрицательное значение в том случае, если первый аргумент предшествует второму в алфавитном порядке, значение нуль, если возвращаются те же самые строки, и положительное значение, если первый следует за вторым в алфавитном порядке. Поэтому можно применить `strcmp()` следующим образом:

```
bool operator>(const String &st1,
 const String &st2)
{
 if (strcmp(st1.str, st2.str) > 0)
 return true;
 else
 return false;
}
```

При реализации функций сравнения значительно упрощаются операции сравнения между объектами `String` и обычными строками С. Например, предположим, что `answer` является объектом `String` и вы располагаете следующим программным кодом:

```
if ("love" == answer)
```

Далее выполняется преобразование следующего вида:

```
if (operator==("love", answer))
```

Затем компилятор применяет один из конструкторов для преобразования кода:

```
if (operator==(String("love"), answer))
```

И это соответствует прототипу. Следует сказать и о новом конструкторе, заданном по умолчанию. Этот конструктор имеет следующий вид:

```
String::String()
{
 len = 0;
 str = new char[1];
 str[0] = '\0'; // строка, заданная по
 // умолчанию
}
```

Вас может заинтересовать, почему код выполняет следующее действие:

```
str = new char[1];
```

но не такое:

```
str = new char;
```

Обе формы распределяют один и тот же объем памяти. Отличие состоит в том, что первая форма совместима с деструктором класса, а вторая — нет. Деструктор, напомним, содержит следующий код:

```
delete [] str;
```

Применение оператора **delete** со скобками совместимо с применением указателей, которые инициализируются с помощью оператора **new** со скобками и с помощью нулевого указателя. Подобный эффект не определен для указателей, которые инициализируются каким-либо иным способом.

Перед тем как приступить к рассмотрению новых листингов, обратим внимание на следующий момент. Предположим, что в объект **String** нужно копировать обычную строку и что для чтения строки применяется функция **getline()**, а строка размещается в объекте **String**. Методы класса позволяют выполнить следующее:

```
String name;
char temp[40];
cin.getline(temp, 40);
name = temp; // примените конструктор для
 // преобразования типа
```

Однако это решение вряд ли можно признать удовлетворительным, если часто к нему обращаться. Для того чтобы понять, почему так происходит, рассмотрим, каким образом функционирует заключительный оператор из приведенного выше листинга:

#### Листинг 11.7 Класс string2.h.

```
// string2.h -- определение класса String
#ifndef _STRNG2_H_
#define _STRNG2_H_
#include <iostream>
using namespace std;
class String
{
private:
 char * str; // указатель для строки
 int len; // длина строки
public:
 String(const char * s); // конструктор
 String(); // конструктор, заданный по умолчанию
 String(const String & st); // деструктор
 ~String(); // деструктор
 int length() const { return len; }
// перегруженные операторы
 String & operator=(const String & st); // оператор присваивания
 String & operator=(const char * s); // оператор присваивания #2
// дружественные функции
 friend bool operator>(const String &st1, const String &st2);
 friend bool operator<(const String &st, const String &st2);
```

1. Программа применяет конструктор **String (const char \*)** для создания временного объекта **String**. Этот объект содержит копию строки, которая хранится в **temp**.

Не забывайте, что конструктор с единственным аргументом является функцией преобразования (см. главу 10).

- Программа применяет оператор **String & String ::operator=(const String &)** для копирования информации из временного объекта в именованный объект.
- Программа вызывает деструктор **~String()** для удаления временного объекта.

Самым простым способом повышения эффективности процесса является такая перегрузка оператора присваивания, которая позволит ему работать непосредственно с обычной строкой. Это приводит к устранению дополнительных этапов и разрушению временного объекта. Ниже показана одна из возможных реализаций:

```
String & String::operator=(const char * s)
{
 delete [] str;
 len = strlen(s);
 str = new char[len + 1];
 strcpy(str, s);
 return *this;
}
```

Как обычно, следует перераспределить память, управление которой ранее осуществлялось с помощью **str**, и распределить объем памяти, достаточный для новой строки.

Листинг 11.7 показывает измененное объявление класса.

```

friend bool operator==(const String &st, const String &st2);
friend ostream & operator<<(ostream & os, const String & st);
friend istream & operator>>(istream & is, String & st);
};

#endif

```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Может быть, ваш компилятор не имеет реализованного типа данных **bool**. В этом случае вместо **bool** можно применять **int**, вместо **false** – **0**, а вместо **true** – **1**.

В листинге 11.8 представлены пересмотренные определения метода.

Перегруженный оператор **>>** поддерживает простой способ чтения строки, которая введена в объект **String** с клавиатуры. Однако этот путь не является ошибочным, при его реализации предусматривается, чтобы длина строки ввода не превышала 79 символов. Не забывайте, что значение объекта **istream** в условии **if** оценивается

как **false**, если при определенных обстоятельствах на этапе ввода происходит сбой. Например, это может произойти при завершении файла или же, как в случае с **get(char \*, int)**, при прочтении пустой строки.

Приступим к изучению класса. Для этого воспользуемся короткой программой, которая позволяет вводить несколько строк. Программа включает вводные замечания пользователя, размещает строки в объекты **String**, отображает их и сообщает, какая из строк является самой короткой, а какая поступит первой в соответствии с порядком по алфавиту. Эта программа представлена в листинге 11.9.

### Листинг 11.8 Программа strng2.cpp.

```

// strng2.cpp -- методы класса String
#include <iostream>
#include <cstring>
using namespace std;
#include "strng2.h"

// class methods

String::String(const char * s) // создание класса String на базе строки С
{
 len = strlen(s);
 str = new char[len + 1]; // распределение памяти
 strcpy(str, s); // инициализация указателя
}

String::String() // конструктор, заданный по умолчанию
{
 len = 0;
 str = new char[1];
 str[0] = '\0'; // строка, заданная по умолчанию
}

String::String(const String & st) // конструктор копирования
{
 len = st.len;
 str = new char[len + 1];
 strcpy(str, st.str);
}

String::~String() // деструктор
{
 delete [] str; // требуется
}

// присваивание значения String
String & String::operator=(const String & st)
{
 if (this == &st)
 return *this;
 delete [] str;
 len = st.len;
 str = new char[len + 1];
 strcpy(str, st.str);
}

```

```

 return *this;
}

// присваивание строки С объекту String
String & String::operator=(const char * s)
{
 delete [] str;
 len = strlen(s);
 str = new char[len + 1];
 strcpy(str, s);
 return *this;
}

// true, если st1 следует за st2 в упорядоченной последовательности
bool operator>(const String &st1, const String &st2)
{
 if (strcmp(st1.str, st2.str) > 0)
 return true;
 else
 return false;
}

// true, если st1 предшествует st2 в упорядоченной последовательности
bool operator<(const String &st1, const String &st2)
{
 if (strcmp(st1.str, st2.str) < 0)
 return true;
 else
 return false;
}

// дружественные конструкции
// true, если st1 является таким же, как st2
bool operator==(const String &st1, const String &st2)
{
 if (strcmp(st1.str, st2.str) == 0)
 return true;
 else
 return false;
}

// отображает строку
ostream & operator<<(ostream & os, const String & st)
{
 os << st.str;
 return os;
}

// быстрый и черновой ввод String
istream & operator>>(istream & is, String & st)
{
 char temp[80];
 is.get(temp, 80);
 if (is)
 st = temp;
 while (is && is.get() != '\n')
 continue;
 return is;
}

```

**Листинг 11.9 Программа sayings1.cpp.**

```

// sayings1.cpp -- применяет расширенный класс string
// Компиляция с помощью strng2.cpp
#include <iostream>
using namespace std;
#include "strng2.h"

```

```

const ArSize = 10;
const MaxLen = 81;
int main()
{
 String name;
 cout << "Hi, what's your name?\n>> ";
 cin >> name;

 cout << name << ", please enter up to " << ArSize << " short sayings
 << empty line to quit:\n";
 String sayings[ArSize]; // массив объектов
 char temp[MaxLen]; // временное хранилище строк
 int i;
 for (i = 0; i < ArSize; i++)
 {
 cout << i+1 << ": ";
 cin.get(temp, MaxLen);
 while (cin && cin.get() != '\n')
 continue;
 if (!cin || temp[0] == '\0') // пустая строка?
 break; // i не имеет приращений
 else
 sayings[i] = temp; // перегружаемое присваивание
 }
 int total = i; // общие # строк для чтения

 cout << "Here are your sayings:\n";
 for (i = 0; i < total; i++)
 cout << sayings[i] << "\n";

 int shortest = 0;
 int first = 0;
 for (i = 1; i < total; i++)
 {
 if (sayings[i].length() < sayings[shortest].length())
 shortest = i;
 if (sayings[i] < sayings[first])
 first = i;
 }
 cout << "Shortest saying:\n" << sayings[shortest] << "\n";
 cout << "First alphabetically:\n" << sayings[first] << "\n";
 return 0;
}

```



### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Более ранние версии `get (char *, int)` не приводят к появлению значения `false` при чтении пустой строки. Во время работы с этими версиями при вводе пустой строки первым символом строки будет нуль. В этом примере применяется следующий код:

```

if (!cin || temp[0] == '\0') // пустая строка?
break; // i не имеет приращений

```

Если реализация проходит по текущему стандарту, при первой проверке в операторе `if` осуществляется тестирование пустой строки, а при второй — тестирование пустой строки для более ранних реализаций.

Программа запрашивает пользователя о необходимости ввода десяти утверждений. Каждое утверждение прочитывается в текущем массиве символов и затем копируется в объект `String`. Если пользователь выполняет ввод пустой строки, оператор `break` прерывает цикл ввода. После кодирования данных ввода программа использу-

ет функции-элементы `length()` и `operator<()` для локализации самой короткой строки и расположенной в алфавитном порядке самой первой строки. Ниже показаны результаты выполнения программы:

```

Hi, what's your name?
>> Misty Gutz
Misty Gutz, please enter up to 10 short
sayings <empty line to quit>:
1: a fool and his money are soon parted
2: penny wise, pound foolish
3: the love of money is the root of much
evil
4: out of sight, out of mind
5: absence makes the heart grow fonder
6: absinthe makes the hart grow fonder
7:
Here are your sayings:
a fool and his money are soon parted
penny wise, pound foolish
the love of money is the root of much evil

```

```

out of sight, out of mind
absence makes the heart grow fonder
absinthe makes the hart grow fonder
Shortest saying:
penny wise, pound foolish
First alphabetically:
a fool and his money are soon parted

```

## Применение оператора new в конструкторах

Как можно заметить, при использовании оператора **new** для инициализации элементов указателя объекта нужно быть предельно внимательным. В частности, необходимо выполнить следующие действия:

- Если оператор **new** применяется для инициализации элемента указателя в конструкторе, в деструкторе необходимо применять оператор **delete**.
- Следует обратить внимание на совместимость операторов **new** и **delete**. Рассматривайте вместе **new** и **delete**, а также **new [ ]** и **delete [ ]**.
- Если имеется несколько конструкторов, они применяют оператор **new** аналогичным образом. В этом случае везде должны быть скобки либо они везде должны отсутствовать. Существует только один деструктор, поэтому все конструкторы должны быть совместимы с этим деструктором. Однако можно инициализировать указатель с помощью оператора **new** в одном из конструкторов и с помощью нулевого указателя (**NULL** или 0) — в другом конструкторе. Дело в том, что к нулевому указателю удобно применять оператор **delete** (со скобками или без них).

### NULL ИЛИ 0?

Нулевой указатель можно представить как **0** или как **NULL**. Эта символическая константа определяется как **0** во многих заголовочных файлах. Программисты, работающие на C, часто применяют **NULL** вместо **0**. **NULL** указывается как визуальное напоминание о том, что значение является значением указателя. Точно так же применяется '**\0**' вместо **0** для нулевого символа в качестве визуального напоминания о том, что это значение является символом. При работе с языком C++ принято использовать просто **0** вместо эквивалентного **NULL**.

- Нужно определить конструктор копирования, который инициализирует один объект для другого путем глубокого копирования. Обычно конструктор может быть представлен следующим примером:

```

String::String(const String & st)
{
 num_strings++; // обработка статического
 // элемента и обновление,
 // если это необходимо
 len = st.len; // такая же длина

```

```

str = new char [len + 1]; // распределение
 // пространства
strcpy(str, st.str); // копирование строки
 // в новое место
}

```

- В частности, конструктор копирования должен распределить пространство, чтобы туда вошли скопированные данные, а не только адреса данных. Следует также обновить все те статические элементы классов, значения которых изменяются под влиянием процесса.
- Необходимо определить оператор присваивания, который копирует один объект в другой, выполняя глубокое копирование. Обычно метод класса моделирует следующий пример:

```

String & String::operator=(const String & st)
{
 if (this == &st) // объект, присвоенный
 // самому себе
 return *this; // все сделано
 delete [] str; // свободная старая строка
 len = st.len;
 str = new char [len + 1]; // получено
 // пространство для новой строки
 strcpy(str, st.str); // копирование строки
 return *this; // возвращение ссылки
 // вызывающему объекту
}

```

- В частности, метод должен производить проверку на самоприсваивание; следует освободить память, которая ранее отмечалась указателем элемента; происходит копирование данных, а не только адреса данных; вызывающему объекту должна возвращаться ссылка.

Следующий листинг содержит два примера, относящиеся к тому, что не следует делать, и один пример удачного конструктора:

```

String::String()
{
 str = "default string"; // отсутствует
 // new []
 len = strlen(str);
}

String::String(const char * s)
{
 len = strlen(s);
 str = new char; // отсутствует []
 strcpy(str, s); // отсутствует свободное
 // место
}

String::String(const String & st)
{
 len = st.len;
 str = new char[len + 1]; // хорошо,
 // распределяется пространство
 strcpy(str, st.str); // хорошо,
 // копируется значение
}

```

Первый конструктор ошибочно использует оператор `new` для инициализации указателя `str`. Деструктор, будучи вызван для объекта, заданного по умолчанию, применит оператор `delete` к `str`. Результат применения оператора `delete` к указателю, который не инициализирован с помощью оператора `new`, будет неопределенным, но, возможно, не совсем удачным. Ниже показаны те действия, которые приведут к успеху:

```
String::String()
{
 len = 0;
 str = new char[1]; // применяется new
 // с []
 str[0] = '\0';
}

String::String()
{
 len = 0;
 str = NULL; // или эквивалент str = 0;
}

String::String()
{
 static const char * s = "C++";
 // инициализируется только один раз
 len = strlen(s);
 str = new char[len +1]; // применяется new
 // с []
 strcpy(str, s);
}
```

Второй конструктор в фрагменте исходного кода использует оператор `new`, но при запросе необходимого объема памяти происходит сбой. Поэтому `new` вернет блок, содержащий пространство только для одного символа. Если пытаться скопировать в это место более длинную строку, можно столкнуться с определенными проблемами. Кроме того, использование `new` без скобок не соответствует корректной форме других конструкторов.

Третий конструктор функционирует превосходно.

Наконец, ниже показан деструктор, который не функционирует корректно с конструкторами, определенными ранее:

```
String::-String()
{
 delete str; // плохо, должен быть
 // оператор delete [] str;
}
```

Деструктор некорректно использует оператор `delete`. Поскольку конструкторы запрашивают массивы символов, деструктор должен удалить массив.

## Применение указателей при работе с объектами

В программах, написанных на C++, часто используются указатели на объекты, поэтому сделаем небольшое

практическое отступление. В листинге 11.9 применялись значения массива индексов для отслеживания самой короткой строки и первой строки, расположенной в алфавитном порядке. Другой подход состоит в применении указателей для того, чтобы отметить текущие заголовки в этих категориях. В листинге 11.10 реализуется этот подход с помощью двух указателей для `String`. Первоначально указатель `shortest` отмечает первый объект в массиве. Всякий раз, когда программа обнаруживает объект с более короткой строкой, происходит вторичная установка `shortest` для указания на этот объект. Аналогично первый указатель отслеживает ту строку, которая в алфавитном порядке находится выше. Обратите внимание на то, что эти два указателя не создают новых объектов; они только отмечают имеющиеся. Поэтому они не применяют оператор `new` для распределения дополнительной памяти.

Для разнообразия программа использует указатель, который выполняет отслеживание нового объекта:

```
String * favorite =
 new String(sayings[choice]);
```

Здесь указатель `favorite` поддерживает доступ только к тем объектам, которые лишены имени и созданы оператором `new`. Этот особый синтаксис показывает инициализацию нового объекта `String` с помощью объекта `sayings [choice]`. Таким образом, вызывается конструктор копирования, поскольку тип аргумента для конструктора копирования (`const String &`) соответствует значению инициализации (`sayings [choice]`). Программа применяет функции `rand()`, `rand()` и `time()` для выделения выбираемого значения случайным образом.

### ИНИЦИАЛИЗАЦИЯ ОБЪЕКТА С ПОМОЩЬЮ ОПЕРАТОРА NEW

В общем, если `Class_name` является классом и если `value` представляет тип `Type_name`, оператор

```
Class_name * pclass = new Class_name(value);
```

вызывает

```
Class_name (Type_name);
```

конструктор. Здесь могут появиться тривиальные преобразования типа

```
Class_name (const Type_name &);
```

Обычные преобразования, которые вызываются при достижении соответствия прототипа, например, как от `int` до `double`, будут реализовываться до тех пор, пока не проявится двусмысленность положения. Инициализация в форме

```
Class_name * ptr = new Class_name;
```

вызывает конструктор, заданный по умолчанию.

**Листинг 11.10 Программа sayings2.cpp.**

```

// sayings2.cpp -- применяет указатели для объектов
// Компиляция с помощью strng2.cpp
#include <iostream>
using namespace std;
#include <cstdlib> // (или stdlib.h) для rand(), srand()
#include <ctime> // (или time.h) для time()
#include "strng2.h"
const ArSize = 10;
const MaxLen = 81;
int main()
{
 String name;
 cout << "Hi, what's your name?\n>> ";
 cin >> name;

 cout << name << ", please enter up to " << ArSize
 << " short sayings <empty line to quit>:\n";
 String sayings[ArSize];
 char temp[MaxLen]; // временное хранилище строки
 int i;
 for (i = 0; i < ArSize; i++)
 {
 cout << i+1 << ": ";
 cin.get(temp, MaxLen);
 while (cin && cin.get() != '\n')
 continue;
 if (!cin || temp[0] == '\0') // пустая строка?
 break; // i не имеет приращений
 else
 sayings[i] = temp; // перегруженное присваивание
 }
 int total = i; // общее количество считываемых строк
 cout << "Here are your sayings:\n";
 for (i = 0; i < total; i++)
 cout << sayings[i] << "\n";
// применение указателей для отслеживания самой короткой среди первых строк
 String * shortest = &sayings[0]; // инициализация первого объекта
 String * first = &sayings[0];
 for (i = 1; i < total; i++)
 {
 if (sayings[i].length() < shortest->length())
 shortest = &sayings[i];
 if (sayings[i] < *first) // сравнение значений
 first = &sayings[i]; // присваивание адреса
 }
 cout << "Shortest saying:\n" << * shortest << "\n";
 cout << "First alphabetically:\n" << * first << "\n";
 srand(time(0));
 int choice = rand() % total; // выберите индекс случайным образом
// примените оператор new для создания, инициализации нового объекта
 String * favorite = new String(sayings[choice]);
 cout << "My favorite saying:\n" << *favorite << "\n";
 delete favorite;
 return 0;
}

```

**ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**

Более ранние реализации компилятора могут требовать включения `stdlib.h` вместо `cstdlib` и `time.h` вместо `ctime`.

Результаты выполнения программы:

`Hi, what's your name?`

`>> Kirt Rood`

`Kirt Rood, please enter up to 10 short`

```

sayings <empty line to quit>
1: a friend in need is a friend indeed
2: neither a borrower nor a lender be
3: a stitch in time saves nine
4: a niche in time saves stine
5: it takes a crook to catch a crook
6: cold hands, warm heart
7:

Here are your sayings:
a friend in need is a friend indeed
neither a borrower nor a lender be
a stitch in time saves nine
a niche in time saves stine
it takes a crook to catch a crook
cold hands, warm heart
Shortest saying:
cold hands, warm heart
First alphabetically:
a friend in need is a friend indeed
My favorite saying:
a stitch in time saves nine

```

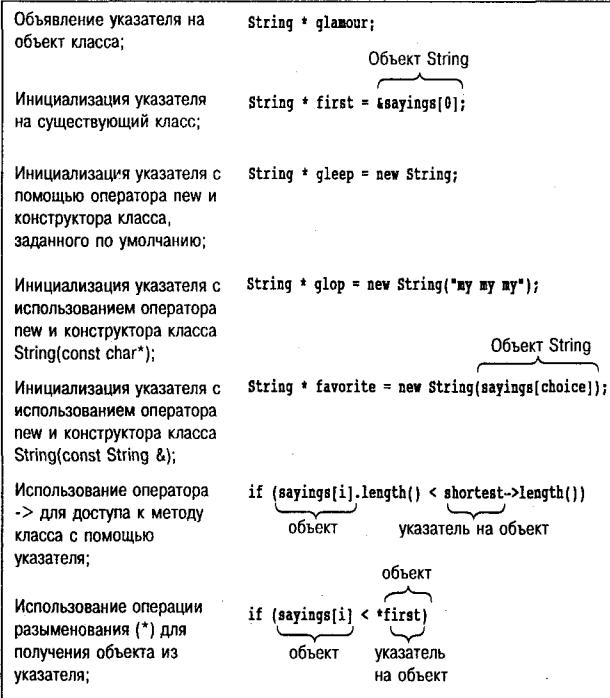
Следует обратить внимание на некоторые моменты, связанные с применением указателей на объекты (также обратите внимание на рис. 11.6).

- Вы объявляете указатель объекта с помощью обычного обозначения:

```
String * glamour;
```

- Можно инициализировать указатель для указания имеющегося объекта:

```
String * first = &sayings[0];
```

- Можно инициализировать указатель с помощью оператора new; это приводит к созданию нового объекта:

```
String * favorite =
 new String(sayings[choice]);
```

- Применение оператора new к классам вызывает соответствующий конструктор класса для инициализации заново созданного объекта:

```
// вызывает заданный по умолчанию
// конструктор
String * gleep = new String;
```

- Вызов конструктора String(const char \*)

```
String * glop = new String("my my my");
```

- Вызов конструктора String(const String &)

```
String * favorite =
 new String(sayings[choice]);
```

- Вы применяете оператор -> для доступа к методу класса с помощью указателя:

```
if (sayings[i].length() < shortest->length())
```

- Вы применяете операцию для разыменования указателя (\*) к указателю объекта для получения объекта:

```
if (&sayings[i] < *first) // сравните
 // значения объекта
first = &sayings[i]; // присваивание
 // адреса объекта
```

## Обзор технических методов

А теперь приступим к рассмотрению методик программирования, которые позволяют решать различные проблемы, связанные с классами. Возможно, при работе с классами вам придется столкнуться с большим количеством затруднений. Итак, кратко опишем несколько методик и условия их применения.

### Перегрузка операции <<

Переопределим операцию << таким образом, чтобы применять ее вместе с cout для отображения содержимого объектов. Определим операторную дружественную функцию в следующем виде:

```
ostream & operator<<(ostream & os,
 const c_name & obj)
{
 os << ... ; // отображение содержимого
 // объекта
 return os;
}
```

Здесь *c\_name* представляет наименование класса. Если класс поддерживает общедоступные методы, которые возвращают требуемое содержимое, эти методы можно использовать в операторной функции и избавить ее от дружественного статуса.

РИСУНОК 11.6 Указатели и объекты.

## Функции преобразования

Для конвертирования отдельного значения к типу класса, создайте конструктор класса в прототипе следующего вида:

```
c_name(type_name value);
```

Здесь `c_name` представляет наименование класса, а `type_name` — наименование типа, который необходимо преобразовать.

Для преобразования типа класса к некоторому другому типу создайте функцию-элемент, которая имеет следующий прототип:

```
operator type_name();
```

Несмотря на то что функция не имеет объявленного возвращаемого типа, она возвращает значение требуемого типа.

Не забывайте, что необходимо быть особенно внимательным, применяя функции преобразований. При объявлении конструктора для того, чтобы предотвратить его использование в неявных преобразованиях, можно воспользоваться ключевым словом `explicit`.

## Классы, конструкторы которых применяют оператор new

Классы, которые применяют оператор `new` для распределения памяти, и выбираются с помощью элемента класса, требуют соблюдения определенных мер предосторожности. (Конечно, далее эти меры предосторожности будут выделены особо, но соответствующие правила довольно сложны для запоминания. Это связано с тем, что компилятор эти правила неизвестны. Поэтому, компилятор нельзя применять для обнаружения ошибок.)

1. Каждый элемент класса, который указывает на память, распределенную с помощью оператора `new`, должен располагать оператором `delete`, который применяется к нему в деструкторе класса. Это приводит к освобождению распределенной памяти.
2. Если деструктор освобождает память, применяя оператор `delete` к указателю, который является элементом класса, тогда каждый конструктор для этого класса должен инициализировать этот указатель. Для достижения этого либо применяется оператор `new`, либо указатель устанавливается равным нулю.
3. Конструкторы должны основываться либо на `new [ ]`, либо на `new`, но ни в коем случае не следует смешивать оба оператора. Деструктор должен применять оператор `delete [ ]`, если конструкторы используют `new [ ]`, и необходимо применять оператор `delete`, если конструкторы используют оператор `new`.

4. Вместо копирования указателя на существующую память следует определить конструктор копирования, который распределяет новую память. Такой подход позволит программе инициализировать один класс объекта другим классом. Конструктор обычно должен иметь следующий прототип:

```
className(const className &)
```

5. Следует определить функцию-элемент класса, которая перегружает оператор присваивания и имеет следующий вид определения функции (здесь `c_pointer` представляет собой элемент класса `c_name` и имеет тип указателя-на-`type_name`):

```
c_name & c_name::operators (const c_name & en)
{
 if (this == & cn_)
 return *this; // выполняется, если
 // осуществляется самоприсваивание
 delete c_pointer;
 c_pointer = new type_name[size];
 // тогда копированные данные отмечаются
 // с помощью сн.c_pointer расположение,
 // отмеченное с помощью c_pointer
 ...
 return *this;
}
```

## Моделирование очереди

Применим наши глубокие представления о классах к решению задач программирования. Банк Bank of Heather выразил желание открыть автоответчик в супермаркете Food Heap. Руководство Food Heap побеспокоится о линиях связи для автоответчика, которые оказывают влияние на показатели обслуживания. Возможно, возникнет необходимость в ограничении числа абонентов, которые могут одновременно обращаться к автоответчику. Сотрудники Bank of Heather желают оценить период времени ожидания на линии. Ваша задача состоит в том, чтобы разработать программу, которая имитирует сложившуюся ситуацию. Причем желательно, чтобы руководство смогло оценить эффект от внедрения автоответчика.

Самым естественным способом представления задачи является использование очереди заказчиков. Очередь представляет собой абстрактный тип данных (ADT), который содержит упорядоченную последовательность элементов. Новые элементы добавляются в конец очереди, а старые удаляются из начала очереди. Очередь немного похожа на стек. Отличие состоит в том, что дополнения и исключения в стеке выполняются в самом конце. Это позволяет заключить, что стек является структурой LIFO (last in-first out — "последним зашел, первым вышел"), в то время как очередь представляет собой структуру FIFO (first in-first out — "первым зашел,

первым вышел"). По своему замыслу очередь имитирует ленту кассового аппарата или автоответчик. Поэтому очередь идеально подходит к данной задаче. Итак, в одной части вашего проекта будет определен класс **Queue**.

Элементами очереди будут заказчики. Представитель Bank of Heather утверждает, что в среднем обслуживание трети заказчиков занимает одну минуту. В других случаях на обслуживание одной трети заказчиков тратится две минуты, и треть заказчиков требуют на свое обслуживание три минуты. Более того, заказчики обращаются с различной периодичностью, но среднее число пользователей, которые обращаются в течение часа, постоянно. В следующих двух частях вашего проекта будут введены заказчики, которые представляют класс. Затем части программы объединяются вместе, имитируя взаимодействие между заказчиками и очередью (рис. 11.7).

## Класс Queue

Первоначальной задачей будет разработка класса **Queue**. Приведем список существующих атрибутов очереди:

- Имеется упорядоченная последовательность элементов.
- Существует ограничение на количество элементов, образующих очередь.
- Нужно научиться создавать пустую очередь.
- Потребуется реализовать проверку, является ли очередь пустой.
- Нужно научиться проверять, является ли очередь заполненной.
- Потребуется реализовать добавление элементов в конец очереди.

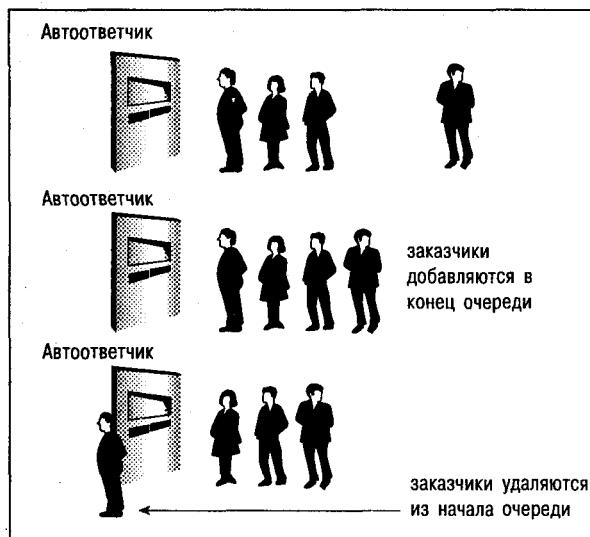


РИСУНОК 11.7 Очередь.

- Необходимо научиться удалять элемент из начала очереди.
- Потребуется определять количество элементов в очереди.

Обычно при создании класса необходимо разработать общедоступный интерфейс и приватную реализацию.

## Интерфейс

Атрибуты очереди предлагают для класса очереди следующий общедоступный интерфейс:

```
class Queue
{
 enum {Q_SIZE = 10};
private:
 // приватное представление будет
 // разработано позднее
public:
 Queue(int qs = Q_SIZE); // создайте
 // очередь с пределом qs
 ~Queue();
 bool isempty() const;
 bool isfull() const;
 int queuecount() const;
 bool enqueue(const Item &item);
 // добавление элемента в конец очереди
 bool dequeue(Item &item); // удаление
 // элемента из начала очереди
};
```

Конструктор создает пустую очередь. По умолчанию в очередь могут входить до 10 элементов, но это обстоятельство может быть изменено с помощью явной инициализации аргументов:

```
Queue line1; // очередь с ограничением в
 // 10 элементов
Queue line2(20); // очередь с ограничением
 // в 20 элементов
```

С помощью очереди можно применить **typedef** для определения **Item**. (В главе 13 рассматривается, каким образом вместо этого можно применить шаблоны класса.)

## Реализация

Приступим к реализации интерфейса. Сначала нужно определить, каким образом представить данные очереди. Один вариант заключается в применении оператора **new** для создания динамического массива с необходимым количеством элементов. Однако массивы не совсем точно соответствуют операциям очереди. Например, удаление элемента из начала массива должно сопровождаться смещением каждого оставшегося элемента на одну единицу по направлению к началу. Если же вы желаете выполнить все необходимое с особой тщательностью, нужно воспринимать массив как круговой. Однако все же *связанный список* в большей степени подходит к тем требованиям, которые выдвигает очередь. Связанный

список состоит из последовательности узлов. Каждый узел содержит информацию, которая предусмотрена списком плюс указатель для следующего узла списка. В этой очереди каждый фрагмент данных будет иметь значение типа **Item**, и для представления узла можно использовать структуру:

```
struct Node
{
 Item item; // данные, которые хранятся в
 // узле
 struct Node * next; // указатель для
 // следующего узла
};
```

На рис. 11.8 показан связанный список. Определенная форма связанных списков называется *отдельным связанным списком*, поскольку каждый узел имеет отдельную связь либо указатель, либо другой узел. Если имеется адрес первого узла, можно следовать за указателями к каждому последующему узлу списка. В общем, указатель последнего узла в списке устанавливается в **NULL** (или в **0**). Это позволяет сделать вывод о том, что далее узлов нет. Для отслеживания связанных списков следует знать адрес первого узла. Чтобы указать на начало списка, можно воспользоваться элементом данных класса **Queue**. В принципе, это и есть вся необходимая информация, с ее помощью можно просмотреть последовательность узлов на предмет обнаружения какого-либо узла. Однако, поскольку в очереди новый элемент всегда добавляется в конец, удобно, чтобы элемент данных указывал и на последний узел (рис. 11.9). Кроме того, элементы данных можно использовать для отслеживания максимального числа элементов, которым разрешается находиться в очереди. Можно уточнять и текущее число элементов. Таким образом, приватная часть объявления класса может иметь следующий вид:

```
class Queue
{
// определения, касающиеся классов
// Узел представляет собой определение
// вложенной структуры, которое является
// локальным для этого класса
 struct Node { Item item; struct Node *
 next;};
 enum {Q_SIZE =10};

private:
 Node * front; // указатель начала Queue
 Node * rear; // указатель конца Queue
 int items; // текущее количество
 // элементов в Queue
 const int qsize; // максимальное
 // количество элементов в Queue
...
public:
//...
};
```

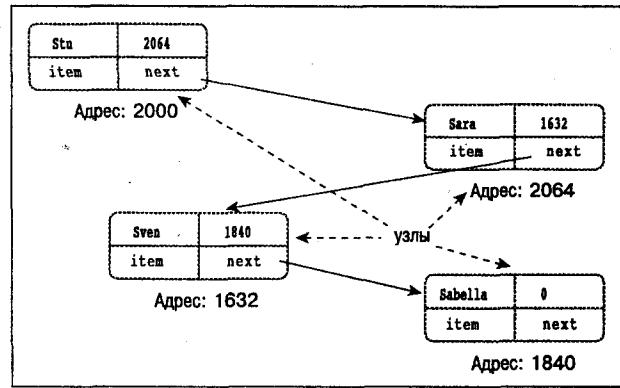


РИСУНОК 11.8 Связанный список.

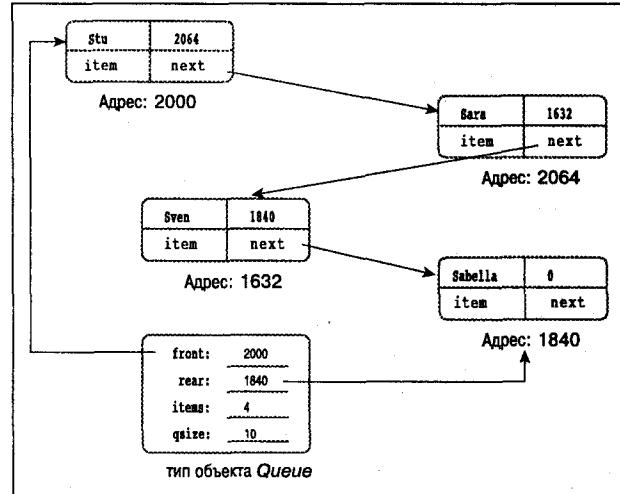


РИСУНОК 11.9 Объект Queue.

В объявлении используется новое свойство C++: возможность размещения в пределах класса вложенных структур или объявление класса внутри другого класса. Размещая объявление **Node** внутри класса **Queue**, вы позволяете ему влиять на весь класс. Иначе говоря, **Node** представляет собой тип, который можно использовать для объявления элементов класса, а также применять в методах класса в качестве наименования типа. Но этот тип ограничен классом.

Таким образом, не следует беспокоиться о том, что данное объявление **Node** вступит в конфликт с каким-либо глобальным объявлением или с **Node**, который объявлен в составе некоторого другого класса. Не все компиляторы в настоящее время поддерживают вложенные структуры и классы. Если ваш компилятор не поддерживает вложенные структуры и классы, следует определить структуру **Node** глобально и позаботиться о том, чтобы она была определена в масштабе файла.

## ВЛОЖЕННЫЕ СТРУКТУРЫ И КЛАССЫ

Структура, класс или нумерация, которая объявляется в пределах объявления класса, считается вложенной в класс. В этом случае она имеет масштаб класса. В результате выполнения подобного объявления не создается объект данных. Вместо этого указывается тип, который может применяться только в составе класса. Если объявление выполняется в приватном разделе класса, тогда объявленный тип может применяться только внутри класса. Если объявление реализуется в общедоступном разделе, то объявленный тип может применяться и вне класса с помощью оператора определения диапазона. Например, если **Node** объявлялся в общедоступном разделе класса **Queue**, тогда вне класса можно объявлять переменные типа **Queue::Node**.

После рассмотрения представления данных приступим к тем правилам, которым следуют методы класса.

### Методы класса

Конструктор класса должен поддерживать значения для элементов класса. Поскольку очередь изначально пуста, следует установить указатели начала и конца очереди в **NULL** (или **0**) и **items** — в значение **0**. Необходимо также установить максимальный размер очереди **qsize**, соответствующим аргументу конструктора **qs**. Ниже приводится реализация, которая не функционирует:

```
Queue::Queue(int qs)
{
 front = rear = NULL;
 items = 0;
 qsize = qs; // не допустимо!
}
```

Проблема состоит в том, что **qsize** имеет тип **const**, поэтому его можно инициализировать значением, но ему нельзя присвоить значение. В общем, при вызове конструктора создается объект еще до того, как будет выполнен код, который находится в скобках. Таким образом, при вызове конструктора **Queue (int qs)** программа сначала распределяет пространство для четырех переменных-элементов. Затем программа обращает внимание на скобки и выполняет обычное присваивание для размещения значений в выделенном пространстве. Поэтому, если необходимо инициализировать элемент данных **const**, нужно выполнить это при создании объекта, перед тем как процесс выполнения будет передан конструктору. C++ поддерживает с этой целью специальный синтаксис. Он именуется *списком инициализатора*. Список инициализатора включает перечень инициализаторов, разделенных запятыми, которым предшествуют двоеточия. Он размещается после закрывающихся круглых скобок в списке аргументов и перед открывающейся скобкой тела функции. Если элемент данных именуется **mdata** и если он инициализируется значением **val**, инициализатор имеет форму **mdata (val)**.

С помощью этого замечания можно создавать конструктор **Queue**, например, следующим образом:

```
Queue::Queue(int qs) : qsize(qs)
{
 front = rear = NULL;
 items = 0;
}
```

В общем случае первоначальное значение может вызывать константы и аргументы из списка аргументов конструктора. Методика не ограничивается инициализацией констант. Конструктор **Queue** также можно создать следующим образом:

```
Queue::Queue(int qs) : qsize(qs),
 front(NULL), rear(NULL), items(0)
{ }
```

Только конструкторы могут применять синтаксис списка-инициализатора. Как вы видите, следует применять этот синтаксис для элементов класса **const**, а также для элементов класса, которые объявляются как ссылки:

```
class Agency {...};
class Agent
{
private:
 Agency & belong; // необходимо применить
 // список инициализатора для
 // инициализации
 ...
};

Agent::Agent(Agency & a) : belong(a) {...}
```

Поэтому ссылки типа **const** могут инициализироваться только при их создании. Для простых элементов данных типа **front** и **items** не имеет большого значения, используется ли список инициализатора или же в теле функции выполняется присваивание. Однако, как показано в главе 13, более эффективно использовать список инициализатора для тех элементов, которые представляют собой объекты класса.

## СИНТАКСИС СПИСКА ИНИЦИАЛИЗАТОРА

Если **Classy** является классом и если **mem1**, **mem2** и **mem3** представляют собой элементы класса данных, конструктор класса может применять для инициализации элементов данных следующий синтаксис:

```
Classy::Classy(int n, int m) : mem1(n), mem2(0),
 mem3(n*m + 2)
{
 //...
}
```

Происходит инициализация **mem1** значением **n**, **mem2** — значением **0**, а **mem3** — значением **n\*m + 2**. В общем, эти инициализации осуществляются в том случае, если объект создан, и перед тем, когда выполняется какой-либо

код, находящийся в скобках. Обратите внимание на следующее:

- Эта форма может быть использована только с конструкторами.
- Следует применять эту форму для инициализации нестатического элемента данных **const**.
- Следует применять эту форму для инициализации элемента данных ссылки.

Элементы данных инициализируются в том порядке, в котором они отображаются в объявлении класса, а не в порядке перечисления инициализаторов.

### ПРЕДОСТЕРЕЖЕНИЕ

Синтаксис списка инициализатора нельзя использовать при работе с методами класса, которые отличаются от конструкторов.

Иногда форма инициализации, которая применяется в списке инициализатора, может применяться и еще где-либо. Поэтому по желанию можно заменить следующий программный код:

```
int games = 162;
double talk = 2.71828;
```

на код

```
int games(162);
double talk(2.71828);
```

Это позволит инициализировать встроенные типы данных, которые выглядят как инициализация объектов класса. Код для **isempty()**, **isfull()** и **queuesize()** прост. Если **items** соответствует 0, очередь является пустой. Если **items** соответствует **qsize**, очередь является заполненной. При возвращении количества элементов получаем ответ на вопрос: сколько элементов в очереди? Позже будет показан код заголовочного файла. Далее происходит добавление элемента в конец очереди. Ниже описывается один из подходов:

```
bool Queue::enqueue(const Item & item)
{
 if (isfull())
 return false;
 Node * add = new Node; // создание узла
 if (add == NULL)
 return false; // выход, если
 // недоступно
 add->item = item; // установка
 // указателей узла
 add->next = NULL;
 items++;
 if (front == NULL) // если очередь
 // пуста,
 front = add; // добавление элемента
 // в начало очереди
 else
 rear->next = add; // либо добавление
 // элемента в конец очереди
```

```
rear = add; // определяется конечная
 // точка для нового узла
return true;
}
```

Условно метод можно разделить на следующие фазы (рис. 11.10):

1. Прерывается выполнение, если очередь уже заполнена.
2. Создается новый узел, происходит прерывание, если этого нельзя сделать, например, если не удается выполнить запрос на выделение большего объема памяти.
3. В узле размещаются подходящие значения. В этом случае код копирует значение **Item** во фрагмент данных узла и устанавливает следующий указатель для узла в **NULL**. Таким образом, узел подготавливается для того, чтобы оказаться последним элементом в очереди.
4. Увеличение счетчика элементов (**items**) на один.
5. Добавление узла в конец очереди. Данный процесс состоит из двух частей. Первая часть заключается в привязывании узла к другим узлам в списке. Это достигается путем размещения указателя **next** текущего конечного узла таким образом, чтобы отмечался новый конечный узел. Во второй части устанавливается указатель **rear** для элемента **Queue**.

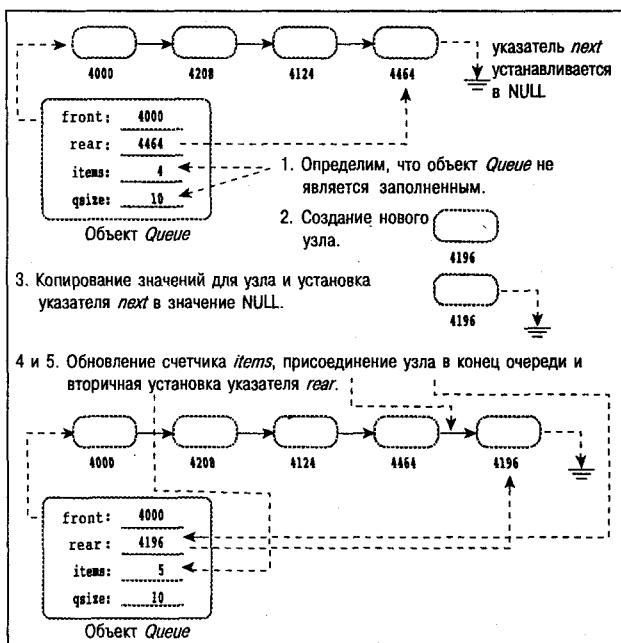


РИСУНОК 11.10 Постановка элемента в очередь.

Этот указатель отмечает новый узел таким образом, чтобы очередь могла получать доступ непосредственно к последнему узлу. Если очередь пуста, вам также следует установить указатель `front` для отметки нового узла. (Если имеется только один узел, то этот узел является и первым, и последним.)

Удаление элемента из начала очереди также занимает несколько этапов.

```
bool Queue::dequeue(Item & item)
{
 if (front == NULL)
 return false;
 item = front->item; // установка
 // элемента первым в очереди
 items--;
 Node * temp = front; // сохранение
 // местоположения первого элемента
 front = front->next; // установка начала к
 // следующему элементу
 delete temp; // удаление элемента,
 // который был первым
 if (items == 0)
 rear = NULL;
 return true;
}
```

Метод состоит из следующих этапов (рис. 11.11):

1. Прерывание процесса удаления элементов, если очередь уже пуста.
2. Поддержка первого пункта в очереди для вызывающей функции. С этой целью происходит копирование порции данных текущего узла `front` в переменную ссылки, которая передается методу.
3. Уменьшение на единицу счетчика элементов (`items`).
4. Сохранение расположения начального узла для удаления его в дальнейшем.
5. Удаление узла из очереди. Это происходит путем размещения указателя `front` элемента `Queue` для указания следующего узла, адрес которого определяется с помощью `front->next`.
6. Для экономии памяти удалите узел, который был первым.
7. Если список в настоящий момент является пустым, присвойте `rear` в значение `NULL`. (Указатель `front` в этом случае уже должен иметь значение `NULL`, после того как он устанавливался в `front->next`.)

Этап 4 необходим, поскольку на этапе 5 из памяти очереди удаляются сведения о том, какой узел является первым.

### Немного сведений о других методах классов

Нужны ли вам другие методы? Конструктор класса не использует оператор `new`, поэтому на первый взгляд ка-

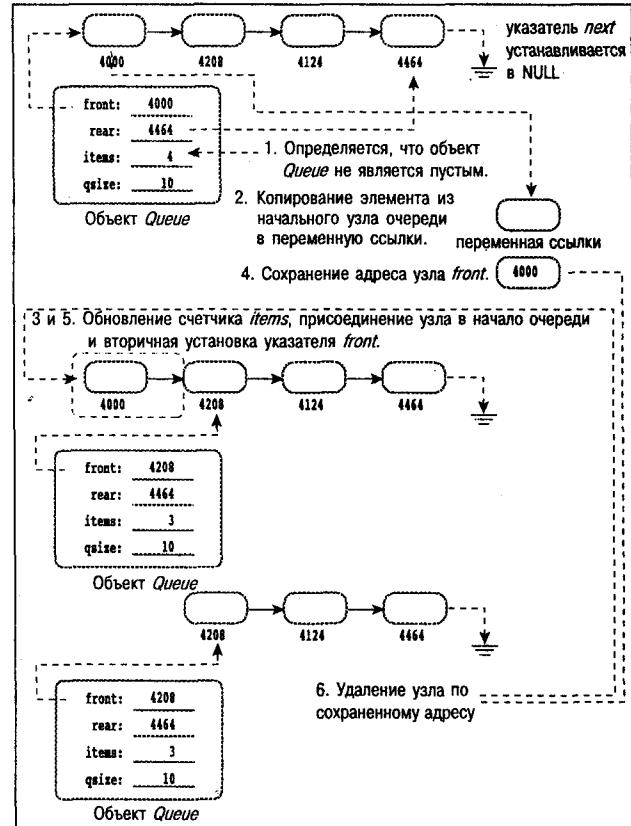


РИСУНОК 11.11 Удаление элемента из очереди.

жется, что при его появлении не стоит беспокоиться о специальных требованиях к тем классам, которые применяют `new` в конструкторах. Конечно, этот подход ошибочен, поскольку при добавлении объектов к очереди происходит вызов оператора `new` для создания новых узлов. Действительно, метод `dequeue()` наводит порядок путем удаления узлов. Но нет никакой уверенности в том, что по завершении этого метода очередь будет пустой. Поэтому класс нуждается в явном деструкторе, который удаляет все оставшиеся узлы. Ниже описывается реализация:

```
Queue::~Queue()
{
 Node * temp;
 while (front != NULL) // если очередь
 // еще не пуста
 {
 temp = front; // сохранение адреса
 // начального элемента
 front = front->next; // переустановка
 // указателя к следующему элементу
 delete temp; // удаление элемента,
 // который ранее
 // находился в начале
 }
}
```

Процесс начинается из начала списка и удаляет каждый узел по очереди.

Как вы видите, классы, применяющие оператор `new`, обычно нуждаются в явных конструкторах копирования и операторах присваивания, которые выполняют глубокое копирование. Так ли это происходит в данном случае? Первый вопрос, на который следует найти ответ, так ли необходимо заданное по умолчанию поэлементное копирование? Ответ отрицательный. Поэлементное копирование объекта `Queue` способствует образованию нового объекта, который указывает начало и конец этого же связанного списка в качестве исходного. Таким образом при добавлении элемента к копии объекта `Queue` изменяется общедоступный связанный список. Это довольно плохой результат. Но еще более плачевен тот факт, что обновляется лишь конечный указатель копии. В связи с этим с точки зрения исходного объекта список существенно нарушается. Ясно, что в этом случае процесс клонирования или копирования очередей нуждается в поддержке со стороны конструктора копирования и конструктора присваивания, которые реализуют глубокое копирование.

Конечно, можно задаться вопросом, зачем нужно копировать очередь? Да, возможно, на различных этапах моделирования вы пожелаете сохранить снимки очереди. Можно также предположить, что вам понадобится обеспечить идентичный ввод для двух различных стратегий. Действительно, может возникнуть необходимость в выполнении действий, которые приведут к разделению очереди. Иногда к подобным действиям прибегают сотрудники супермаркета, устанавливая дополнительное препятствие при входе в магазин. Аналогично может потребоваться объединить две очереди в одну или урезать очередь.

Но при данном моделировании подобные действия предприниматься не будут. Нельзя ли просто игнорировать эти обстоятельства и применять лишь имеющиеся методы? Конечно, можно. Однако через некоторое время может возникнуть необходимость в повторном применении очереди, причем уже с выполнением копирования. Вы можете забыть, что допустили оплошность и не позаботились о подходящем коде для выполнения копирования. Поэтому ваши программы будут выполнять компиляцию и функционировать, но никаких приемлемых результатов ожидать не приходится. В связи с этим желательно уделить больше внимания конструктору копирования и оператору присваивания, даже если они не нужны вам в настоящий момент.

К счастью, обычно все признают, что невыгодно сокращать объем выполняемой работы, если это может повлечь за собой сбои в дальнейшем функционировании

программы. Основная идея состоит в определении требуемых методов в качестве замены приватных методов:

```
class Queue
{
private:
 Queue(const Queue & q) : qsize(0) { }
 // преимущественное определение
 Queue & operator=(const Queue & q)
 { return *this; }
//...
};
```

Такой подход чреват проявлением двух эффектов. Во-первых, отвергаются заданные по умолчанию определения метода, которые в противном случае генерируются автоматически. Во-вторых, поскольку эти методы являются приватными, ими нельзя пользоваться повсеместно. Иначе говоря, если `nip` и `tuck` являются объектами `Queue`, компилятор не позволит выполнить следующие действия:

```
Queue snick(nip); // не разрешается
tuck = nip; // не разрешается
```

Поэтому для устранения непонятных сбоев в дальнейшей работе лучше отслеживать те ошибки компилятора, которые свидетельствуют о недоступности данных методов. Этот прием также полезен при определении класса, объекты которого не подлежат копированию.

Имеются ли какие-либо иные эффекты? Конечно. Напомним, что конструктор копирования вызывается в том случае, если объекты передаются (или возвращаются) с помощью значения. Однако, если вы воспользуетесь распространенной практикой и будете передавать объекты как ссылки, все проблемы будут устранины. Конструктор копирования применяется для создания и других временных объектов. Но в определении класса `Queue` недостает задания действий, приводящих к появлению временных объектов, типа операции по перегрузке дополнительного оператора.

## Класс `Customer`

Теперь необходимо образовать класс клиентов (`customer`). Клиент автоответчика обладает многими свойствами, например, имеет имя, учетный номер и учетную запись. Однако эти свойства необходимы для моделирования лишь в том случае, если клиент присоединяется к очереди и требуется время для входящего уведомления. Если в процессе моделирования образуется новый клиент, программа создает новый объект клиента, где будет храниться информация о времени прибытия клиента и значение для времени формирования входящего уведомления. Последнее значение генерируется случайным образом. Если клиент оказывается в начале очереди, программа отметит начальное время

и вычтет время, затраченное на постановку в очередь. Таким образом, будет получено время ожидания клиента. Ниже показано, каким образом можно определить и реализовать класс **Customer**:

```
class Customer
{
private:
 long arrive; // время прибытия клиента
 int processtime; // время обработки для
 // клиента

public:
 Customer() { arrive = processtime = 0; }
 void set(long when);
 long when() const { return arrive; }
 int ptime() const
 { return processtime; }
};

void Customer::set(long when)
{
```

Листинг 11.11 Объявления классов queue.h.

```
// queue.h -- интерфейс для очереди
#ifndef _QUEUE_H_
#define _QUEUE_H_
// Эта очередь содержит элементы Customer
class Customer
{
private:
 long arrive; // время прибытия клиента
 int processtime; // время обработки для клиента

public:
 Customer() { arrive = processtime = 0; }
 void set(long when);
 long when() const { return arrive; }
 int ptime() const { return processtime; }
};

typedef Customer Item;

class Queue
{
// определение диапазона доступа класса
 // Node представляет собой определение вложенной структуры, локальной для этого класса
 struct Node { Item item; struct Node * next;};
 enum {Q_SIZE = 10};

private:
 Node * front; // указатель начала Queue
 Node * rear; // указатель конца Queue
 int items; // текущее число элементов в Queue
 const int qsize; // максимальное число элементов в Queue
 // преимущественные определения, препятствующие публичному копированию
 Queue(const Queue & q) : qsize(0) {}
 Queue & operator=(const Queue & q) { return *this; }

public:
 Queue(int qs = Q_SIZE); // образование очереди с пределом qs
 ~Queue();
 bool isempty() const;
 bool isfull() const;
 int queuecount() const;
 bool enqueue(const Item &item); // добавление элемента в конец очереди
 bool dequeue(Item &item); // удаление элемента из начала очереди
};
```

```
 processtime = rand() % 3 + 1;
 arrive = when;
}
```

Заданный по умолчанию конструктор создает нулевой клиент. Функция-элемент **set()** устанавливает время прибытия для соответствующего аргумента и случайным образом выбирает значение для времени обработки. Это значение содержится в интервале от 1 до 3.

В листинге 11.11 показаны все объявления классов **Queue** и **Customer**, а в листинге 11.12 — реализация этих методов.

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Возможно, что ваш компилятор не имеет реализации типа переменной **bool**. В этом случае вместо **bool** можно использовать **int**, вместо **false** — 0, вместо **true** — 1. Возможно, вам придется воспользоваться **stdlib.h** вместо более новой реализации **cstdlib**.

## Листинг 11.12 Программа queue. cpp.

```

// queue.cpp -- методы Queue и Customer
#include "queue.h"
#include <cstdlib> // (или stdlib.h) для rand()

// метод Queue
Queue::Queue(int qs) : qsize(qs)
{
 front = rear = NULL;
 items = 0;
}

Queue::~Queue()
{
 Node * temp;
 while (front != NULL) // пока очередь еще не пуста
 {
 temp = front; // сохранение адреса начального элемента
 front = front->next; // вторичная установка указателя для следующего элемента
 delete temp; // удаление элемента, который был начальным ранее
 }
}

bool Queue::isempty() const
{
 return items == 0;
}

bool Queue::isfull() const
{
 return items == qsize;
}

int Queue::queuecount() const
{
 return items;
}

// добавление элемента к очереди
bool Queue::enqueue(const Item & item)
{
 if (isfull())
 return false;
 Node * add = new Node; // создание узла
 if (add == NULL)
 return false; // оставить, если недоступно
 add->item = item; // установка указателей узла
 add->next = NULL;
 items++;
 if (front == NULL) // если очередь пуста,
 front = add; // размещение элемента в начале очереди
 else
 rear->next = add; // размещение элемента где-то в конце очереди
 rear = add; // имеется конечная точка для нового узла
 return true;
}

// размещение начального элемента в переменной item и удаление из очереди
bool Queue::dequeue(Item & item)
{
 if (front == NULL)
 return false;
 item = front->item; // установка item к первому элементу в очереди
 items--;
 Node * temp = front; // сохранение местоположения первого элемента
 front = front->next; // вторичная установка начала к следующему элементу
 delete temp; // удаление элемента, который был ранее первым
}

```

```

if (items == 0)
 rear = NULL;
return true;
}

// метод customer

// время прибытия задается переменной when, а время обработки устанавливается
// случайным образом в диапазоне 1-3
void Customer::set(long when)
{
 processtime = rand() % 3 + 1;
 arrive = when;
}

```

## Моделирование

В настоящее время в вашем распоряжении имеются инструментальные средства, которые необходимы для моделирования функции автоответчика. Программа разрешит пользователю ввести три количественных значения: максимальный размер очереди, количество часов, в течение которых программа будет выполнять моделирование, и среднее число клиентов, которые обслуживаются ежечасно. Программа будет применять цикл, в котором каждый период цикла будет охватывать одну минуту. Во время каждой минуты этого периода программа выполняет следующее:

1. Определяет, прибыл ли новый клиент. После его прибытия добавьте заказчика к очереди, если имеется свободное пространство. В противном случае верните заказчика назад.
2. Если обработка не выполняется, обратите внимание на первого клиента в очереди. Уточните, как долго длилось ожидание, и установите счетчик `wait_time` для времени обработки, в котором нуждается новый заказчик.
3. Если обработка выполняется, уменьшите значение счетчика `wait_time` на одну минуту.
4. Проследите различные значения, в частности, количество обслуженных заказчиков, количество заказчиков, которые были отправлены назад, общее время, затраченное на ожидание на линии, и общую длину очереди.

После завершения цикла моделирования программа сообщит о различных статистических выводах.

Довольно интересно узнать, каким образом программа определяет, прибыл ли новый клиент. Предположим, что в среднем в течение часа прибывает 10 клиентов. Это значит, что в среднем в течение 6 минут прибывает один клиент. Программа вычисляет и сохраняет это значение с помощью переменной `min_per_cust`. Однако предполо-

жение о том, что в течение 6 минут прибывает ровно один клиент, нереально. С определенной долей уверенности можно заключить (по крайней мере, так будет происходить почти всегда), что существует случайный процесс, который описывает появление клиентов и согласуется с тем, что среднее применяемое значение — один клиент за 6 минут. Программа использует эту функцию для определения того, появится ли заказчик во время ожидания цикла:

```

bool newcustomer (double x)
{
 return (rand() * x / RAND_MAX < 1);
}

```

Рассмотрим, каким образом функционирует эта программа. Значение `RAND_MAX` определяется в файле `cstdlib` (ранее `stdlib.h`) и представляет самое большое значение, которое может возвратить функция `rand()` (0 является минимальным значением). Предположим, что `x`, среднее значение перерыва между прибытиями заказчиков, равно 6. Тогда значение `rand() * x / RAND_MAX` будет заключено в интервале от 0 до 6. В частности, это значение в среднем меньше единицы на протяжении шестой части отрезка времени. Однако, возможно, эта функция приведет к появлению двух заказчиков с интервалом в 1 минуту в один из промежутков времени и с интервалом в 20 минут — в другой промежуток времени. Этот подход приводит к появлению такой группировки, которая делает различным протекание реальных процессов и размеренную регулярность. В данном случае регулярность характеризуется появлением одного заказчика через каждые 6 минут. Этот метод не будет работать, если среднее время между прибытиями заказчиков сокращается и составляет менее 1 минуты. В процессе моделирования не происходит обработка сценария. Если же встретился подобный случай, следует изменить выборку. Возможно, в этом случае каждый цикл будет составлять 10 секунд.

## ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Некоторые компиляторы не определяют **RAND\_MAX**. Если вы столкнулись с подобной ситуацией, значение для **RAND\_MAX** можно определить самостоятельно, применяя **#define** или же **const int**. Если не получается разыскать задокументированное корректное значение, попробуйте применить самое большое из возможных значений **int**. Это значение предоставляется **INT\_MAX** в **climits** или в заголовочном файле **limits.h**.

В листинге 11.13 представлены подробности процесса моделирования. Если заниматься моделированием на больших промежутках времени, развивается умение оценивать средние значения на больших отрезках времени. При моделировании на коротких промежутках развивается умение оценивать средние значения на небольших отрезках времени.

### Листинг 11.13 Программа bank.cpp.

```
// bank.cpp -- применение интерфейса Queue
#include <iostream>
using namespace std;
#include <cstdlib> // для rand() и srand()
#include <ctime> // для time()
#include "queue.h"
const int MIN_PER_HR = 60;

bool newcustomer(double x); // это и есть новый клиент?

int main()
{
// установка объектов
 srand(time(0)); // случайная инициализация rand()

 cout << "Case Study: Bank of Heather Automatic Teller\n";
 cout << "Enter maximum size of queue: ";
 int qs;
 cin >> qs;
 Queue line(qs); // строка очереди поддерживается для клиентов,
 // количество которых описывается qs

 cout << "Enter the number of simulation hours: ";
 int hours; // часы моделирования
 cin >> hours;
// в процессе моделирования будет выполняться 1 цикл в минуту
 long cyclelimit = MIN_PER_HR * hours; // # циклов

 cout << "Enter the average number of customers per hour: ";
 double perhour; // среднее # ежечасных прибытий
 cin >> perhour;
 double min_per_cust; // среднее время между прибытиями
 min_per_cust = MIN_PER_HR / perhour;

 Item temp; // данные нового клиента
 long turnaways = 0; // уход от полной очереди
 long customers = 0; // присоединение к очереди
 long served = 0; // обслуживание во время моделирования
 long sum_line = 0; // совокупная длина линии
 int wait_time = 0; // период времени, в течение которого автоответчик свободен
 long line_wait = 0; // общее время в линии

// выполнение моделирования
 for (int cycle = 0; cycle < cyclelimit; cycle++)
 {
 if (newcustomer(min_per_cust)) // имеется вновь прибывший
 {
 if (line.isfull())
 turnaways++;
 else
 {
 customers++;
 temp.set(cycle);
 line.enqueue(temp); // cycle = время прибытия
 // добавление к линии новичка
 }
 }
 }
}
```

```

if (wait_time <= 0 && !line.isempty())
{
 line.dequeue (temp); // сопровождение следующего заказчика
 wait_time = temp.ptime(); // для wait_time минут
 line_wait += cycle - temp.when();
 served++;
}
if (wait_time > 0)
 wait_time--;
sum_line += line.queuecount();
}

// сообщение результатов
if (customers > 0)
{
 cout << "customers accepted: " << customers << '\n';
 cout << "customers served: " << served << '\n';
 cout << "turnaways: " << turnaways << '\n';
 cout << "average queue size: ";
 cout.precision(2);
 cout.setf(ios_base::fixed, ios_base::floatfield);
 cout.setf(ios_base::showpoint);
 cout << (double) sum_line / cyclelimit << '\n';
 cout << "average wait time: " << (double) line_wait / served << " minutes\n";
}
else
 cout << "No customers!\n";
return 0;
}

// x = среднее время (в минутах) между прибытиями клиентов
// возвращенное значение истинно, если клиент обнаруживается в эту минуту
bool newcustomer(double x)
{
 return (rand() * x / RAND_MAX < 1);
}

```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если ваш компилятор не имеет встроенного типа данных **bool**, используйте **int** вместо **bool**, **0** вместо **false**, и **1** вместо **true**. А может быть, вы примените **stdlib.h** и **time.h** вместо более новых **cstdlib** и **ctime**. Можно самостоятельно определить **RAND\_MAX**.

Ниже приводятся несколько примеров для более длинного промежутка времени:

```

Case Study: Bank of Heather Automatic
Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 100
Enter the average number of customers per
hour: 15
customers accepted: 1485
customers served: 1485
turnaways: 0
average queue size: 0.15
average wait time: 0.63 minutes

```

```

Case Study: Bank of Heather Automatic
Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 100
Enter the average number of customers per

```

```

hour: 30
customers accepted: 2896
customers served: 2888
turnaways: 101
average queue size: 4.64
average wait time: 9.63 minutes

```

```

Case Study: Bank of Heather Automatic
Teller
Enter maximum size of queue: 20
Enter the number of simulation hours: 100
Enter the average number of customers per
hour: 30
customers accepted: 2943
customers served: 2943
turnaways: 93
average queue size: 13.06
average wait time: 26.63 minutes

```

Заметим, что переход от 15 клиентов в час до 30 не приводит к изменению в 2 раза среднего времени ожидания, время ожидания увеличивается в соответствии с множителем 15. Если очередь удлиняется, то это только ухудшит дело. Однако процесс моделирования не предусматривает такой вариант — заказчики, устав от длительного ожидания, покидают очередь.

Ниже рассматриваются еще несколько примеров в качестве образца. Они иллюстрируют различные варианты, связанные с небольшими промежутками времени. При этом не столь важно, что количество заказчиков, которые появляются ежечасно, представляет собой постоянную величину.

**Case Study: Bank of Heather Automatic Teller**

```
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 114
customers served: 110
turnaways: 0
average queue size: 2.15
average wait time: 4.52 minutes
```

**Case Study: Bank of Heather Automatic Teller**

```
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 121
customers served: 116
turnaways: 5
average queue size: 5.28
average wait time: 10.72 minutes
```

**Case Study: Bank of Heather Automatic Teller**

```
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 112
customers served: 109
turnaways: 0
average queue size: 2.41
average wait time: 5.16 minutes
```

## Резюме

В этой главе рассматривается большое число аспектов, отражающих определение и применение классов. В некоторых из них представлены сложные понятия, о которых нелегко судить после первоначального ознакомления. Если определенные понятия остались для вас неясными или слишком сложными, не расстраивайтесь. Подобная реакция является закономерным итогом первоначального знакомства с языком C++. Чаше всего знакомство с такими важными понятиями, как конструкторы копирования, бывает вынужденным. Обычно к этим понятиям обращаются в том случае, если нельзя их проигнорировать. Поэтому часть материала этой главы может быть слишком далека от вашего понимания до тех пор, пока ваши представления не будут обогащены практикой. Подведем некоторые итоги.

В конструкторе класса можно применять оператор `new` в целях распределения памяти для данных и присваивания адреса памяти элементу класса. Тогда с помощью класса можно будет, например, обрабатывать строки различных размеров. При этом можно обойтись без привязки самого класса к фиксированному размеру массива. Применение оператора `new` в конструкторах класса также приводит к появлению потенциальных проблем при дальнейшем разрушении объекта. Если объект имеет указатели элемента, которые отмечают память, распределенную с помощью `new`, освобождение памяти проходит со следующими особенностями. При освобождении памяти, которая применяется для хранения объекта, не происходит автоматического освобождения памяти, отмеченной указателями объектного элемента. Поэтому при использовании оператора `new` в конструкторе класса для распределения памяти следует применить оператор `delete` в деструкторе класса для освобождения этой памяти. Таким образом, при отказе от объекта автоматически запускается процедура удаления отмеченной памяти.

Объекты, располагающие элементами, которые отмечают память, распределенную с помощью `new`, также имеют проблемы с инициализацией одного объекта к другому или с присваиванием одного объекта другому. По умолчанию C++ выполняет поэлементную инициализацию и присваивание. Это значит, что инициализированный или присвоенный чему-либо объект упорядочивается в соответствии с точными копиями исходных элементов объекта. Если исходный элемент отмечает блок данных, скопированный элемент отмечает аналогичный блок. Если программа случайно удаляет два объекта, класс деструктора будет дважды предпринимать попытку удаления этого же блока памяти, что является ошибочным. Для разрешения этого противоречия нужно определить специальную копию конструктора, которая повторно задаст инициализацию и перегрузит оператор присваивания. Новое определение должно образовать дубликаты всех отмеченных данных и располагать новым объектом, который укажет на копии. Таким образом, как старый, так и новый объект располагают отдельными, но идентичными ссылками, данные не перекрываются. Аналогично обращаются и с определением оператора присваивания. В любом случае цель состоит в создании глубокой копии, т.е. в копировании реальных данных, но не указателей на них.

C++ позволяет размещать внутри класса структуру, класс и перечисленные определения. Подобную вложенную природу имеет пространство класса. Это значит, что эти образования локальны для класса и не вступают в конфликт со структурами, классами и перечисленными образованиями аналогичного наименования, которые могут быть определены в любом месте.

C++ поддерживает специальный синтаксис для конструктиров класса, который может применяться для инициализации элементов данных. Этот синтаксис состоит из двоеточия, за которым следует перечень инициализаторов, разделенных запятыми. Этот синтаксис размещается после закрывающей скобки, содержащей аргументы конструктора, и перед открытой фигурной скобкой для тела функции. Каждый инициализатор состоит из наименования элемента, который инициализируется. Затем следуют скобки, куда входит значение инициализации. По идеи, эти инициализации имеют место при создании объекта и перед выполнением каких-либо операторов в теле функции. Синтаксис имеет следующий вид:

```
queue(int qs) : qsize(qs), items(0),
 front(NULL), rear(NULL) { }
```

Эта форма является обязательной, если элемент данных представляет собой нестатический элемент `const` или ссылку.

Как вы уже заметили, работа с классами требует тщательности и повышенного внимания. Нужно концентрировать больше усилий на детализации, чем это может потребоваться при работе с простыми структурами C-стиля. В свою очередь, вы можете получить большой эффект от реализации потенциальных возможностей.

## Вопросы для повторения

1. Предположим, что класс `String` включает следующие приватные элементы:

```
class String
{
private:
 char * str; // указывает на строку,
 // которая размещена с
 // помощью оператора new
 int len; // содержит длину строки
//...
};
```

a. Что неверно в конструкторе, заданном по умолчанию?

```
String::String() {}
```

b. Что неверно в этом конструкторе?

```
String::String(const char * s)
{
 str = s;
 len = strlen(s);
}
```

c. Что неверно в этом конструкторе?

```
String::String(const char * s)
{
 strcpy(str, s);
 len = strlen(s);
}
```

2. Укажите три проблемы, с которыми можно столкнуться при определении следующего класса. В этом классе элемент указателя инициализируется с помощью оператора `new` и указывает способ восстановления.

3. Какие методы класса заставляют компилятор выполнять генерирование в автоматическом режиме, если этот процесс не поддерживается в явном виде? Опишите, каким образом неявный подход оказывается на поведении функций.

4. Идентифицируйте и скорректируйте ошибки в следующем объявлении класса:

```
class nifty
{
// данные
 char personality[];
 int talents;
// методы
 nifty();
 nifty(char * s);
 ostream & operator<<(ostream & os,
 nifty & n);
}
nifty::nifty()
{
 personality = NULL;
 talents = 0;
}
nifty::nifty(char * s)
{
 personality = new char [strlen(s)];
 personality = s;
 talents = 0;
}
ostream & nifty::operator<<(ostream & os,
 nifty & n)
{
 os << n;
}
```

5. Рассмотрите следующее объявление класса:

```
class Golfer
{
private:
 char * fullname; // отмечает строку,
 // содержащую имя игрока в гольф
 int games; // содержит число сыгранных
 // игр в гольф
 int * scores; // отмечает первый элемент
 // массива очков при игре в гольф
public:
 Golfer();
 Golfer(const char * name, int g= 0);
 // образует пустой динамический массив
 // из g элементов if g > 0
 Golfer(const Golfer & g);
 ~Golfer();
};
```

- a. Какие методы класса должны быть вызваны в каждом из этих операторов?

```
Golfer nancy; // #1
Golfer lulu("Little Lulu"); // #2
Golfer roy("Roy Hobbs", 12); // #3
Golfer * par = new Golfer; // #4
Golfer next = lulu; // #5
Golfer hazzard = "Weed Thwacker"; // #6
*par = nancy; // #7
nancy = "Nancy Putter"; // #8
```

- b. Ясно, что классу для выявления его полезности необходимы и некоторые другие методы. Какой дополнительный метод потребуется для защиты данных от повреждения?

## Упражнения по программированию

1. Рассмотрите следующее объявление класса:

```
class Cow {
 char name[20];
 char * hobby;
 double weight;
public:
 Cow();
 Cow(const char * nm, const char * ho,
 double wt);
 Cow(const Cow & c);
 ~Cow();
 Cow operator=(const Cow & c);
 void ShowCow(); // отображает имя,
 // любимый корм и вес коровы
};
```

Выполните реализацию для этого класса и создайте короткую программу, которая использует все функции-элементы.

2. Усовершенствуйте объявление класса **String** (т.е. обновите **strng2.h** к **strings.h**), выполняя следующие действия:

- Перегрузите операцию **+**, которая позволяет объединять две строки в одну.
- Создайте функцию-элемент **stringlow()**, которая конвертирует все символы алфавита в строке к нижнему регистру. (Не забывайте набор символьных функций **cctype**.)
- Создайте функцию-элемент **stringup()**, которая конвертирует все символы алфавита в строке к верхнему регистру.
- Создайте функцию-элемент, которая получает аргумент **char** и возвращает число появлений символа в строке.

Проверьте результаты своей работы в следующей программе:

```
// pell_1.cpp
#include <iostream>
using namespace std;
#include "strng3.h"
int main()
{
 String s1(" and I am a C++ student.");
 String s2 = "Please enter your name: ";
 String s3;
 cout << s2; // перегруженная операция <<
 cin >> s3; // перегруженная операция >>
 s2 = "My name is " + s3;
 // перегруженные операции =, +
 cout << s2 << ".\n";
 s2 = s2 + s1;
 s2.stringup(); // преобразует символы
 // строки к верхнему регистру
 cout << "The string\n" << s2
 << "\ncontains " << s2.has('A')
 << " 'A' characters in it.\n";
 s1 = "red"; // String(const char *),
 // тогда String &
 // operator=(const String&)
 String rgb[3] = { String(s1),
 String("green"), String("blue") };
 cout << "Enter the name of a primary
 color for mixing light: ";
 String ans;
 bool success = false;
 while (cin >> ans)
 {
 ans.stringlow(); // преобразует
 // символы строки к нижнему регистру
 for (int i = 0; i < 3; i++)
 {
 if (ans == rgb[i]) //перегруженный
 //оператор ==
 {
 cout << "That's right!\n";
 success = true;
 break;
 }
 }
 if (success)
 break;
 else
 cout << "Try again!\n";
 }
 cout << "Bye\n";
 return 0;
}
```

Результаты выполнения программы должны выглядеть подобно следующему:

```
Please enter your name: Fretta Farbo
My name is Fretta Farbo.
The string
MY NAME IS FRETTA FARBO AND I AM A C++ STUDENT.
contains 6 'A' characters in it.
Enter the name of a primary color for
mixing light: yellow
Try again!
```

**BLUE**

That's right!

Bye

3. Перепишите класс **Stock**, введенный в листингах 9.7 и 9.8, так, чтобы для хранения наименований акций использовалась динамически распределяемая память вместо статических массивов. Замените функцию-элемент **show( )** перегруженным определением **operator<<( )**. Протестируйте новую программу из листинга 9.9.

4. Рассмотрите следующий вариант класса **Stack**, который определен в листинге 9.10.

```
// stack.h - объявление класса для стека
// ADT
typedef unsigned long Item;

class Stack
{
private:
 enum {MAX = 10}; // определяемая классом
 // константа
 Item * pitems; // содержит элементы
 // стека
 int size; // количество элементов в
 // стеке
 int top; // индекс для верхнего пункта
 // стека
public:
 Stack(int n = 10); // образует стек
 // с n элементами
 Stack(const Stack & st);
 ~Stack();
 bool isempty() const;
 bool isfull() const;
 // push() возвращает False, если стек
 // уже полон, True - в противном случае
 bool push(const Item & item);
 // добавление элемента в стек
 // pop() возвращает False, если стек уже
 // пуст, True - в противном случае
 bool pop(Item & item); // перемещение
 // верхнего элемента в Item
 Stack operator=(const Stack & st);
};
```

Как предполагается для приватных элементов, этот класс для хранения элементов стека применяет динамически распределенный массив. Перепишите методы таким образом, чтобы они подошли для этого нового представления, и создайте программу, которая демонстрирует все эти методы. Сюда также следует включить конструктор копирования и оператор присваивания.

5. Банк Bank of Heather выполнил исследование, которое показывает, что клиенты, которые обращаются к автоответчику, не желают ожидать на линии более одной минуты. С помощью моделирования из листинга 11.13 найдите значение для количества клиентов, которые обращаются ежесчно при условии, что среднее время ожидания составит одну минуту. (Воспользуйтесь, по крайней мере, испытательным периодом длительностью в 100 часов.)
6. Банк Bank of Heather желает узнать, что же произойдет, если добавить второй автоответчик. Внесите изменения в процесс моделирования таким образом, чтобы существовало две очереди. Потребуйте, чтобы клиент присоединялся к первой очереди, если туда входит меньше людей, чем во вторую, а в противном случае чтобы он присоединялся ко второй очереди. Снова найдите значение для количества клиентов, обращающихся ежесчно, при условии, что среднее время ожидания составит одну минуту. (Примечание: вы столкнетесь с нелинейной задачей. Увеличение числа компьютеров в 2 раза не приводит к прямо пропорциональному увеличению количества клиентов, которые могут обслуживаться ежесчно с максимальным временем ожидания в одну минуту.)

# Наследование классов

**В этой главе рассматривается следующее:**

- Наследование в качестве отношения *is-a*
- Общедоступное наследование классов
- Защищенный доступ
- Списки инициализатора конструктора
- Приведение вверх и приведение вниз
- Виртуальные функции-элементы
- Раннее (статическое) и позднее (динамическое) связывание
- Полностью виртуальные функции
- Работа с методом общедоступного наследования

Одна из основных целей объектно-ориентированного программирования — обеспечение кода многократного использования. При разработке нового проекта, особенно если проект большой, хорошо иметь возможность повторно использовать проверенный код, а не изобретать его снова. Использование старого кода позволяет экономить время и, поскольку он уже использовался и проверен, может помочь избежать появления ошибок в программе. Кроме того, чем меньше приходится вникать в детали, тем больше можно сконцентрироваться на общей стратегии программы.

Традиционные библиотеки функций C обеспечивают возможность многократного использования посредством предопределенных, заранее скомпилированных функций типа `strlen()` и `rand()`, которые можно использовать в пользовательских программах. Многие поставщики поставляют специализированные библиотеки C, содержащие функции, которые выходят за рамки функций стандартной библиотеки C. Например, можно приобрести библиотеки функций управления базы данных и функций управления отображением на экране. Однако библиотеки функций имеют ограничение. Если поставщик не обеспечивает исходный код для библиотечных функций (а часто это именно так), нельзя расширять или изменять функции, чтобы они удовлетворяли конкретным потребностям. Вместо этого приходится видоизменять свою программу, чтобы она соответствовала функциям библиотеки. Даже если

поставщик обеспечивает исходный код, в ходе внесения изменений существует риск неумышленного изменения работы части функции или связей между библиотечными функциями.

Классы C++ обеспечивают более высокий уровень возможностей многократного использования. В настоящее время многие поставщики предлагают библиотеки классов, которые состоят из объявлений и реализаций классов. Поскольку класс объединяет представление данных с методами класса, он предоставляет более комплексный пакет, чем библиотека функций. Например, одиночный класс может обеспечивать все ресурсы, необходимые для управления диалоговым окном. Часто библиотеки классов доступны в виде исходного кода, и, следовательно, их можно изменять для удовлетворения конкретных потребностей. Однако для расширения и изменения классов в C++ существует лучший метод, чем модификация кода. Этот метод, названный *наследованием классов*, позволяет производить новые классы из старых, причем производный класс наследует свойства, включая методы, старого класса, называемого базовым. Так же, как наследовать деньги обычно проще, чем накопить их с нуля, выведение класса путем наследования обычно проще, чем разработка нового класса. Вот некоторые действия, которые можно выполнять с наследованием:

- Добавлять функциональные возможности к существующему классу. Например, имея базовый класс массива, можно добавлять арифметические операции.

- Выполнять добавление элементов к данным, представляющим класс. Например, имея базовый класс строки, можно получить класс, который добавляет элемент данных, представляющий цвет, который должен использоваться при отображении строки.
- Изменять поведение метода класса. Например, располагая классом **Passenger**, описывающим услуги, предоставляемые пассажирам авиалинии, можно получить класс **Upgrade**, который обеспечивает более высокий уровень услуг.

Конечно, этих же целей можно было бы достигнуть, дублируя код исходного класса и изменяя его, но механизм наследования позволяет выполнить задачу, обеспечивая только новые свойства. Для получения производного класса даже не требуется доступ к исходному коду. Таким образом, при приобретении библиотеки классов, которая предоставляет только заголовочные файлы и скомпилированный код для методов класса, все же можно производить новые классы, основанные на библиотечных классах. И наоборот, можно предоставлять свои собственные классы другим пользователям, храня части своей реализации в секрете и все же предоставляя клиентам возможность добавлять свойства к вашим классам.

Наследование — замечательная концепция, а ее базовая реализация достаточно проста. Однако управление наследованием, чтобы оно выполнялось правильно во всех ситуациях, требует некоторых корректировок. В этой главе рассмотрены как простые, так и более тонкие аспекты наследования.

## Простой базовый класс

Когда один класс наследуется из другого, исходный класс называется *базовым*, а наследующий — *производным*. Таким образом, чтобы проиллюстрировать наследование, следует начать с базового класса. Весьма удачно оказалось, что Понтонский национальный банк нуждается в классе, представляющем его основной расчетный счет, так называемый "Медный счет". В листинге 12.1 показан заголовок для упрощенного класса **BankAccount**, удовлетворяющего этой потребности. Он включает элементы данных, представляющие имя клиента, номер счета и баланс. Класс содержит методы для создания счета, внесения вкладов, снятия со счетов и отображения данных счета. Класс не в состоянии представлять реальный банковский счет, но он имеет достаточно свойств, чтобы удовлетворить вымышленный Понтонский национальный банк и чтобы с него можно было начать изучение наследования.

### Листинг 12.1 Программа bankacct.h

```
// bankacct.h - простой класс BankAccount.
#ifndef _BANKACCT_H_
#define _BANKACCT_H_

class BankAccount
{
private:
 enum {MAX = 35};
 char fullName[MAX];
 long acctNum;
 double balance;
public:
 BankAccount(const char *s = "Nullbody",
 long an = -1, double bal = 0.0);
 void Deposit(double amt);
 void Withdraw(double amt);
 double Balance() const;
 void ViewAcct() const;
};

#endif
```

Класс использует методику создания константы области класса путем использования функции **enum**.

Далее следуют методы класса, как показано в листинге 12.2.

### Листинг 12.2 Программа bankacct.cpp.

```
// bankacct.cpp - методы для класса
// BankAccount.
#include <iostream>
using namespace std;
#include "bankacct.h"
#include <cstring>

BankAccount::BankAccount(const char *s,
 long an, double bal)
{
 strncpy(fullName, s, MAX - 1);
 fullName[MAX - 1] = '\0';
 acctNum = an;
 balance = bal;
}

void BankAccount::Deposit(double amt)
{
 balance += amt;
}

void BankAccount::Withdraw(double amt)
{
 if (amt <= balance)
 balance -= amt;
 else
 cout << "Withdrawal amount of $" << amt
 << " exceeds your balance.\n"
 << "Withdrawal canceled.\n";
}

double BankAccount::Balance() const
{
 return balance;
}
```

```

void BankAccount::ViewAcct() const
{
 // установка формата ####.##
 ios_base::fmtflags initialState =
 cout.setf(ios_base::fixed,
 ios_base::floatfield);
 cout.setf(ios_base::showpoint);
 cout.precision(2);
 cout << "Client: " << fullName << endl;
 cout << "Account Number: " << acctNum
 << endl;
 cout << "Balance: $" << balance << endl;
 cout.setf(initialState); // восстановление
 // первоначального формата
}

```



### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Используйте `string.h`, если транслятор не поддерживает `cstring`. Если транслятор не поддерживает `ios_base`, используйте вместо этого `ios`. В этом случае в качестве типа для `initialState` следует использовать `long` вместо `ios_base::fmtflags`.

Существует несколько обстоятельств, связанных с реализацией, на которые следует обратить внимание. Во-первых, метод `Withdraw()` проверяет, что на счету имеется достаточно средств, чтобы покрыть снятие денег со счета. Если это не так, снятие денег не разрешается. Это пример интерфейса, предоставляющего защиту, которая отсутствовала бы, если бы разрешался прямой доступ к элементам данных.

Во-вторых, метод `ViewAcct()` использует команды форматирования для отображения денежных значений в формате \$2356.32. В ранее приведенных примерах использовались аналогичные параметры настройки (которые подробнее освещены в главе 16), но этот метод идет дальше, сохраняя информацию о первоначальном форматировании:

```

ios_base::fmtflags initialState =
 cout.setf(ios_base::fixed,
 ios_base::floatfield);

```

Здесь `ios_base::fmtflags` — тип, определенный в классе `ios_base`. (Библиотеки, которые используют более старый класс `ios`, применяют вместо него тип `long`.) При этом вызове метода `setf()` устанавливается формат с фиксированной десятичной точкой для вывода и возвращается настройка флага формата, которая существовала до выполнения вызова. Это позволяет функции восстанавливать первоначальные параметры настройки по завершении ее работы:

```

cout.setf(initialState); // восстановление
 // первоначального формата

```

Далее в листинге 12.3 представлена небольшая программа, иллюстрирующая короткий список свойств класса.

### Листинг 12.3 Программа usebank.cpp.

```

// usebank.cpp
// компилируется с bankacct.cpp
#include <iostream>
using namespace std;
#include <cstring>
#include "bankacct.h"

int main()
{
 BankAccount Porky("Porcelot Pigg",
 381299, 4000.00);

 Porky.ViewAcct();
 Porky.Deposit(5000.00);
 cout << "New balance: $"
 << Porky.Balance() << endl;
 Porky.Withdraw(8000.00);
 cout << "New balance: $"
 << Porky.Balance() << endl;
 Porky.Withdraw(1200.00);
 cout << "New balance: $"
 << Porky.Balance() << endl;

 return 0;
}

```



### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Используйте `string.h`, если компилятор не поддерживает `cstring`.

Ниже приведен вывод программы:

```

Client: Porcelot Pigg
Account Number: 381299
Balance: $4000.00
New balance: $9000.00
New balance: $1000.00
Withdrawal amount of $1200.00 exceeds your
balance.
Withdrawal canceled.
New balance: $1000.00

```

## Наследование — отношение *is-a*

Теперь, когда `BankAccount` получил ранг рабочего класса, из него можно получить новый класс. Однако перед этим давайте рассмотрим модель, лежащую в основе наследования C++. Фактически C++ имеет три разновидности наследования: общедоступную, защищенную и приватную. Общедоступное наследование — наиболее общая форма, и она моделируется отношением *is-a* ("является объектом"). Это сокращенная форма выражения того, что объект производного класса должен также быть объектом базового класса. Необходимо, чтобы манипу-

лировать объектом производного класса можно было точно так же, как объектом базового класса. Предположим, например, что имеется класс **Fruit**. В нем можно было бы задавать, скажем, вес и энергетическую ценность плода. Поскольку банан — конкретный вид плода, класс **Banana** можно было бы получить из класса **Fruit**. Новый класс наследовал бы все элементы данных исходного класса, так что объект **Banana** содержал бы элементы, представляющие вес и энергетическую ценность банана. Новый класс **Banana** мог бы также добавлять элементы, которые присущи именно бананам, а не плодам вообще, такие как индекс шкурки, введенный Институтом бананов. Поскольку производный класс может добавлять свойства, вероятно, более точным описанием отношения был бы термин *is-a-kind-of* ("является объектом типа"), но *is-a* — общепринятый термин.

Чтобы выяснить, что собой представляет отношение *is-a*, давайте рассмотрим несколько примеров, которые не соответствуют этой модели. Общедоступное наследование не моделирует отношение *has-a* ("имеет объект"). Например, завтрак мог бы содержать плод. Но завтрак вообще — это не плод. Следовательно, предпринимая попытку поместить плод в завтрак, не следует получать класс **Lunch** из класса **Fruit**. Правильный способ обработки помещения плода в завтрак состоит в рассмотрении задачи в качестве отношения *has-a*: завтрак содержит плод. Как будет показано в главе 13, это легче всего смоделировать путем включения объекта **Fruit** в качестве элемента данных класса **Lunch** (рис. 12.1).

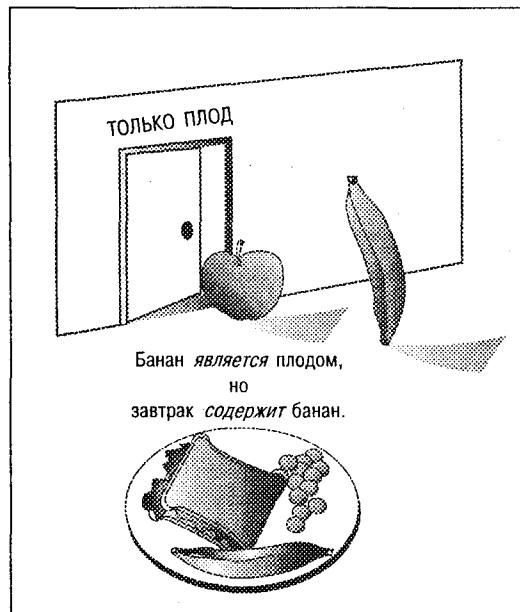


РИСУНОК 12.1 Отношения *is-a* и *has-a*.

Общедоступное наследование не моделирует отношение *is-like-a* ("подобный объекту"), т.е. оно не создает подобия. Часто говорят, что адвокаты подобны акулам. Но это не означает буквально, что адвокат — акула. Например, акулы могут жить под водой. Следовательно, вам не требуется получать класс **Lawyer** из класса **Shark**. Путем наследования можно добавлять свойства к базовому классу; но не удалять свойства из него. В некоторых случаях общие характеристики могут обрабатываться путем разработки класса, определяющего эти характеристики, и последующего использования этого класса в отношении *is-a* или *has-a* для определения связанных классов.

Общедоступное наследование не моделирует отношение *is-implemented-as-a* ("реализован в качестве объекта"). Например, стек можно было бы реализовать путем использования массива. Однако было бы неверным получать класс **Stack** из класса **Array**. Стек — это не массив. Например, индексация массива не является свойством стека. Кроме того, стек мог бы быть реализован каким-либо другим способом, например, путем использования связанного списка. Правильным подходом было бы скрытие реализации массива, присвоение стеку элемента приватного объекта **Array**.

Общедоступное наследование не моделирует отношение *uses-a* ("использует объект"). Например, компьютер может использовать лазерный принтер, но нет смысла получать класс **Printer** из класса **Computer** или наоборот. Однако можно было бы создать удобные функции или классы для выполнения обмена данными между объектами **Printer** и **Computer**.

Ничто в языке C++ не препятствует использованию метода общедоступного наследования для моделирования отношений *has-a*, *is-implemented-as-a* или *uses-a*. Но обычно это приводит к проблемам программирования. Поэтому давайте придерживаться отношений *is-a*. Снова весьма кстати оказалось, что Понтонский национальный банк также предоставляет текущий счет "Медный Плюс". Этот счет имеет все свойства регулярного "Медного счета", а также обеспечивает блокировку от превышения кредита. Так, если вы выписываете чек на сумму, превышающую (но не слишком) имеющуюся на текущем счету, банк покроет чек, выставив вам счет на оплату излишка и наложив определенный штраф. Желательно было бы использовать класс **BankAccount**, но он не поддерживает новое свойство. В этом случае можно определить новый класс, который наследует все свойства класса **BankAccount** и дополнительно имеет необходимые новые функциональные возможности. Создавая новый класс на основе класса **BankAccount**, вместо того чтобы начинать с нуля, можно воспользоваться результатами работы, которая уже была предде-

лана при разработке класса **BankAccount**, и тем, что класс **BankAccount** уже проверен. Другими словами, в данном случае повторно используется проверенный код. В результате приходится выполнять меньший объем работы и, вероятно, при этом качество конечной программы оказывается выше.

Удовлетворяет ли новый класс (назовем его **Overdraft**) условиям отношения *is-a*? Безусловно. Все, что является истинным для объекта **BankAccount**, будет истинным и для объекта **Overdraft**. Это значит, что можно вносить вклады, снимать деньги со счета и отображать информацию о счете. Обратите внимание, что в общем случае отношение *is-a* не является обратимым. В общем случае плод не всегда будет бананом. Объект **BankAccount** не будет иметь всех возможностей объекта **Overdraft**.

## Объявление производного класса

Производный класс должен идентифицировать класс, из которого он произведен. Метод реализации этого в C++ заключается во включении имени базового класса в объявление производного класса. Если класс **Overdraft** производится из класса **BankAccount**, объявление класса начиналось бы подобно следующему:

```
class Overdraft : public BankAccount
{
```

Двоеточие указывает, что класс **Overdraft** основан на классе **BankAccount**. Этот конкретный заголовок указывает, что **BankAccount** — общедоступный базовый класс; такой процесс называется *общедоступным произведением*. Объект производного класса включает в себя объект базового класса. При осуществлении общедоступного произведения общедоступные элементы базового класса станут общедоступными элементами производного класса. Приватные разделы базового класса станут частью производного класса, но к ним можно обращаться только посредством общедоступных и защищенных методов базового класса. (Вскоре мы рассмотрим защищенные элементы.)

Например, функция **Deposit()** становится также общедоступной функцией класса **Overdraft**. Элемент **balance** объекта **BankAccount** становится частью объекта **Overdraft**, но к нему можно обращаться только посредством таких методов **BankAccount**, как **Deposit()** и конструкторы **BankAccount**. Одним словом, класс **Overdraft** наследует общедоступные элементы из базового класса наряду с доступом к ним. Их не нужно переопределять для нового класса. Производный класс содержит приватные элементы базового класса, но не может обращаться к ним иначе, кроме как используя общедоступные и защищенные методы базового класса (рис. 12.2).

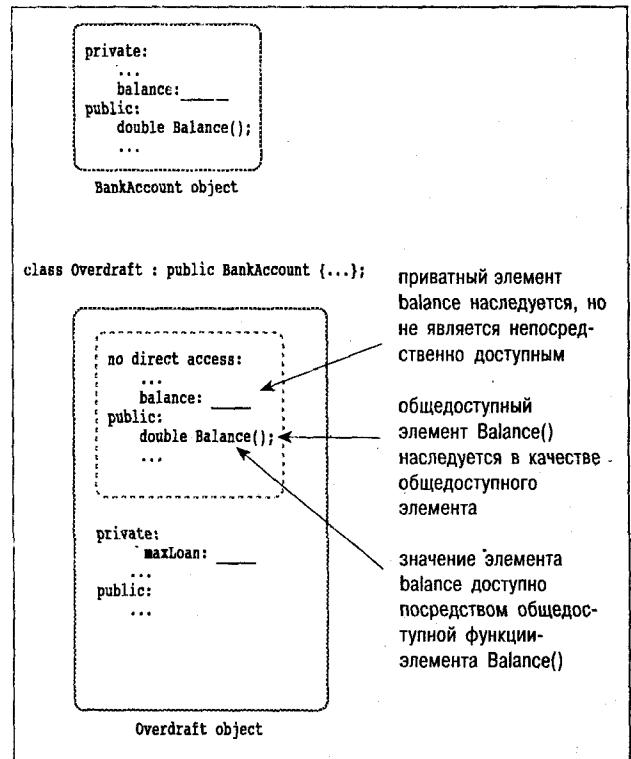


РИСУНОК 12.2 Объекты базового и производного классов.

C++ также поддерживает защищенные и частные произведения:

```
class Computer : protected HardDisk
{...} ;
class House : private Study
{...};
```

Эти формы будут рассмотрены в главе 13. А пока главное — отметить, что, если опустить ключевое слово доступа, C++ сделает произведение приватным:

```
class House : Study // то же, что и
 // private Study
{...};
```

После получения производного класса к нему можно добавлять новые элементы. Фактически необходимо обеспечить новые конструкторы. Дело в том, что имя конструктора совпадает с именем класса:

```
BankAccount bogey; // требуется
// конструктор BankAccount()
Overdraft orson; // требуется
// конструктор Overdraft()
```

При создании объекта производного класса вначале программа вызывает конструктор для базового класса, а затем конструктор для производного класса. В этом есть смысл, поскольку конструктор для производного класса может строить поверх элементов данных из базового

класса; следовательно, объект базового класса должен быть создан первым. Это несколько напоминает строительство первого этажа здания перед монтажом второго этажа. Таким образом, при определении новых конструкторов они не должны дублировать работу базовых конструкторов. Вместо этого необходимо обрабатывать только дополнительные детали, которые требуются производному классу. Например, конструктор мог бы инициализировать новые элементы данных. В общем случае конструктор производного класса должен также передавать информацию конструктору базового класса; вскоре мы рассмотрим методику для выполнения этого.

Новый деструктор не нужно добавлять явно, если только новый класс не требует очистки, кроме выполняемой базовым деструктором. Когда срок существования объекта истекает, программа вначале вызывает деструктор для производного класса, если таковой существует, а затем базовый деструктор.

### ПОМНИТЕ

При создании объекта производного класса вначале программа вызывает конструктор базового класса, а затем конструктор производного класса. Когда срок существования объекта производного класса истекает, программа вначале вызывает деструктор производного класса, а затем деструктор базового класса.

В общем случае производный класс наследует функции-элементы базового класса. Если функции-элементы базового класса являются общедоступными или защищенными, объекты производного класса могут их вызывать. Конструкторы и деструкторы являются исключениями, но конструкторы и деструкторы производного класса могут использовать конструкторы и деструкторы базового класса, как будет показано в примерах. Еще одна ненаследуемая функция-элемент — оператор присвоения. Он заслуживает специального рассмотрения, которое будет дано в свое время. Обратите внимание, что дружественные конструкции не являются функциями-элементами, поэтому они не наследуются.

Читатели уже знают, как начинается объявление производного класса:

```
class Overdraft : public BankAccount
{
```

Однако перед завершением объявления нужно точно знать, какие свойства добавляет счет "Медный Плюс". В результате обсуждения с дружественным представителем Понтонского национального банка были определены следующие свойства:

- Счет "Медный Плюс" ограничивает денежные суммы, предоставляемые банком для покрытия превышения кредита. Значение, принятое по умолчанию, — \$500,

но некоторые клиенты могут начинать с другого ограничения.

- Банк может изменять ограничение на превышение кредита клиента.
- Счет "Медный Плюс" выставляет счет оплаты процента по ссуде. Значение, принятое по умолчанию, — 10%, но некоторые клиенты могут начинать с другого тарифа.
- Банк может изменять величину процентной ставки.
- Счет отслеживает задолженность клиента банку (ссуды на превышение кредита плюс проценты за предоставление ссуды). Пользователь не может оплачивать эту сумму путем обычного вклада на счет или переводом с другого счета. Вместо этого он должен оплатить сумму наличными специальному чиновнику банка, который в случае необходимости разыщет клиента. Как только долг оплачен, счет может сбрасывать задолженность до 0.

Последнее свойство — необычный способ ведения банковских дел, но оно обладает удачным побочным эффектом, упрощая проблему программирования.

Этот список предполагает, что новый класс нуждается в элементах данных для хранения максимального значения задолженности, тарифом процентной ставки и текущей задолженности. Новому классу требуются конструкторы, которые обеспечивают информацию о счете и которые вносят предел задолженности со значением, равным по умолчанию \$500 и процентную ставку со значением, равным по умолчанию 10%. Кроме того, должны существовать методы для переопределения предела задолженности, величины процентной ставки и текущей задолженности. Все эти объекты необходимо добавить к классу **BankAccount**, и они будут объявлены в объявлении класса **Overdraft**.

В то же время имеются методы **BankAccount**, поведение которых должно быть изменено. В частности, методы **Withdraw()** и **ViewAcct()** должны выполнять больше функций, чем прежде. Общедоступное наследование позволяет производному классу переопределять методы базового класса, поэтому объявление класса **Overdraft** должно будет переопределить методы **Withdraw()** и **ViewAcct()**.

Метод **Deposit()** работает одинаковым образом для обоих классов. Производный класс использует метод базового класса, если только это не переопределено. Поэтому, чтобы сохранить использование метода **Deposit()** базового класса, в объявлении класса **Overdraft** не выполняются никакие действия. То же самое справедливо и по отношению к методу **Balance()**.

В листинге 12.4 показано объявление класса, соответствующее этим условиям. Обратите внимание, что с

базовым классом не выполняются никакие действия, чтобы получить производный класс. Определение производного класса происходит при определении нового класса и его методов. Таким образом, производный класс можно получить даже при отсутствии доступа к исходному коду для базового класса.

#### Листинг 12.4 Объявление класса overdraft.h.

```
// overdraft.h --объявление класса Overdraft
#ifndef _OVERDRFT_H_
#define _OVERDRFT_H_
#include "bankacct.h"

class Overdraft : public BankAccount
{
private:
 double maxLoan;
 double rate;
 double owesBank;
public:
 Overdraft(const char *s = "Nullbody",
 long an = -1, double bal = 0.0,
 double ml = 500, double r = 0.10);
 Overdraft(const BankAccount & ba,
 double ml = 500, double r = 0.1);
 void ViewAcct() const;
 void Withdraw(double amt);
 void ResetMax(double m) { maxLoan = m; }
 void ResetRate(double r) { rate = r; }
 void ResetOwes() { owesBank = 0; }
};

#endif
```

## Реализация производного класса

Давайте исследуем, как реализовать производный класс, и рассмотрим объяснение некоторых из методов, начиная с конструкторов. Вначале давайте подумаем о процессе конструирования. Программа не может создать объект **Overdraft** до тех пор, пока вначале не создаст объект **BankAccount**. Поэтому базовый конструктор должен быть вызван прежде, чем программа введет код для производного конструктора. С другой стороны, базовый конструктор не может быть вызван до тех пор, пока не вызван производный конструктор, поскольку именно при обращении к производному конструктору программе сообщается о потребности в базовом конструкторе. Давайте рассмотрим следующий конструктор **Overdraft**:

```
Overdraft(const char *s = "Nullbody", long
 an = -1, double bal = 0.0,
 double ml = 500, double r = 0.10);
```

При этом имеется пять аргументов, три из которых обеспечивают значения для раздела **BankAccount**, а два — значения для раздела **Overdraft**. Последние два аргумента использовать достаточно просто:

```
// незавершенная версия
Overdraft::Overdraft(const char *s, long an,
 double bal, double ml, double r)
{
 maxLoan = ml;
 owesBank = 0.0; // начало без
 // задолженности
 rate = r;
}
```

А как дела обстоят с компонентом **BankAccount**? Вначале давайте рассмотрим, что произошло бы в случае использования этой незавершенной версии конструктора. Объект базового класса создается прежде, чем добавляется производный компонент. С точки зрения синтаксиса данного конструктора это означает, что объект базового класса создается прежде, чем выполнение программы передается оператору в теле конструктора. Поскольку никакой конструктор не упоминается явно, это означает, что конструктор **BankAccount** по умолчанию используется для создания компонента базового класса. Давайте перефразируем это: если не указано иное, конструктор производного класса вызывает заданный по умолчанию базовый конструктор перед вхождением в тело конструктора производного класса.

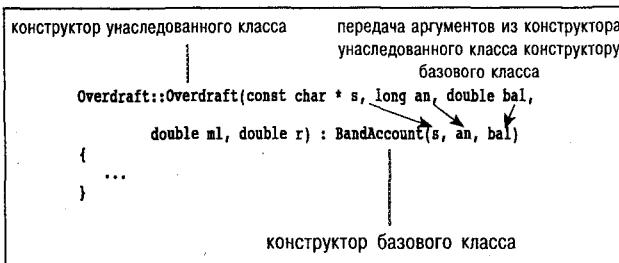
Однако в данном случае заданный по умолчанию конструктор некорректен, поскольку он использует значения, принятые по умолчанию, вместо требуемых значений. C++ предлагает специальный синтаксис, который позволяет определять, какой конструктор должен использоваться. Это разновидность синтаксиса списка инициализатора, который был приведен в главе 11, за исключением того, что вместо имени элемента используется имя класса:

```
Overdraft::Overdraft(const char *s, long an,
 double bal, double ml, double r)
: BankAccount(s, an, bal)
{
 maxLoan = ml;
 owesBank = 0.0;
 rate = r;
}
```

Здесь часть

```
: BankAccount(s, an, bal)
```

означает: "Вызовите конструктор **BankAccount(const char \*, double, double)** для создания части базового класса объекта **Overdraft**". Путем использования этого механизма значения первых трех аргументов конструктора **Overdraft** передаются конструктору **BankAccount**. Таким образом, конструктор **BankAccount** устанавливает унаследованные элементы, а тело конструктора **Overdraft** устанавливает новые элементы (рис. 12.3).



**РИСУНОК 12.3** Передача аргументов конструктору базового класса.

Итак, конструктор производного класса всегда вызывает конструктор базового класса перед выполнением операторов в теле конструктора производного класса. Программа использует заданный по умолчанию базовый конструктор, если только другой конструктор не указан явно путем использования синтаксиса списка инициализатора. В этом случае аргументы из конструктора производного класса можно использовать в качестве аргументов конструктора базового класса.

### ПОМНИТЕ

Конструкторы производного класса ответственны за инициализацию любых элементов данных, добавленных к унаследованным из базового класса. Конструкторы базового класса ответственны за инициализацию унаследованных элементов данных. Для указания подлежащего использованию конструктора базового класса можно использовать синтаксис списка инициализатора. В противном случае используется заданный по умолчанию конструктор базового класса. .

### СПИСКИ ИНИЦИАЛИЗАТОРА

Конструктор для производного класса может использовать механизм списка инициализатора для передачи значений конструктору базового класса.

```
derived::derived(type1 x, type2 y) : base(x,y)
// список инициализатора
{
 ...
}
```

Здесь **derived** – производный класс, **base** – базовый класс, а **x** и **y** – переменные, используемые конструктором базового класса. Если, скажем, производный конструктор получает аргументы 10 и 12, то этот механизм затем передает 10 и 12 базовому конструктору, определенному в качестве принимающего аргументы этих типов. За исключением случая виртуальных базовых классов (см. главу 14), класс может передавать значения обратно только непосредственному базовому классу. Однако этот класс может использовать тот же самый механизм для передачи возвращаемой информации своему непосредственному базовому классу и т.д. Если конструктор базового класса отсутствует в списке инициализатора, программа будет использовать заданный по умолчанию конструктор базового класса. Список инициализатора может использоваться только с конструкторами.

В главе 11 вы столкнулись с синтаксисом для инициализации конкретных составных элементов класса. Например, конструктор

```
Queue::Queue(int qs) : qsize(qs)
// initialize qsize to qs
{
 front = rear = NULL;
 items = 0;
}
```

инициализирует элемент **qsize** объекта **Queue** значением **qs**. Синтаксис, который был здесь использован, – вариант этой более ранней формы. Различие состоит в том, что имя элемента класса используется для инициализации конкретного элемента объекта, но имя базового класса используется для инициализации компонента базового класса объекта.

### Инициализация объектов объектами

Давайте рассмотрим второй конструктор – **Overdraft**:

```
Overdraft(const BankAccount & ba, double ml
= 500, double r = 0.1);
```

Он предназначен для того, чтобы позволить преобразование от "Медного счета" в счет "Медный Плюс". Здесь аргумент **ba** обеспечивает информацию старого счета, а остальные аргументы обеспечивают информацию для новых элементов данных. Вопрос состоит в том, как использовать аргумент **BankAccount** для инициализации раздела **BankAccount**. Поскольку это действие создает копию объекта **BankAccount**, необходимо использовать конструктор копии:

```
Overdraft::Overdraft(const BankAccount & ba,
double ml, double r) : BankAccount(ba)
{
 maxLoan = ml;
 owesBank = 0.0;
 rate = r;
}
```

Действительно, объявление **BankAccount** не определяет конструктор копии явно. Однако вспомните, что компилятор обеспечивает заданный по умолчанию конструктор копии, если конструктор копии необходим и ни один не определен. Он выполняет копирование с учетом элементов, которые прекрасно подходит для объекта **BankAccount**.

### Другие функции-элементы

Класс **Overdraft** не определяет функцию **Deposit()**, и, следовательно, объект **Overdraft** будет использовать **BankAccount::Deposit()**. Такое же поведение сохраняется для функции **Balance()**. Но новый класс определяет метод **ViewAcct()**, значит, объект **Overdraft** будет использовать **Overdraft::ViewAcct()**. Давайте посмотрим, как это реализовать. Во-первых, кое-что не будет работать:

```

void Overdraft::ViewAcct() const // НЕВЕРНАЯ
 // ВЕРСИЯ
{
 // установка формата ###.##
 ios_base::fmtflags initialState =
 cout.setf(ios_base::fixed,
 ios_base::floatfield);
 cout.setf(ios_base::showpoint);
 cout.precision(2);
 cout << "Client: " << fullName << endl;
 // неверно
 cout << "Account Number: " << acctNum
 << endl; // неверно
 cout << "Balance: $" << balance
 << endl; // неверно
 cout << "Maximum loan: $" << maxLoan
 << endl;
 cout << "Owed to bank: $" << owesBank
 << endl;
 cout.setf(initialState);
}

```

Проблема такова, что должно стать очевидным следующее: производный класс не может непосредственно обращаться к приватным данным и методам базового класса. Так, объект **Overdraft** содержит объект **BankAccount** с элементами **fullName**, **acctNum** и **balance**, но он не может обращаться к ним по имени. Дело в том, что общедоступный раздел базового класса определяет интерфейс для того класса, а остальная часть программы, включая производные классы, должна использовать этот интерфейс. В данном случае класс **Overdraft** может использовать общедоступный интерфейс класса **BankAccount**, чтобы получить доступ к данным **BankAccount**. Например, метод **Overdraft** может использовать метод **BankAccount::ViewAcct()**:

```

void Overdraft::ViewAcct() const
{
 // установка формата ###.##
 ios_base::fmtflags initialState =
 cout.setf(ios_base::fixed,
 ios_base::floatfield);
 cout.setf(ios_base::showpoint);
 cout.precision(2);
 BankAccount::ViewAcct(); // отображение
 // базовой части
 cout << "Maximum loan: $" << maxLoan
 << endl;
 cout << "Owed to bank: $" << owesBank
 << endl;
 cout.setf(initialState);
}

```

Здесь обязательно нужно было использовать оператор определения диапазона. Если его пропустить, метод **Overdraft::ViewAcct()** будет выполнять рекурсивный вызов самого себя:

```

void Overdraft::ViewAcct() const
{
 ...
}

```

```

ViewAcct(); // НЕТ! рекурсивное
 // обращение к Overdraft::ViewAcct()
BankAccount::ViewAcct(); // вызов
 // версии базового класса
...
}

```

Давайте подытожим, когда какие методы используются. Если производный класс не переопределяет метод базового класса, объект производного класса использует метод базового класса. Если производный класс переопределяет метод, объекты производного класса используют новое определение. Мы говорим, что определение производного класса *отменяет* определение базового класса:

```

BankAccount bretta;
Overdraft ophelia;
bretta.Deposit(20); // использование
 // BankAccount::Deposit()
ophelia.Deposit(40); // использование
 // BankAccount::Deposit()
bretta.ViewAcct(); // использование
 // BankAccount::ViewAcct()
ophelia.ViewAcct(); // использование
 // Overdraft::ViewAcct()

```

Имеется еще одна функция, требующая переопределения. Новая версия **Withdraw()** должна обрабатывать защиту от превышения кредита. Помните, что она может использовать версию базового класса метода **Withdraw()**, но что она не может обращаться к элементу **balance** непосредственно. Это предполагает следующую структуру:

```

void Overdraft::Withdraw(double amt)
{
 double bal = Balance();
 if (amt <= bal)
 BankAccount::Withdraw(amt);
 else if (amt <= bal + maxLoan -
 owesBank)
 {
 double advance = amt - bal;
 owesBank += advance * (1.0 +
 rate);
 cout << "Bank advance: $"
 << advance << endl;
 cout << "Finance charge: $"
 << advance * rate << endl;
 Deposit(advance);
 BankAccount::Withdraw(amt);
 }
 else
 cout << "Credit limit exceeded."
 << Transaction cancelled.\n";
}

```

Если баланс покрывает объем денег, снятых со счета, используйте версию метода базового класса **Withdraw()**. Если защита от превышения кредита необходима, а его величина достаточна мало, чтобы быть обработанным счетом "Медный Плюс", увеличьте счет

клиента на необходимую сумму, выставьте клиенту счет за покрытие разницы плюс процент за предоставление кредита, а затем выполните операцию снятия со счета. В листинге 12.5 приведена полная реализация класса *Overdraft*.

#### Листинг 12.5 Программа *overdrft.cpp*.

```
// overdrft.cpp--методы класса Overdraft
#include <iostream>
using namespace std;
#include "overdrft.h"

Overdraft::Overdraft(const char *s, long an, double bal,
 double ml, double r) : BankAccount(s, an, bal)
{
 maxLoan = ml;
 owesBank = 0.0;
 rate = r;
}

Overdraft::Overdraft(const BankAccount & ba, double ml, double r)
 : BankAccount(ba) // использует заданный по умолчанию конструктор копии
{
 maxLoan = ml;
 owesBank = 0.0;
 rate = r;
}

// переопределение работы ViewAcct()
void Overdraft::ViewAcct() const
{
 // установка формата ###.##
 ios_base::fmtflags initialState = cout.setf(ios_base::fixed, ios_base::floatfield);
 cout.setf(ios_base::showpoint);
 cout.precision(2);

 BankAccount::ViewAcct(); // отображение базовой части
 cout << "Maximum loan: $" << maxLoan << endl;
 cout << "Owed to bank: $" << owesBank << endl;
 cout.setf(initialState);
}

// переопределение работы Withdraw()
void Overdraft::Withdraw(double amt)
{
 // установка формата ###.##
 ios_base::fmtflags initialState = cout.setf(ios_base::fixed, ios_base::floatfield);
 cout.setf(ios_base::showpoint);
 cout.precision(2);

 double bal = Balance();
 if (amt <= bal)
 BankAccount::Withdraw(amt);
 else if (amt <= bal + maxLoan - owesBank)
 {
 double advance = amt - bal;
 owesBank += advance * (1.0 + rate);
 cout << "Bank advance: $" << advance << endl;
 cout << "Finance charge: $" << advance * rate << endl;
 Deposit(advance);
 BankAccount::Withdraw(amt);
 }
 else
 cout << "Credit limit exceeded. Transaction cancelled.\n";
 cout.setf(initialState);
}
```

Следующий шаг заключается в проверке производного класса. Короткая программа, приведенная в листинге 12.6, выполняет эту операцию. Она должна быть скомпилирована с файлами *overdrft.cpp* и *bankacct.cpp*, потому что производный класс использует определения базового класса.

**Листинг 12.6 Программа useover.cpp.**

```

// useover.cpp -- проверяет класс Overdraft
// компилируется с файлами bankacct.cpp и
// overdrft.cpp
#include <iostream>
using namespace std;
#include "overdrft.h"

int main()
{
 BankAccount Porky("Porcelot Pigg",
 381299, 4000.00);
 // преобразование Porcelot в новый тип счета
 Overdraft Porky2(Porky);
 Porky2.ViewAcct();
 cout << "Depositing $5000:\n";
 Porky2.Deposit(5000.00);
 cout << "New balance: $"
 << Porky2.Balance() << "\n\n";
 cout << "Withdrawing $8000:\n";
 Porky2.Withdraw(8000.00);
 cout << "New balance: $"
 << Porky2.Balance() << "\n\n";
 cout << "Withdrawing $1200:\n";
 Porky2.Withdraw(1200.00);
 Porky2.ViewAcct();
 cout << "\nWithdrawing $500:\n";
 Porky2.Withdraw(500.00);
 Porky2.ViewAcct();

 return 0;
}

```

Результаты выполнения программы:

```

Client: Porcelot Pigg
Account Number: 381299
Balance: $4000.00
Maximum loan: $500.00
Owed to bank: $0.00
Depositing $5000:
New balance: $9000.00

Withdrawing $8000:
New balance: $1000.00

Withdrawing $1200:
Bank advance: $200.00
Finance charge: $20.00
Client: Porcelot Pigg
Account Number: 381299
Balance: $0.00
Maximum loan: $500.00
Owed to bank: $220.00

Withdrawing $500:
Credit limit exceeded. Transaction
cancelled.
Client: Porcelot Pigg
Account Number: 381299
Balance: $0.00
Maximum loan: $500.00
Owed to bank: $220.00

```

**Примечания к программе**

Давайте более пристально исследуем части программы. Вначале оператор

**Overdraft Porky2(Porky);**

создает объект **Overdraft**, инициализируя базовый компонент **BankAccount** значениями, сохраненными в объекте **Porky**. Значения, принятые по умолчанию (\$500 и 10%), используются для инициализации новых элементов **maxLoan** и **rate**.

Все операторы

**Porky2.ViewAcct();**

вызывают метод **Overdraft::ViewAcct()**, который, в свою очередь, явно вызывает метод **BankAccount::ViewAcct()** для отображения элементов базового класса.

Оператор

**Porky2.Deposit(5000.00);**

вызывает **BankAccount::Deposit()**, чтобы добавить \$5000 к счету.

Оператор

**cout << "Withdrawing \$1200:\n";**

вызывает метод **Overdraft::Withdraw()**, который добавляет к счету пользователя \$200, вносит в дебет пользователя \$220, а затем метод **BankAccount::Withdraw()** выполняет операцию действительного снятия денег со счета.

**Управление доступом — protected**

До сих пор в примерах классов для управления доступом к элементам класса использовались ключевые слова **public** и **private**. Существует еще одна категория доступа, обозначаемая ключевым словом **protected**. Ключевое слово **protected** подобно ключевому слову **private** тем, что внешний мир может получать доступ к элементам класса в защищенном разделе только путем использования элементов класса. Различие между **private** и **protected** выражается на сцену только внутри классов, производных от базового класса. Элементы производного класса могут обращаться к защищенным элементам базового класса непосредственно, но они не могут непосредственно обращаться к приватным элементам базового класса. Таким образом, элементы в защищенной категории ведут себя подобно приватным элементам до тех пор, пока дело касается внешнего мира, но действуют подобно общедоступным элементам, когда речь идет о производных классах.

Например, предположим, что класс **BankAccount** объявил элемент **balance** в качестве защищенного (**protected**):

```
class BankAccount
{
protected:
 double balance;
};
```

В этом случае класс **Overdraft** мог бы обращаться к **balance** непосредственно, не используя методы класса **BankAccount**. Например, ядро метода **Overdraft::Withdraw()** могло бы быть записано следующим образом:

```
void Overdraft::Withdraw(double amt)
{
 if (amt <= balance) // обращается к
 // balance непосредственно
 balance -= amt;
 else if (amt <= balance + maxLoan -
 owesBank)
 {
 double advance = amt - balance;
 owesBank += advance * (1.0 +
 rate);
 cout << "Bank advance: $"
 << advance << endl;
 cout << "Finance charge: $"
 << advance * rate << endl;
 Deposit(advance);
 balance -= amt;
 }
 else
 cout << "Credit limit exceeded.
Transaction cancelled.\n";
}
```

Использование защищенных элементов данных может упростить написание кода, но это имеет свои недостатки. Например (продолжая рассмотрение примера **Overdraft**), если бы элементы **balance** были защищены, можно было бы записать код, подобный следующему:

```
void Overdraft::Reset(double amt)
{
 balance = amt;
}
```

Класс **BankAccount** был разработан так, чтобы интерфейс **Deposit()** и **Withdraw()** обеспечивал средства только для изменения элемента **balance**. Но метод **Reset()** по существу делает **balance** общедоступной переменной, пока затрагиваются объекты **Overdraft**, игнорируя, например, меры предосторожности, которые использует метод **Withdraw()**.

### ПРЕДОСТЕРЖЕНИЕ

Для управления защищенным доступом к элементам данных класса следует отдавать предпочтение приватному доступу, а для обеспечения производным классам доступа к данным базового класса используйте методы базового класса.

Однако управление защищенным доступом может быть достаточно полезным для функций-элементов. При

этом производным классам предоставляется доступ к внутренним функциям, которые не являются общедоступными.

### Отношение *is-a*, ссылки и указатели

Одним из способов, с помощью которого общедоступное наследование позволяет моделировать отношение *is-a*, заключается в том, как при этом обрабатываются указатели и ссылки на объекты. Обычно C++ не позволяет присваивать адрес одного типа указателю другого типа. Он также не позволяет ссылке на один тип ссылаться на другой тип:

```
double x = 2.5;
int * pi = &x; // недопустимое
 // присвоение, несоответствующие
 // типы указателей
long & rl = x; // недопустимое
 // назначение, несоответствующие
 // типы ссылок
```

Однако ссылка или указатель на базовый класс может ссылаться на объект производного класса без выполнения явного приведения типа. Например, следующие инициализации являются допустимыми:

```
Overdraft dilly ("Annie Dill", 493222,
 2000);
BankAccount * pb = &dilly; // верно
BankAccount & rb = dilly; // верно
```

Преобразование ссылки или указателя производного класса в ссылку или указатель базового класса называется *приведением вверх*, и оно всегда допускается для общедоступного наследования, не требуя явного приведения типа. Это правило — часть выражения отношения *is-a*. Объект **Overdraft** является объектом **BankAccount** в том смысле, что он наследует все элементы данных и функции-элементы объекта **BankAccount**. Следовательно, объектом **Overdraft** можно манипулировать так же, как и объектом **BankAccount**. Так, функция, созданная для обработки ссылки **BankAccount**, может без особых проблем выполнять те же самые действия по отношению к объекту **Overdraft**. Эта же идея применяется при передаче указателя на объект в качестве аргумента функции.

Противоположный процесс — преобразование указателя или ссылки базового класса в указатель или ссылку производного класса — называется *приведением вниз*, и он не допускается без явного приведения типа. Причина этого ограничения заключается в том, что в общем случае отношение *is-a* необратимо. Производный класс мог бы добавлять новые элементы данных, а использующие эти элементы данных функции-элементы класса не были бы применимы к базовому классу. Например, предположим, что вы производите класс **Singer** из класса **Employee**, добавляя элемент данных, представляющий

вокальный диапазон певца, и функцию-элемент, названную `range()`, которая сообщает значение вокального диапазона. Было бы бессмысленным применять метод `range()` к объекту `Employee`. Но если бы неявное приведение вниз было допустимым, можно было бы случайно установить указатель на `Singer` на адрес объекта `Employee` и использовать этот указатель для вызова метода `range()` (рис. 12.4).

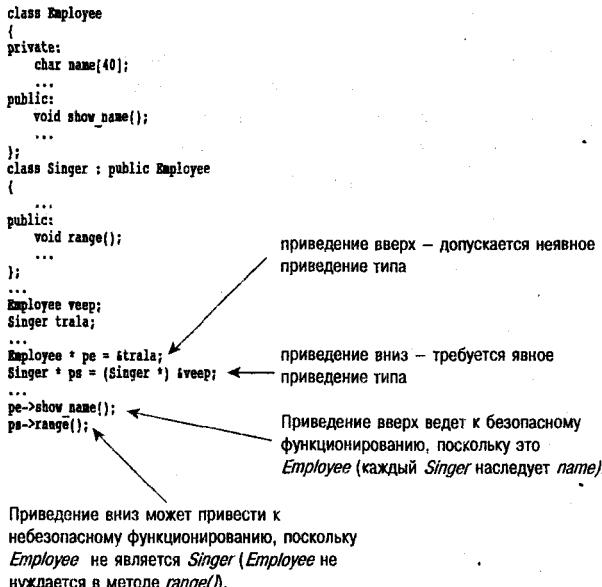
## Виртуальные функции-элементы

Давайте подробнее рассмотрим, какие методы вызываются, когда производный класс переопределяет метод базового класса. Ранее было показано, что тип объекта, вызывающего метод, определял использованный метод:

```
BankAccount bretta; // объект базового
 // класса
Overdraft ophelia; // объект производного
 // класса
bretta.ViewAcct(); // использует метод
 // BankAccount::ViewAcct()
ophelia.ViewAcct(); // использует метод
 // Overdraft::ViewAcct()
```

Предположим, что для вызова метода используется указатель:

```
BankAccount * bp = &bretta; // указывает
 // на объект BankAccount
bp->ViewAcct(); // использует метод
 // BankAccount::ViewAcct()
bp = &ophelia; // указатель
 // BankAccount на объект Overdraft
bp->ViewAcct(); // какая версия?
```



Если компилятор следует за типом указателя, последний оператор вызвал бы метод `BankAccount::ViewAcct()`, но если он следует за типом объекта, на который указывает указатель, он вызвал бы метод `Overdraft::ViewAcct()`. Так что же выбирает компилятор?

По умолчанию C++ применяет тип указателя или ссылки для выбора функций, которые следует использовать, игнорируя тип объекта, на который указывает указатель или делается ссылка. Таким образом, в выше приведенном примере программа использовала бы метод `BankAccount::ViewAcct()`. Для этого существует веская причина — часто тип неизвестен компилятору. Например, давайте рассмотрим следующий программный код:

```
cout << "Enter 1 for Brass Account, 2 for
Brass Plus Account: ";
int kind;
cin >> kind;
BankAccount * bp;
if (kind == 1)
 bp = new BankAccount;
else if (kind == 2)
 bp = new Overdraft;
bp->ViewAcct();
```

Во время компиляции компилятор не может знать, какой выбор (1 или 2) будет сделан во время выполнения, так как он не может знать, на какой тип объект указывает `bp`. Поэтому единственный выбор, который компилятор может сделать во время компиляции, — это поставить методы класса в соответствие с типом ссылки или указателя. Эта стратегия называется *ранним* или *статическим связыванием*. Термин "связывание" относится к присоединению вызова функции к конкретному определению функции. Например, в данном случае компилятор должен связать вызов функции `bp->ViewAcct()` с одним из двух методов `ViewAcct()`. (В языке С может существовать только одна функция с данным именем, поэтому выбор очевиден. Но C++, поддерживающий перегрузку функций и переопределяемые функции-элементы, может иметь больше чем одну функцию, соответствующую данному имени.)

```
// при выполнении статического связывания
bp = &ophelia; // указатель BankAccount
 // на объект Overdraft
bp->ViewAcct(); // использование метода
 // BankAccount::ViewAcct()
```

Что ж, использование метода `BankAccount::ViewAcct()` с объектом `Overdraft` не причиняет вреда; только при этом отображаются не все доступные данные. Поэтому было бы хорошо, если бы в результате вызова функции `bp->ViewAcct()` можно было бы так или иначе включать тип объекта вместо типа указателя и вызывать метод `Overdraft::ViewAcct()` вместо `BankAccount::ViewAcct()`. В этом случае C++ предлагает вторую стратегию, назван-

РИСУНОК 12.4 Приведение вверх и приведение вниз

ную поздним или динамическим связыванием. При использовании этой стратегии компилятор не решает, какой метод класса использовать. Он передает ответственность программе, которая затем принимает решение во время выполнения всякий раз, когда она фактически выполняет вызов функции метода. При использовании этой стратегии программа может выбирать метод, исходя из типа объекта, к которому обращается ссылка или указатель:

```
// при выполнении динамического связывания
BankAccount * bp = &beretta; // указывает
 // на объект BankAccount
bp->ViewAcct(); // использование метода
 // BankAccount::ViewAcct()
bp = &ophelia; // указатель BankAccount на
 // объект Overdraft object
bp->ViewAcct(); // использование метода
 // Overdraft::ViewAcct()
```

В большинстве случаев динамическое связывание весьма удобно.

Проведенное рассмотрение должно было породить ряд вопросов:

- Как активизировать динамическое связывание?
- Почему имеется два вида связывания?
- Если динамическое связывание настолько хорошо, почему оно не выбирается по умолчанию?
- Как оно работает?

Далее мы постараемся ответить на все эти вопросы.

## Активизация динамического связывания

Динамическое связывание используется только для функций-элементов. Для этого перед прототипом функции в объявлении базового класса необходимо ввести ключевое слово **virtual**. После этого метод будет обозначаться термином *виртуальный метод*. Если затем переопределить функцию в производном классе, программа использует динамическое связывание для выбора подлежащего использованию определения. Как только метод станет виртуальным, он остается виртуальным для всех классов, производных из базового, а также для любых классов, производных от производных классов. Для данного метода ключевое слово **virtual** должно использоваться только однажды — в базовом классе, в котором виртуальный метод определен впервые.

### ПОМНИТЕ

Виртуальные функции-элементы создаются путем помещения ключевого слова **virtual** перед прототипом. Программы C++ выполняют динамическое или позднее связывание для виртуальных методов и статическое или раннее связывание — для невиртуальных методов. Для виртуальных функций тип объекта, на который указывает ссылка или указатель, определяет метод, вызываемый указателем или ссылкой.

В листинге 12.7 приведено объявление **BankAccount** после выполнения изменения для создания виртуальных функций. Этот листинг идентичен листингу 12.4, за исключением того, что ключевое слово **virtual** было вставлено дважды — при объявлении **Deposit()** и при объявлении **ViewAcct()**. Изменения этих двух объявлений — единственные, которые необходимо выполнить. Остальные три файла, поддерживающие оба класса, остаются неизменными. Однако их следует скомпилировать снова, используя новый заголовок.

### Листинг 12.7 Класс bankacct.h

```
// bankacct.h - простой класс BankAccount с
// виртуальными функциями
#ifndef _BANKACCT_H_
#define _BANKACCT_H_

class BankAccount
{
private:
 enum {MAX = 35};
 char fullName[MAX];
 long acctNum;
 double balance;
public:
 BankAccount(const char *s = "Nullbody",
 long an = -1, double bal = 0.0);
 void Deposit(double amt);
 virtual void Withdraw(double amt);
 // виртуальный метод
 double Balance() const;
 virtual void ViewAcct() const;
 // виртуальный метод
};

#endif
```

Чтобы продемонстрировать различие между виртуальными и невиртуальными функциями, выполните программу, приведенную в листинге 12.8, дважды. В первый раз скомпилируйте программу, используя первоначальную, невиртуальную версию **bankacct.h** (листинг 12.4). Во второй раз скомпилируйте ее, используя новую, виртуальную версию **bankacct.h** (листинг 12.7). Сама программа использует небольшой массив указателей на **BankAccount**. В результате решения, принятого во время выполнения, каждый указатель в массиве может указывать либо на объект **BankAccount**, либо на объект **Overdraft**. Это использование массива указателей базового класса — общепринятая методика программирования. При наличии смешанного множества объектов **BankAccount** и **Overdraft** их нельзя поместить в один и тот же массив, потому что каждый элемент массива должен иметь тот же самый тип. Но благодаря свойству приведения вверх, присущему языку C++, можно создавать массив указателей базового класса, которые могут указывать либо на объекты базового класса, либо на объекты производного класса.

## Листинг 12.8 Программа useover1.cpp.

```

// useover1.cpp--проверяет класс Overdraft
// компилировать с bankacct.cpp и overdraft.cpp
#include <iostream>
using namespace std;
#include "overdrft.h"
const int ASIZE = 3;
const int MAX = 35;
inline void EatLine() {while (cin.get() != '\n') continue; }

int main()
{
 BankAccount * baps[ASIZE];
 char name[MAX];
 long acctNum;
 double balance;
 int acctType;
 int i;
 for (i = 0; i < ASIZE; i++)
 {
 cout << "Enter client's name: ";
 cin.get(name,MAX);
 EatLine();
 cout << "Enter client's account number: ";
 cin >> acctNum;
 cout << "Enter client's initial balance: ";
 cin >> balance;
 cout << "Enter 1 for Brass Account, 2 for Brass Plus " << "Account: ";
 cin >> acctType;
 EatLine();
 if (acctType == 2) baps[i] = new Overdraft(name, acctNum, balance);
 else
 {
 baps[i] = new BankAccount(name, acctNum, balance);
 if (acctType != 1) cout << "I'll interpret that as a 1.\n";
 }
 }
 for (i = 0; i < ASIZE; i++)
 {
 baps[i]->ViewAcct();
 cout << endl;
 }
 cout << "Bye!\n";
 return 0;
}

```

Таким образом, можно использовать единственный массив указателей базового класса для управления смешанным множеством типов объектов.

Прежде всего посмотрим, как выглядит типовое выполнение программы с использованием невиртуальных функций:

```

Enter client's name: Rufus Overbeam
Enter client's account number: 123984
Enter client's initial balance: 3000
Enter 1 for Brass Account, 2 for Brass
 Plus Account: 2
Enter client's name: Lily Goldleaf
Enter client's account number: 829302
Enter client's initial balance: 4000
Enter 1 for Brass Account, 2 for Brass

```

```

Plus Account: 1
Enter client's name: Harry Grub
Enter client's account number: 111223
Enter client's initial balance: 22480
Enter 1 for Brass Account, 2 for Brass
 Plus Account: 1
Client: Rufus Overbeam
Account Number: 123984
Balance: $3000.00
Client: Lily Goldleaf
Account Number: 829302
Balance: $4000.00
Client: Harry Grub
Account Number: 111223
Balance: $22480.00
Bye!

```

Обратите внимание, что даже при том, что клиент Rufus Overbeam имеет счет "Медный Плюс" и представлен объектом **Overdraft**, использование указателя **BankAccount** приводит к задействованию метода **BankAccount::ViewAcct()**. Метод соответствует типу указателя; статическое связывание получает приоритет.

А вот как выглядит типичное выполнение программы с использованием тех же самых входных данных, но при задействовании виртуальной версии заголовочного файла (листинг 12.7):

```
Enter client's name: Rufus Overbeam
Enter client's account number: 123984
Enter client's initial balance: 3000
Enter 1 for Brass Account, 2 for Brass
 Plus Account: 2
Enter client's name: Lily Goldleaf
Enter client's account number: 829302
Enter client's initial balance: 4000
Enter 1 for Brass Account, 2 for Brass
 Plus Account: 1
Enter client's name: Harry Grub
Enter client's account number: 111223
Enter client's initial balance: 22480
Enter 1 for Brass Account, 2 for Brass
 Plus Account: 1
Client: Rufus Overbeam
Account Number: 123984
Balance: $3000.00
Maximum loan: $500.00
Owed to bank: $0.00

Client: Lily Goldleaf
Account Number: 829302
Balance: $4000.00

Client: Harry Grub
Account Number: 111223
Balance: $22480.00
Bye!
```

Rufus Overbeam так же, как и в первом случае, имеет счет "Медный Плюс" и представлен объектом **Overdraft**, но на сей раз программа использует метод **Overdraft::ViewAcct()**. Метод соответствует типу объекта; динамическое связывание получает приоритет.

Поскольку функция **ViewAcct()** виртуальная, оператор

```
baps[i]->ViewAcct();
```

может вызывать метод **Overdraft::ViewAcct()** в одних случаях и **BankAccount::ViewAcct()** — в других. Подобно перегрузке функций, это служит примером полиморфизма. Один и тот же код может относиться к различным классам и различным функциям, в зависимости от контекста.

## Зачем нужны два вида связывания?

Динамическое связывание позволяет переопределять методы класса, в то время как статическое связывание является лишь частичным средством. В связи с этим возникает вопрос: зачем вообще нужно статическое связывание? На это существует две причины, связанные с эффективностью и концептуальной моделью.

Вначале давайте рассмотрим вопросы эффективности. Чтобы программа могла принимать решение во время выполнения, она должна располагать каким-либо способом, позволяющим отслеживать, на какого рода объект ссылается указатель или ссылка базового класса, а это влечет за собой некоторую дополнительную перегрузку во время обработки. (Один из методов динамического связывания будет описан далее.) Если, например, вы разрабатываете класс, который не будет использоваться в качестве базового класса для наследования, свойство динамического связывания не требуется. В этом случае имеет смысл выполнять статическое связывание и получить некоторый выигрыш в эффективности. Именно потому, что статическое связывание является более эффективным, в C++ оно выбрано по умолчанию. Как сказал Стравуструп (Stroustrup), один из основных принципов C++ заключается в том, что не следует платить (дополнительным объемом используемой памяти или временем обработки) за те свойства, которые не используются. К виртуальным функциям следует прибегать, только если конкретная программа нуждается в них.

Теперь давайте рассмотрим концептуальную модель. При разработке класса могут использоваться функции-элементы, которые не нужно переопределять в производных классах. Например, функцию **BankAccount::Balance()**, которая возвращает баланс счета, не следует переопределять. Делая эту функцию не виртуальной, вы достигаете двух целей. Во-первых, она становится более эффективной. Во-вторых, вы объявляете о том, что эта функция не должна переопределяться. Это диктует следующее важное правило.



### СОВЕТ

Если метод в базовом классе будет переопределен в производном классе, его следует делать виртуальным. Если метод не должен быть переопределен, делайте его не виртуальным.

Конечно, во время разработки класса не всегда ясно, к какой категории относится метод. Подобно многим аспектам реальной жизни, разработка класса — не однозначный процесс.

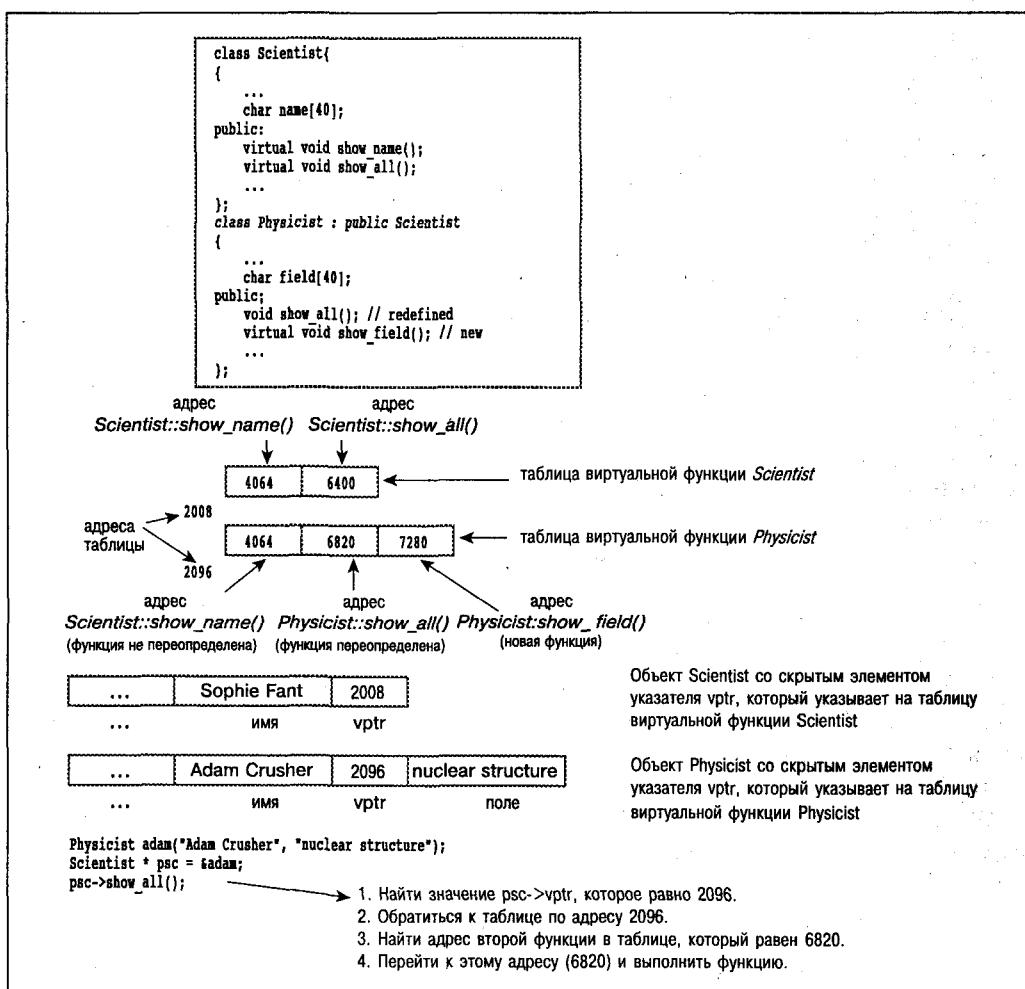
## Как работают виртуальные функции

C++ определяет, как виртуальные функции должны вести себя, но реализацию он оставляет автору компилятора. Чтобы использовать виртуальные функции, вовсе не обязательно знать метод реализации, но исследование того, как это выполнено, может помочь лучше понять концепции, поэтому давайте продолжим рассмотрение.

Обычно компиляторы обрабатывают виртуальные функции, добавляя скрытый элемент к каждому объекту. Скрытый элемент содержит указатель на массив адресов функций. Обычно такой массив называют *таблицей*. Таблица содержит адреса виртуальных функций, объявленных для объектов этого класса. Например, объект базового класса будет содержать указатель на таблицу адресов всех виртуальных функций для этого класса. Объект производного класса будет содержать указатель на отдельную таблицу адресов. Если производный класс обеспечивает новое определение виртуальной фун-

кции, таблица содержит адрес новой функции. Если производный класс не переопределяет виртуальную функцию, таблица содержит адрес первоначальной версии функции. Если производный класс определяет новую функцию и делает ее виртуальной, ее адрес добавляется в таблицу (рис. 12.5). Обратите внимание, что независимо от того, определяется ли одна или десять виртуальных функций для класса, к объекту добавляется только один элемент адреса; это размер таблицы, который изменяется.

Когда вызывается виртуальная функция, программа просматривает адрес таблицы, сохраненный в объекте, и обращается к соответствующей таблице адресов функций. Если используется первая виртуальная функция, определенная в объявлении класса, программа использует первый адрес функции в массиве и выполняет функцию, имеющую этот адрес. Если используется третья виртуальная функция в объявлении класса, программа использует функцию, адрес которой находится в третьем элементе массива.



Таким образом, использование виртуальных функций сопряжено со следующими небольшими издержками в смысле используемого объема памяти и быстродействия программы:

- Размер каждого объекта увеличивается на величину, необходимую для хранения адреса.
- Для каждого класса компилятор создает таблицу (массив) адресов виртуальных функций.
- Для каждого обращения к функции выполняется дополнительное продвижение по таблице для отыскания адреса.

Имейте в виду, что, хотя невиртуальные функции несколько более эффективны, чем виртуальные, они не обеспечивают динамическое связывание.

## Что следует знать о виртуальных функциях

Мы уже рассмотрели основные моменты относительно виртуальных функций:

- Начало объявления метода класса в базовом классе с ключевого слова `virtual` делает функцию виртуальной для базового класса и всех классов, производных из базового, включая классы, производные из производных классов.
- Если виртуальный метод вызывается с помощью ссылки на объект или указателя на объект, программа будет использовать метод, определенный для типа объекта, а не метод, определенный для типа указателя или ссылки. Это называется динамическим или поздним связыванием. Это поведение важно, поскольку оно всегда допускается, чтобы указатель или ссылка базового класса ссылались на объект производного типа.
- Если вы определяете класс, который будет использоваться как базовый для наследования, в качестве виртуальных функций следует объявлять те методы класса, которые возможно придется переопределять в производных классах.

Существуют и другие моменты, касающиеся виртуальных функций и заслуживающие внимание. Давайте рассмотрим их.

### Конструкторы

Конструкторы не могут быть виртуальными. Производный класс не наследует конструкторы базового класса, поэтому нет особого смысла делать их виртуальными.

### Деструкторы

Деструкторы должны быть виртуальными, если только класс не должен использоваться в качестве базового. Например, предположим, что `Employee` — базовый класс,

а `Singer` — производный класс, который добавляет элемент `char *`, указывающий на память, выделенную объектом `new`. Когда срок существования объекта `Singer` истечет, важно, чтобы деструктор `~Singer()` был вызван для освобождения этой памяти.

Теперь давайте рассмотрим следующий код:

```
Employee * pe = new Singer; // допустимо,
 // поскольку Employee является
 // базовым для Singer
...
delete pe; // ~Employee() или ~Singer()?
```

Если выполняется заданное по умолчанию статическое связывание, оператор `delete` вызовет деструктор `~Employee()`. При этом освобождается память, указанная компонентами `Employee` объекта `Singer`, но не память, указанная новыми элементами класса. Однако, если деструкторы являются виртуальными, тот же самый код вызывает деструктор `~Singer()`, который освобождает память, указанную компонентом `Singer`, а затем вызывает деструктор `~Employee()`, чтобы освободить память, указанную компонентом `Employee`.

Помните, что, даже если базовый класс не требует выполнения явного деструктора, не следует полагаться на заданный по умолчанию конструктор. Вместо этого обеспечьте наличие виртуального деструктора, даже если ему нечего делать:

```
virtual ~BaseClass() { }
```



### СОВЕТ

Целесообразно снабдить базовый класс виртуальным деструктором, даже если класс не нуждается в нем.

## Дружественные конструкции

Дружественные функции не могут быть виртуальными, поскольку они не являются элементами класса. Если это создает проблему для проекта, можно выйти из положения, применив дружественные функции, которые используют внутри себя виртуальные функции-элементы.

### Отсутствие переопределения

Если производному классу не удается переопределить виртуальную функцию, класс использует версию функции базового класса. Если производный класс — часть длинной цепочки получения производных классов, он будет использовать ту версию виртуальной функции, которая определена последней по времени. Исключением является ситуация, когда базовые версии скрыты, как описано далее.

### Переопределение скрывает методы

Предположим, что вы создаете следующую конструкцию:

```

class Dwelling
{
public:
 virtual void showperks(int a) const;
...
};

class Hovel : public Dwelling
{
public:
 void showperks();
...
};

```

Это создает проблему. При этом можно получить такое предупреждение компилятора:

```
Warning: Hovel::showperks(void) hides
Dwelling::showperks(int)
```

Возможно, предупреждение и не будет получено. В любом случае код имеет следующие толкования:

```

Hovel trump;
trump.showperks(); // допустимо
trump.showperks(5); // недопустимо

```

Новое объявление определяет функцию `showperks()`, которая не принимает никакие аргументы. Вместо того чтобы создавать две перегруженные версии функции, это переопределение скрывает версию базового класса, которая принимает аргумент `int`. Одним словом, переопределение унаследованных методов не является разновидностью перегрузки. В случае переопределения функции в производном классе она не просто замещает объявление базового класса сигнатурой той же самой функции. Вместо этого переопределение скрывает все методы базового класса с тем же самым именем, независимо от сигнатур аргумента.

Это обстоятельство диктует несколько важных правил. Во-первых, если вы переопределяете унаследованный метод, убедитесь в точном соответствии первоначальному прототипу. Единственное исключение состоит в том, что возвращаемый тип, являющийся ссылкой или указателем на базовый класс, может быть заменен ссылкой или указателем на производный класс. (Это исключение является новым, и еще не все компиляторы его распознают. Обратите внимание также, что это исключение применяется только к возвращаемым значениям, но не к аргументам.) Во-вторых, если объявление базового класса перегружается, переопределите все версии базового класса в производном классе:

```

class Dwelling
{
public:
// три перегруженные функции showperks()
 virtual void showperks(int a) const;
 virtual void showperks(double x)
 const;

```

```

 virtual void showperks() const;
};

class Hovel : public Dwelling
{
public:
// три переопределенные функции showperks()
 void showperks(int a) const;
 void showperks(double x) const;
 void showperks() const;
...
};

```

Если переопределить только одну версию, две другие становятся скрытыми и не могут использоваться объектами производного класса. Обратите внимание, что, если никакое изменение не требуется, в результате переопределения может просто вызываться версия базового класса.

## Наследование и присваивание

Оператор присваивания, как упоминалось ранее, является одной из функций-элементов, которая не наследуется. Давайте подробнее обсудим тему присваивания и наследования. Начнем с некоторых основ. Рассмотрите следующий код:

```

BankAccount darf("Darfa Flemwit", 121234,
100);
BankAccount temp1;
Overdraft bip("Bipp Fardbag", 212143, 200);
Overdraft temp2;
temp1 = darf;
temp2 = bip;

```

Присваивания переменным `temp1` и `temp2` эквивалентны следующим обращениям к функциям:

```
temp1.operator=(durf); // вызов #1
temp2.operator=(bip); // вызов #2
```

Поскольку обращение #1 вызывается объектом `BankAccount`, компилятор будет искать функцию `BankAccount::operator=()`. Аргумент также является объектом `BankAccount`, поэтому компилятор будет искать функцию, которая принимает аргумент `BankAccount`. Оба требования выполняются заданным по умолчанию оператором присваивания который, как вы помните, имеет следующий прототип:

```
BankAccount & BankAccount::operator=(const
BankAccount &);
```

Аналогично, вызов #2, приводится в соответствие следующим прототипом:

```
Overdraft & Overdraft::operator=(const
Overdraft &);
```

Это опять-таки функция `operator=()`, сгенерированная по умолчанию для класса `Overdraft`. Она имеет кор-

ректную сигнатуру функции для присваивания одного объекта **Overdraft** другому. Но если класс **Overdraft** унаследовал функцию **operator=()** **BankAccount**, унаследованная версия будет иметь неправильную сигнатуру функции для обработки объектов **Overdraft**, потому что ее аргумент — ссылка **BankAccount**. Таким образом, вместо того чтобы позволить производному классу наследовать оператор присваивания, компилятор автоматически определяет новый оператор для класса.

## Смешанное присваивание

Что ж, можно присвоить один объект **BankAccount** другому так же, как и объект **Overdraft**, используя заданные по умолчанию операторы присваивания. А как насчет присвоения объекта **Overdraft** объекту **BankAccount** и наоборот? Давайте вначале рассмотрим присвоение производного объекта базовому объекту:

```
BankAccount temp;
Overdraft bip("Bipp Fardbag", 212143, 200);
temp = bip; // это возможно?
```

Итак, присваивание выполняется, хотя и несколько необычно. Вначале компилятор помещает следующую конструкцию в запись функции:

```
temp.operator=(bip);
```

Затем компилятор пытается найти оператор присваивания, который соответствует вызову функции. Поскольку вызывающий объект принадлежит к классу **BankAccount**, а аргумент принадлежит к классу **Overdraft**, идеальным соответствием было бы следующее:

```
BankAccount & BankAccount::operator=(const
Overdraft &);
```

Обратите внимание, что функция обязательно является элементом класса **BankAccount**. Что ж, это не заданный по умолчанию оператор присваивания, и он не объявлен в классе **BankAccount**, поэтому нет никакого точного соответствия. Следовательно, компилятор выясняет, имеется ли функция оператора, которая будет работать после преобразования типов для аргумента. И такая функция существует:

```
BankAccount & BankAccount::operator=(const
BankAccount &);
```

Вспомните, что ссылка базового класса может ссылаться на объект производного класса (приведение вверх) без явного приведения типа. Таким образом, заданный по умолчанию оператор присваивания для базового класса примет объект производного класса. И это означает, что можно присваивать объект производного класса объекту базового класса. В результате только часть базового класса производного объекта копируется.

### ПОМНИТЕ

Можно присваивать объект производного класса объекту базового класса. Компилятор использует оператор присваивания базового класса и скопирует только часть базового класса.

Далее давайте попытаемся присвоить объект базового класса объекту производного класса:

```
BankAccount darf("Darfa Flemwit", 121234,
100);
Overdraft temp;
temp = darf; // это возможно?
```

На сей раз возможно. Этот оператор присваивания преобразуется в следующий вызов:

```
temp.operator=(darf);
```

Поскольку вызывающий объект является объектом **Overdraft**, соответствующая функция присваивания, если она существует, должна быть элементом класса **Overdraft** с аргументом класса **BankAccount**:

```
Overdraft & Overdraft::operator=(const
BankAccount &);
```

Ни одна такая функция не существует, но имеется заданный по умолчанию оператор присваивания:

```
Overdraft & Overdraft::operator=(const
Overdraft &);
```

Может ли он использоваться? Это требовало бы наличия ссылки класса **Overdraft**, ссылающейся на объект класса **BankAccount**. Тогда ссылка производного класса должна была бы ссылаться на объект базового класса. Такое приведение вниз допускается только с явным приведением типа, поэтому обычно это присваивание не допускается. Однако в данном случае имеется конструктор с единственным аргументом, который обеспечивает неявное преобразование от **BankAccount** к **Overdraft**:

```
Overdraft(const BankAccount & ba, double ml
= 500, double r = 0.1);
```

Таким образом, в этом конкретном случае программа вызывает конструктор **Overdraft(darf)**, чтобы сгенерировать временный объект **Overdraft**, который используется в качестве аргумента для заданной по умолчанию функции **operator=()** класса **Overdraft**. Если конструктор был объявлен как явный, присваивание будет отвергнуто.

### ПОМНИТЕ

В общем случае нельзя присваивать объект базового класса объекту производного класса. Однако такое присваивание возможно, если имеется конструктор, который определяет преобразование базового класса в производный класс.

## Присваивание и динамическое распределение памяти

Теперь посмотрим, как динамическое распределение памяти взаимодействует с присваиванием и наследованием. Вначале давайте изменим класс `BankAccount` так,

чтобы он выполнял динамическое распределение памяти. В листинге 12.9 приведено новое объявление класса, а в листинге 12.10 — новая реализация. Как обычно, добавление динамического распределения памяти потребовало добавления явного деструктора, конструктора копии и оператора присваивания.

### Листинг 12.9 Класс bankdyn.h.

```
// bankdyn.h - простой класс BankAccountD с DMA
#ifndef _BANKDYN_H_
#define _BANKDYN_H_

class BankAccountD
{
private:
 char * fullName;
 long acctNum;
 double balance;
public:
 BankAccountD(const char *s = "Nullbody", long an = -1, double bal = 0.0);
 BankAccountD(const BankAccountD & ba);
 virtual ~BankAccountD();
 void Deposit(double amt);
 virtual void Withdraw(double amt); // виртуальный метод
 double Balance() const;
 virtual void ViewAcct() const; // виртуальный метод
 BankAccountD & operator=(const BankAccountD & ba);
};
#endif
```

### Листинг 12.10 Программа bankdyn.cpp.

```
// bankdyn.cpp--методы для класса BankAccountD
#include <iostream>
using namespace std;
#include "bankdyn.h"
#include <cstring>

BankAccountD::BankAccountD(const char *s, long an, double bal)
{
 fullName = new char[strlen(s) + 1];
 strcpy(fullName, s);
 acctNum = an;
 balance = bal;
}

BankAccountD::BankAccountD(const BankAccountD & ba)
{
 fullName = new char[strlen(ba.fullName) + 1];
 strcpy(fullName, ba.fullName);
 acctNum = ba.acctNum;
 balance = ba.balance;;
}

BankAccountD::~BankAccountD()
{
 delete [] fullName;
}

void BankAccountD::Deposit(double amt)
{
 balance += amt;
}
```

```

void BankAccountD::Withdraw(double amt)
{
 if (amt <= balance)
 balance -= amt;
 else
 cout << "Withdrawal amount of $" << amt << " exceeds your balance.\n"
 << "Withdrawal canceled.\n";
}
double BankAccountD::Balance() const
{
 return balance;
}
void BankAccountD::ViewAcct() const
{
 // set up ####.## format
 ios_base::fmtflags initialState = cout.setf(ios_base::fixed, ios_base::floatfield);
 cout.setf(ios_base::showpoint);
 cout.precision(2);
 cout << "Client: " << fullName << endl;
 cout << "Account Number: " << acctNum << endl;
 cout << "Balance: $" << balance << endl;
 cout.setf(initialState); // восстановление исходного формата
}
BankAccountD & BankAccountD::operator=(const BankAccountD & ba)
{
 if (this == &ba)
 return *this;
 delete [] fullName;
 fullName = new char[strlen(ba.fullName) + 1];
 strcpy(fullName, ba.fullName);
 acctNum = ba.acctNum;
 balance = ba.balance;
 return *this;
}

```

### Случай 1. Производный класс не использует оператор new

Предположите, что вы получаете класс **Overdraft** из **BankAccountD**, а не из **BankAccount**. Нужно ли теперь определять явный деструктор, конструктор копии и оператор присваивания для класса **Overdraft**? Если к классу **Overdraft** не нужно добавлять новые свойства, которым требуются эти методы, ответ — нет.

Вначале давайте выясним, существует ли потребность в деструкторе. Если его не определить, компилятор определит заданный по умолчанию конструктор, который не выполняет никаких функций. Фактически, заданный по умолчанию конструктор для производного класса всегда несет какую-то функциональную нагрузку; после выполнения собственного кода он вызывает деструктор базового класса. Поскольку элементы **Overdraft** не требуют никакого специального действия, заданный по умолчанию деструктор прекрасно подходит.

Теперь рассмотрим конструктор копии. Вы видели, что заданный по умолчанию конструктор копии делает копирование зависящим от элементов, что не подходит для динамического распределения памяти. Однако за-

висящее от элементов копирование прекрасно подходит для трех новых элементов класса **Overdraft**. При этом остается проблема с унаследованным объектом **BankAccountD**. Следует иметь в виду, что при выполнении зависящего от элементов копирования используется форма копирования, которая определена для рассматриваемого типа данных. Так, копирование **long** в **long** выполняется путем использования обычного присваивания. Но копирование элементов класса или унаследованного компонента класса выполняется путем использования конструктора копии для этого класса. Таким образом, заданный по умолчанию конструктор копии для класса **Overdraft** использует явный конструктор копии **BankAccountD** для копирования части **BankAccountD** объекта **Overdraft**. Таким образом, если заданный по умолчанию конструктор копии подходит для новых членов объекта **Overdraft**, он подходит также и для унаследованного объекта **BankAccountD**.

По существу та же самая ситуация характерна и для присваивания. Заданный по умолчанию оператор присваивания для класса автоматически использует оператор присваивания базового класса для компонента базового класса. Так что он тоже прекрасно подходит.

## Случай 2. Производный класс использует оператор new

Предположим, что производный класс использует оператор `new`. Тогда нужно определить явный деструктор, конструктор копии и оператор присваивания для производного класса. Давайте посмотрим, как это выполнено. Предположим, было решено, что держатели счета "Медный Плюс" получают кодовое имя, поэтому для обработки добавлен новый элемент указателя. В листинге 12.11 приведено пересмотренное объявление класса, указывающее новые элементы.

Первоначальные конструкторы должны быть пересмотрены как обычно, чтобы выделить место для строки, а функция `ViewAcct()` должна будет добавить код для отображения нового элемента; эти подробности можно найти в листинге 12.12. А пока давайте сосредоточим свое внимание на новых функциях-элементах.

Прежде всего, имеется конструктор копии. Он может быть закодирован следующим образом:

```
OverdraftD::OverdraftD(const OverdraftD &
 od) : BankAccountD(od)
{
 codeName = new char[strlen(od.codeName) + 1];
 strcpy(codeName, od.codeName);
 maxLoan = od.maxLoan;
 owesBank = od.owesBank;
 rate = od.rate;
}
```

### Листинг 12.11 Объявление overdyn2.h.

```
// overdyn2.h --объявление класса OverdraftD с DMA
#ifndef _OVERDYN_H_
#define _OVERDYN_H_
#include "bankdyn.h"

class OverdraftD : public BankAccountD
{
private:
 double maxLoan;
 double rate;
 double owesBank;
 char * codeName; // новый
public:
 OverdraftD(const char * s = "Nullbody", const char * cn = "cent",
 long an = -1, double bal = 0.0, double ml = 500, double r = 0.10);
 OverdraftD(const BankAccountD & ba,const char * cn = "cent",
 double ml = 500, double r = 0.1);
 OverdraftD(const OverdraftD & od); // новый // новый
 ~OverdraftD(); // новый
 void ViewAcct() const;
 void Withdraw(double amt);
 void ResetMax(double m) { maxLoan = m; }
 void ResetRate(double r) { rate = r; }
 void ResetOwes() { owesBank = 0; }
 OverdraftD & operator=(const OverdraftD & od); // новый
};

#endif
```

Ответственность за создание части `OverdraftD` объекта лежит на конструкторе `OverdraftD`, и функция выполняет распределение памяти для строки `codeName` обычным образом. Задача создания части базового класса лежит на конструкторах базового класса. В данном случае метод использует синтаксис списка инициализатора для вызова конструктора копии базового класса. Обратите внимание, что конструктор копии `OverdraftD` передает ссылку на объект `OverdraftD` конструктору копии `BankAccountD`. Аргумент последнего объявляется имеющим тип `const BankAccountD &`. Однако, поскольку неявное приведение вверх является правилом, ссылка `BankAccountD` может ссылаться на объект `OverdraftD`. В частности, она обратится к компоненту `BankAccountD` объекта `Overdraft`.

Теперь рассмотрим деструктор. Он должен управлять только компонентом `OverdraftD`:

```
OverdraftD::~OverdraftD()
{
 delete [] codeName;
}
```

Помните, что, после того как он вызван, деструктор базового класса вызывается автоматически, освобождая память, указанную переменной `fullName`.

И наконец, имеется оператор присваивания. Здесь важным является следующий момент. Если вы опреде-

ляет оператор присваивания для производного класса, убедитесь, что это присваивание выполняется для частей как производного, так и базового класса. Кроме того, синтаксис списка инициализатора нельзя использовать для части базового класса, так как этот синтаксис может использоваться только в конструкторах. Поэтому оператор присваивания базового класса нужно вызвать в теле функции. Легче всего это выполнить, используя запись оператора присваивания в виде функции:

```
OverdraftD & OverdraftD::operator=(const
 OverdraftD & od)
{
 if (this == &od)
 return *this;
 BankAccountD::operator=(od);
 // присваивание базового класса
 delete [] codeName;
 codeName =
 new char[strlen(od.codeName) + 1];
 strcpy(codeName, od.codeName);
 maxLoan = od.maxLoan;
```

**Листинг 12.12 Программа overdyn2. cpp.**

```
// overdyn2.cpp--методы класса OverdraftD с DMA
#include <iostream>
using namespace std;
#include "overdyn2.h"
OverdraftD::OverdraftD(const char * s, const char * cn, long an, double bal,
 double ml, double r) : BankAccountD(s, an, bal)
{
 codeName = new char[strlen(cn) + 1];
 strcpy(codeName, cn);
 maxLoan = ml;
 owesBank = 0.0;
 rate = r;
}
OverdraftD::OverdraftD(const BankAccountD & ba, const char * cn, double ml, double r)
 : BankAccountD(ba) // использует явный конструктор копии
{
 codeName = new char[strlen(cn) + 1];
 strcpy(codeName, cn);
 maxLoan = ml;
 owesBank = 0.0;
 rate = r;
}
OverdraftD::OverdraftD(const OverdraftD & od) : BankAccountD(od)
{
 codeName = new char[strlen(od.codeName) + 1];
 strcpy(codeName, od.codeName);
 maxLoan = od.maxLoan;
 owesBank = od.owesBank;
 rate = od.rate;
}
OverdraftD::~OverdraftD()
{
 delete [] codeName;
}
// переопределение ViewAcct()
void OverdraftD::ViewAcct() const
{
 // установка формата ###.##
 ios_base::fmtflags initialState =
 cout.setf(ios_base::fixed, ios_base::floatfield);
 cout.setf(ios_base::showpoint);
 cout.precision(2);
 BankAccountD::ViewAcct(); // отображение базовой части
 cout << "Code Name: " << codeName << endl;
```

```
 owesBank = od.owesBank;
 rate = od.rate;
 return *this;
}
```

Оператор:

```
BankAccountD::operator=(od); // присваивание
 // базового класса
```

является краткой формой записи следующего:

```
this->BankAccountD::operator=(od);
 // присваивание базового класса
```

Другими словами, версию **BankAccountD** присваивания следует использовать для присваивания **od** объекту **\*this**. В результате часть **BankAccountD** объекта **od** копируется в часть **BankAccountD** объекта **\*this**. Полная реализация показана в листинге 12.12.

Листинг 12.13, который должен компилироваться совместно с листингами 12.12 и 12.10, выполняет тестирование пересмотренных классов.

```

cout << "Maximum loan: $" << maxLoan << endl;
cout << "Owed to bank: $" << owesBank << endl;
cout.setf(initialState);
}
// переопределение работы Withdraw()
void OverdraftD::Withdraw(double amt)
{
 // установка формата ####.##
 ios_base::fmtflags initialState = cout.setf(ios_base::fixed, ios_base::floatfield);
 cout.setf(ios_base::showpoint);
 cout.precision(2);
 double bal = Balance();
 if (amt <= bal)
 BankAccountD::Withdraw(amt);
 else if (amt <= bal + maxLoan - owesBank)
 {
 double advance = amt - bal;
 owesBank += advance * (1.0 + rate);
 cout << "Bank advance: $" << advance << endl;
 cout << "Finance charge: $" << advance * rate << endl;
 Deposit(advance);
 BankAccountD::Withdraw(amt);
 }
 else
 cout << "Credit limit exceeded. Transaction cancelled.\n";
 cout.setf(initialState);
}
OverdraftD & OverdraftD::operator=(const OverdraftD & od)
{
 if (this == &od)
 return *this;
 BankAccountD::operator=(od); // присваивание базового класса
 delete [] codeName;
 codeName = new char[strlen(od.codeName) + 1];
 strcpy(codeName, od.codeName);
 maxLoan = od.maxLoan;
 owesBank = od.owesBank;
 rate = od.rate;
 return *this;
}

```

### Листинг 12.13 Программа usedyn2. cpp.

```

// usedyn2.cpp--тестирует класс OverdraftD
// Компилировать с bankdyn.cpp и overdyn2.cpp
#include <iostream>
using namespace std;
#include "overdyn2.h"
int main()
{
 BankAccountD dolly("Dahlia Dahl", 453216, 6000);
 BankAccountD temp;
 temp = dolly;
 temp.ViewAcct();
 cout << endl;
 OverdraftD roddy("Roland Rayleigh", "Rocky", 223391, 8000);
 OverdraftD dup;
 dup = roddy;
 dup.ViewAcct();
 cout << endl;
 dup = dolly;
 dup.ViewAcct();
 cout << "Bye!\n";
 return 0;
}

```

Результаты выполнения программы:

```
Client: Dahlia Dahl
Account Number: 453216
Balance: $6000.00
```

```
Client: Roland Rayleigh
Account Number: 223391
Balance: $8000.00
Code Name: Rocky
Maximum loan: $500.00
Owed to bank: $0.00
```

```
Client: Dahlia Dahl
Account Number: 453216
Balance: $6000.00
Code Name: cent
Maximum loan: $500.00
Owed to bank: $0.00
Bye!
```

Как видите, присваивание было выполнено по отношению к обоим классам. А благодаря конструктору в объекте **OverdraftD**, который действует в качестве неявной функции преобразования **BankAccountD**-в-**OverdraftD**, выполнено даже перекрестное присваивание. Обратите внимание, что присваивание **dolly** объекту **dup** привело к тому, что объект **dup** получил значения, принятые по умолчанию, для членов **OverDraftD**, поскольку конструктор преобразования делает следующее:

```
OverdraftD(const BankAccountD & ba, const
char * cn = "cent", double ml =
500, double r = 0.1);
```

## Абстрактные базовые классы

Иногда применение правила *is-a* не столь просто, как это могло бы показаться. Предположим, например, что вы разрабатываете графическую программу, которая, как предполагается, рисует, кроме всего прочего, окружности и эллипсы. Окружность — специальный случай эллипса; это эллипс, длинная ось которого равна короткой оси. Следовательно, все окружности — эллипсы, и это обстоятельство приводит к мысли получить класс **Circle** из класса **Ellipse**. Но как только дело доходит до деталей реализации, могут возникнуть проблемы.

Чтобы убедиться в этом, вначале рассмотрим объекты, которые можно было бы включить в качестве части класса **Ellipse**. Элементы данных могли бы включать координаты центра эллипса, главную полуось (половину длинного диаметра), малую полуось (половину короткого диаметра) и угол ориентации, определяющий угол между горизонтальной осью координат и главной полуосью эллипса. Класс также мог бы включать методы для

перемещения эллипса, для возвращения площади эллипса, для вращения эллипса и для масштабирования главной и малой полуосей:

```
class Ellipse
{
private:
 double x; // координата x центра
 // эллипса
 double y; // координата y центра
 // эллипса
 double a; // главная полуось
 double b; // малая полуось
 double angle; // угол ориентации в
 // градусах
 ...
public:
 ...
 void Move(int nx, ny) { x = nx; y =
 ny; }
 virtual double Area() const { return
 3.14159 * a * b; }
 virtual void Rotate(double nang) {
 angle = nang; }
 virtual void Scale(double sa, double
 sb) { a *= sa; b *= sb; }
 ...
};
```

Теперь предположите, что вы получаете класс **Circle**:

```
class Circle : public Ellipse
{
 ...
};
```

Хотя окружность — это эллипс, такое образование не слишком удачно. Например, для описания размера и формы окружности требуется только единственное значение — радиус вместо значений главной полуоси (**a**) и малой полуоси (**b**). Конструкторы **Circle** могут позаботиться об этом, присваивая одно и то же значение элементам **a** и **b**, но тогда будет избыточное представление одной и той же информации. Параметр **angle** и метод **Rotate()** в действительности не имеют смысла для окружности, а метод **Scale()** в существующем виде может превращать окружность в другую геометрическую форму, масштабируя две оси по-разному. Можно попытаться поправить положение дел с помощью таких приемов, как помещение переопределенного метода **Rotate()** в приватном разделе класса **Circle**, — чтобы **Rotate()** нельзя было использовать вместе с окружностью. Но в целом, похоже, проще определить класс **Circle**, не используя метод наследования:

```
class Circle // наследование отсутствует
{
```

```

private:
 double x; // координата x центра
 // окружности
 double y; // координата y центра
 // окружности
 double r; // радиус
 ...
public:
 ...
 void Move(int nx, ny) { x = nx; y = ny; }
 double Area() const { return 3.14159 * r * r; }
 void Scale(double sr) { r *= sr; }
 ...
};


```

Теперь класс содержит только те элементы, в которых он нуждается. Все же это решение также кажется не самым лучшим. Классы **Ellipse** и **Circle** имеет много общего, но при раздельном их определении общие свойства не используются.

Существует другое решение, и оно заключается в извлечении из классов **Ellipse** и **Circle** общих свойств и помещении их в *абстрактный базовый класс* (ABC). Затем следует получить классы **Circle** и **Ellipse** из ABC. После этого, например, можно использовать массив указателей базового класса для управления смешанным множеством объектов **Ellipse** и **Circle** (т.е. можно использовать полиморфный подход). В данном случае общими для обоих классов являются координаты центра формы. Метод **Move()** работает одинаково для обоих классов, а метод **Area()** — по-разному. Действительно, метод **Area()** даже не может быть реализован для ABC, поскольку он не имеет необходимых элементов данных. В C++ существует способ обеспечения нереализованной функции — использование *чисто виртуальной функции*. Чисто виртуальная функция содержит последовательность = 0 в конце своего объявления, как показано ниже:

```

class BaseEllipse // абстрактный базовый
 // класс
{
private:
 double x; // координата x центра
 double y; // координата y центра
 ...
public:
 BaseEllipse(double x0 = 0, double y0 = 0)
 : x(x0), y(y0) { }
 virtual ~BaseEllipse() { }
 void Move(int nx, ny)
 { x = nx; y = ny; }
 virtual double Area() const = 0;
 // чисто виртуальная функция
 ...
};


```

Когда объявление класса содержит чисто виртуальную функцию, нельзя создавать объект этого класса.

Идея состоит в том, что классы с чисто виртуальными функциями существуют исключительно для того, чтобы служить базовыми классами. Чтобы класс был подлинным абстрактным базовым классом, он должен иметь, по крайней мере, одну чисто виртуальную функцию.

Теперь можно получить классы **Ellipse** и **Circle** из класса **BaseEllipse**, добавляя элементы, необходимые для завершения каждого класса. Обратите внимание, что класс **Circle** всегда представляет окружности, в то время как класс **Ellipse** представляет эллипсы, которые также могут быть окружностями. Однако окружность класса **Ellipse** может повторно масштабироваться к другой геометрической фигуре, в то время как окружность класса **Circle** должна остаться окружностью.

Программа, использующая эти классы, будет способна создавать объекты **Ellipse** и **Circle**, но не объекты **BaseEllipse**. Поскольку объекты **Circle** и **Ellipse** имеют один и тот же базовый класс, совокупность таких объектов может управляться массивом указателей **BaseEllipse**.

Итак, ABC описывает интерфейс, использующий, по меньшей мере, одну чисто виртуальную функцию, а классы, производные из ABC, используют регулярные виртуальные функции для реализации интерфейса с учетом свойств конкретного производного класса.

## Обзор структуры класса

C++ может применяться к широкому кругу проблем программирования, и разработку класса нельзя свести к некоторой подпрограмме "представления посредством чисел". Однако существуют некоторые руководящие принципы, и сейчас самое время бегло ознакомиться с ними, подведя итоги и выделив наиболее важные моменты, освещенные ранее.

## Функции-элементы, которые генерирует компилятор

Как известно, компилятор автоматически генерирует некоторые общедоступные функции-элементы (см. главу 11), являющиеся очень важными. Давайте еще раз рассмотрим некоторых из них.

### Заданный по умолчанию конструктор

Заданный по умолчанию конструктор — это конструктор без аргументов или конструктор, для которого все аргументы являются заданными по умолчанию. Если вы не определяете какие-либо конструкторы, компилятор сам определяет заданный по умолчанию конструктор. Он не выполняет никаких функций, но он должен существовать, чтобы можно было выполнять определенные действия. Например, предположим, что **Star** — это класс.

Вы нуждаетесь в заданном по умолчанию конструкторе, чтобы выполнить следующее:

```
Star rigel; // создать объект без явной
 // инициализации
Star pleiades[6]; // создать массив
 // объектов
```

Предположим, что создается конструктор производного класса без явного вызова конструктора базового класса из списка инициализатора. При этом компилятор использует заданный по умолчанию конструктор базового класса, чтобы создать часть базового класса нового объекта.

При определении конструктора любого вида компилятор не будет определять заданный по умолчанию конструктор. В этом случае лучше самому обеспечить заданный по умолчанию конструктор, если он необходим.

Обратите внимание, что одна из мотиваций наличия конструкторов — необходимость гарантирования правильности инициализации объектов во всех ситуациях. Кроме того, если класс имеет любые элементы указателей, они должны быть инициализированы. Таким образом, имеет смысл обеспечить явный заданный по умолчанию конструктор, который инициализирует все элементы данных класса приемлемыми значениями.

### Конструктор копирования

Конструктор копирования — это конструктор, который в качестве своего аргумента принимает постоянную ссылку на тип класса. Например, конструктор копирования для класса `Star` имел бы следующий прототип:

```
Star(const Star &);
```

Конструктор копирования класса используется в следующих ситуациях:

- Когда новый объект инициализирован объектом того же самого класса
- Когда объект передан функции по значению
- Когда функция возвращает объект по значению
- Когда компилятор генерирует временный объект

Если программа не использует конструктор копирования (явный или неявный), компилятор обеспечивает прототип, но не определение функции. В противном случае программа определяет конструктор копирования, который выполняет инициализацию с учетом элементов. Иначе говоря, каждый элемент нового объекта инициализируется значением соответствующего элемента первоначального объекта.

В некоторых случаях инициализация с учетом элементов нежелательна. Например, указатели элементов, инициализированные оператором `new`, вообще требуют, чтобы было установлено глубокое копирование, как имеет место с классом `BankAccountD`. Или же класс может иметь статическую переменную, которая должна

изменяться. В таких случаях нужно определить свой собственный конструктор копирования.

### Оператор присваивания

Заданный по умолчанию оператор присваивания выполняет присваивание одного объекта другому объекту того же самого класса. Не путайте присваивание с инициализацией. Если оператор создает новый объект, он выполняет инициализацию, а если он изменяет значение существующего объекта, это — присваивание:

```
Star sirus;
Star alpha = sirus; // инициализация
 // (одна из записей)
Star dogstar;
dogstar = sirus; // присваивание
```

Если нужно явно определить конструктор копирования, то по тем же самым причинам нужно явно определить оператор присваивания. Прототип для оператора присваивания класса `Star` следующий:

```
Star & Star::operator=(const Star &);
```

Обратите внимание, что функция оператора присваивания возвращает ссылку на объект `Star`. Класс `BankAccountD` представляет типичный пример функции явного оператора присваивания.

Компилятор не генерирует операторы присваивания для присваивания одного типа другому. Предположим, что нужно иметь возможность присваивать строку объекту `Star`. Один из подходов — явное определение такого оператора:

```
Star & Star::operator=(const char *) { ... }
```

Второй подход заключается в использовании функции преобразования для преобразования строки в объект `Star` и использовании функции присваивания `Star-Star`. Первый подход выполняется более быстро, но требует большого объема кода. Подход с использованием функции преобразования может вести к ситуациям "одурманивания" компилятора.

### Другие соображения по поводу методов класса

Существует еще несколько моментов, которые нужно иметь в виду при определении класса. В следующих разделах перечислены некоторые из них.

### Конструкторы

Конструкторы отличаются от других методов класса тем, что они создают новые объекты, в то время как другие методы вызываются существующими объектами.

### Деструкторы

Не забудьте определить явный деструктор, который удаляет любую память, распределенную оператором `new` в

конструкторах класса, и заботится о любых других специальных действиях по наведению порядка, требуемых для разрушения объекта класса. Если класс должен использоваться в качестве базового, делайте деструктор виртуальным.

## Преобразования

Любой конструктор, который может вызываться только с одним аргументом, определяет преобразование от типа аргумента к типу класса. Например, рассмотрите следующие прототипы конструктора для класса Star:

Конструкторы преобразования применяются, например, когда преобразуемый тип передается функции, определенной в качестве принимающей аргумент класса. Например, предположите, что имеется следующее:

```
Star north;
north = "polaris";
```

Второй оператор вызвал бы функцию `Star::operator=(const Star &)`, использующую `Star::Star(const char *)` для генерирования объекта `Star`, который нужно использовать в качестве аргумента для функции оператора присваивания. Это предполагает, что оператор присваивания (`char`) \* — на — `Star` не был определен.

При использовании ключевого слова `explicit` в прототипе для конструктора с одним аргументом запрещаются неявные преобразования, но все же разрешаются явные преобразования:

```
class Star
{
...
public:
 explicit Star(const char *);
...
};

Star north;
north = "polaris"; // не допускается
north = Star ("polaris"); // допускается
```

Для преобразования объекта класса в какой-либо другой тип необходимо определить функцию преобразования (см. главу 10). Функция преобразования — это функция-элемент класса без аргументов или объявленный возвращаемый тип с именем типа, в который должно быть выполнено преобразование. Несмотря на отсутствие какого-либо объявленного возвращаемого типа, функция должна возвратить желательное значение преобразования. Вот несколько примеров:

```
Star::Star double() { ... } // преобразует
 // star в double
Star::Star const char * () { ... }
 // преобразует в const char
```

Необходимо соблюдать осторожность с такими функциями, используя их, только если для этого существует веская причина. В случаях с некоторыми структурами классов наличие функций преобразования увеличивает вероятность создания неоднозначного кода. Например, предположим, что было определено двойное преобразование для типа `vector` из главы 10 и при этом имелся следующий код:

```
vector ius(6.0, 0.0);
vector lux = ius + 20.2; // неоднозначно
```

Компилятор мог бы преобразовать тип `iws` в `double` и использовать добавление элемента типа `double` или же преобразовать `20.2` в `vector` (используя один из конструкторов) и применить добавление элемента `vector`. Вместо этого он не выполняется ни то, ни другое, а сообщается о неоднозначности конструкции.

## *Передача объекта по значению и передача по ссылке*

В общем случае, если вы создаете функцию, используя аргумент объекта, нужно передавать объект посредством ссылки, а не значения. Одна из причин этого — большая степень эффективности в первом случае. Передача объекта по значению включает генерирование временной копии, что означает вызов конструктора копирования, а затем позже — вызов деструктора. Вызов этих функций требует времени, и копирование большого объекта может осуществляться намного медленнее, чем передача ссылки. Если функция не изменяет объект, аргумент следует объявлять как ссылку **const**.

Существует и другая причина для передачи объектов посредством ссылки. В случае выполнения наследования, при котором используются виртуальные функции, функция, определенная как принимающая аргумент ссылки базового класса, может также успешно использовать с производными классами, как было показано ранее в этой главе. (Обратитесь к рассмотрению виртуальных методов далее в этой главе.)

## Возврат объекта и ссылки

Некоторые методы класса возвращают объекты. Читатели, вероятно, заметили, что одни из этих элементов возвращают объекты непосредственно, в то время как другие возвращают ссылки. Иногда метод должен возвратить объект, но, если это не обязательно, можно использовать ссылку. Давайте рассмотрим этот момент подробнее.

Прежде всего, единственное различие в коде между непосредственным возвратом объекта и возвратом ссылки заключается в прототипе функции и заголовке:

```
Star nova1(const Star &); // возвращает
 // объект Star
Star & nova2(const Star &); // возвращает ссылку на Star
```

Причина, по которой следует возвратить ссылку, а не объект, состоит в том, что возврат объекта требует генерирования временной копии возвращенного объекта. Именно копия доступна вызывающей программе. Таким образом, возврат объекта требует дополнительных затрат времени на вызов конструктора копирования для генерирования копии и на вызов деструктора для избавления от копии. Возврат ссылки позволяет экономить время и используемый объем памяти. Непосредственный возврат объекта аналогичен передаче объекта по значению: оба процесса генерируют временные копии. А возврат ссылки аналогичен передаче объекта посредством ссылки: и вызывающая и вызываемая функции действуют по отношению к тому же самому объекту.

Однако не всегда можно возвратить ссылку. Функция не должна возвращать ссылку на временный объект, созданный в функции, поскольку ссылка становится недопустимой, когда функция завершается и объект исчезает. В этом случае код должен возвратить объект, чтобы сгенерировать копию, которая будет доступна вызывающей программе.

Следует запомнить важное правило: если функция возвращает временный объект, созданный в функции, не следует использовать ссылку. Например, следующий метод использует конструктор для создания нового объекта, а затем возвращает копию этого объекта:

```
Vector Vector::operator+(const Vector & b)
{
 const
 {
 return Vector(x + b.x, y + b.y);
 }
}
```

Если функция возвращает объект, который был передан ей посредством ссылки или указателя, следует возвращать объект посредством ссылки. Например, следующий код возвращает посредством ссылки любой объект, который вызывает функцию, в противном же случае объект передается в качестве аргумента:

```
const Stock & Stock::topval(const Stock &
 s) const
{
 if (s.total_val > total_val)
 return s; // объект аргумента
 else
 return *this; // вызывающий
 // объект
}
```

## Использование `const`

Помните о возможностях использования `const`. Ее можно использовать, чтобы гарантировать, что метод не изменяет аргумент:

```
Star::Star(const char * s) {...} // не
 // будет изменяться строка,
 // на которую указывает
```

`const` можно использовать, чтобы гарантировать, что метод не изменит вызывающий его объект:

```
void Star::show() const {...} // не
 // будет изменять вызывающий объект
```

Здесь `const` означает `const Star * this`, где `this` указывает на вызывающий объект.

Обычно функция, которая возвращает ссылку, может находиться в левой части оператора присваивания. Это означает, что объекту, на который осуществляется ссылка, можно присвоить значение. Но `const` можно использовать, чтобы гарантировать, что ссылка или возвращаемое значение указателя не может использоваться для изменения данных в объекте:

```
const Stock & Stock::topval(const Stock &
 s) const
{
 if (s.total_val > total_val)
 return s; // объект аргумента
 else
 return *this; // вызывающий
 // объект
}
```

Здесь метод возвращает ссылку либо на `this`, либо на `s`. Поскольку `this` и `s` объявляются как `const`, функции не разрешается изменять их, и, следовательно, возвращаемая ссылка также должна быть объявлена как `const`.

Обратите внимание, что, когда функция объявляет аргумент в качестве ссылки или указателя на `const`, она не может передавать этот аргумент другой функции, если та функция не гарантирует, что не изменит аргумент.

## Соображения по поводу общедоступного наследования

Естественно, добавление наследования к программе требует учета множества факторов. Давайте рассмотрим некоторые из них.

### Отношение `is-a`

Руководствуйтесь отношением `is-a`. Если предложенный вами производный класс — не конкретный вид базового класса, не используйте общедоступное произведение. Например, не следует получать класс `Brain` из класса `Programmer`. Используйте объект класса `Brain` только в качестве элемента класса `Programmer`.

В некоторых случаях наилучшим подходом может быть создание абстрактного класса данных с чисто виртуальными функциями и получение других классов из него.

Помните, что одно из выражений отношения `is-a` — то, что указатель базового класса может указывать на объект производного класса и что ссылка базового класса может ссылаться на объект производного класса без яв-

ного приведения типа. Помните также, что обратное утверждение несправедливо. Таким образом, нельзя иметь указатель или ссылку производного класса, которая ссылается на объект базового класса без явного приведения типа. В зависимости от объявлений класса такое явное приведение типа (приведение вниз) либо может иметь смысл, либо нет. (Возможно, следует обратиться к рис. 12.4.)

### Какие объекты не наследуются

Конструкторы не наследуются. Однако конструкторы производного класса обычно используют синтаксис списка инициализатора для обращения к конструкторам базового класса для создания части базового класса производного объекта. Если конструктор производного класса явно не вызывает базовый конструктор при использовании синтаксиса списка инициализатора, он использует заданный по умолчанию конструктор базового класса. В цепочке наследований каждый класс может использовать список инициализатора для возврата информации своему непосредственному базовому классу.

Деструкторы тоже не наследуются. Но когда объект разрушается, программа вначале вызывает производный деструктор, а затем базовый деструктор. Если имеется заданный по умолчанию деструктор базового класса, компилятор генерирует заданный по умолчанию деструктор производного класса. Вообще, если класс служит в качестве базового, деструктор должен быть виртуальным.

Не наследуется оператор присваивания. Он имеет интересные свойства, которые мы рассмотрим далее.

### Оператор присваивания

Компилятор автоматически обеспечивает каждый класс оператором присваивания для присвоения одного объекта другому объекту того же самого класса. Заданная по умолчанию версия этого оператора выполняет присваивание с учетом элементов, когда каждому элементу целевого объекта присваивается значение соответствующего элемента исходного объекта. Однако, если объект принадлежит производному классу, компилятор использует оператор присваивания базового класса в целях обработки присваивания для части базового класса производного объекта. Если вы явно обеспечили использование оператора присваивания для базового класса, то будет использоваться этот оператор. Аналогично, если класс содержит элемент, который является объектом другого класса, оператор присваивания для этого класса используется для этого элемента.

Как уже было сказано неоднократно, необходимо обеспечить использование явного оператора присваива-

ния, если конструкторы класса используют оператор `new` для инициализации указателей. Поскольку C++ использует оператор присваивания базового класса для основной части производных объектов, не нужно переопределять оператор присваивания для производного класса, если только он не добавляет элементы данных, которые требуют особой заботы. Например, класс `BankAccountD` определил присваивание явно, но производный класс `Overdraft` использует заданный по умолчанию оператор присваивания, сгенерированный для этого класса.

Предположите, что производный класс использует `new` и что нужно обеспечить использование явного оператора присваивания, причем для каждого элемента класса, а не только для новых членов. Класс `OverdraftD` иллюстрирует, как это может быть выполнено:

```
OverdraftD & OverdraftD::operator=(const
 OverdraftD & od)
{
 if (this == &od)
 return *this;
 BankAccountD::operator=(od);
 // присваивание базового класса
 delete [] codeName;
 codeName =
 new char[strlen(od.codeName) + 1];
 strcpy(codeName, od.codeName);
 maxLoan = od.maxLoan;
 owesBank = od.owesBank;
 rate = od.rate;
 return *this;
}
```

А что можно сказать относительно присваивания производного объекта базовому объекту? (Примечание. Это не то же самое, что инициализация ссылки базового класса на производный объект.)

```
BankAccountD blips; // базовый класс
OverdraftD snips("Ranele Posh", "Topper",
 222333, 3993); // производный класс
blips = snips; // присваивание
// производного объекта базовому объекту
```

Какой оператор присваивания используется? Не забудьте, что оператор присваивания преобразуется в метод, вызываемый объектом, расположенным слева:

```
blips.operator=(snips);
```

Здесь таким объектом является объект `BankAccountD`, поэтому он вызывает функцию `BankAccountD::operator= (const BankAccountD &)`. Отношение *is-a* позволяет ссылке `BankAccountD` ссылаться на объект производного класса, такой как `snips`. Оператор присваивания имеет дело только с элементами базового класса, поэтому элемент `codeName` и другие элементы `OverdraftD` объекта `snips` игнорируются в присваивании. Таким образом, производный объект

можно присваивать базовому объекту, и при этом используются только элементы базового класса.

А можно ли присваивать объект базового класса производному объекту?

```
BankAccount gp("Griff Parker", 21234,
 1200); // базовый класс
Overdraft temp; // производный класс
temp = gp; // это возможно?
```

Здесь оператор присваивания был бы преобразован следующим образом:

```
temp.operator=(gp);
```

Слева расположен объект **Overdraft**, поэтому он вызывает функцию **Overdraft::operator=(const Overdraft &)**. Однако ссылка производного класса не может автоматически ссылаться на базовый объект, поэтому данный код *не будет* выполняться без наличия конструктора преобразования:

```
Overdraft(const BankAccount &);
```

(В данном случае имеется конструктор с дополнительными аргументами, причем они имеют значения, заданные по умолчанию.) В этом случае программа будет использовать этот конструктор для создания временного объекта **Overdraft** из **gp**, который затем будет использоваться как аргумент для оператора присваивания.

Можно было бы также определить оператор присваивания для присваивания базового класса производному классу:

```
Overdraft & Overdraft ::operator=(const
 BankAccount &) {...}
```

Здесь типы точно соответствуют оператору присваивания, и никакие преобразования типов не нужны.

### **Приватные и защищенные элементы**

Вспомните, что защищенные элементы подобны общедоступным элементам, если речь идет о производном классе, в остальных случаях они подобны приватным элементам. Производный класс может обращаться к защищенным элементам базового класса непосредственно, а к приватным элементам — только посредством функций-элементов базового класса. Таким образом, при создании элементов базового класса в качестве приватных обеспечивается более высокая степень безопасности, а при создание их в качестве защищенных упрощается кодирование и ускоряется доступ. Страуструп считает, что лучше использовать приватные элементы данных, а не защищенные, но при этом он убежден, что защищенные методы полезны. (Bjarne Stroustrup, *The Design and Evolution of C++*. Reading, MA: Addison-Wesley Publishing Company, 1994.)

### **Виртуальные методы**

При разработке базового класса необходимо решить, делать ли методы класса виртуальными или нет. Если нужно, чтобы производный класс был способным переопределять метод, определите метод как виртуальный в базовом классе. В результате будет активизировано позднее, или динамическое, связывание. Если переопределение метода не требуется, не делайте его виртуальным. Это не мешает кому-либо переопределить метод, но должно интерпретироваться как указание того, что переопределение метода нежелательно.

Обратите внимание, что несоответствующий код может обходить динамическое связывание. Рассмотрите, например, следующие две функции:

```
void show(const BankAccount & rba)
{
 rba.ViewAcct();
 cout << endl;
}

void sloppy(BankAccount ba)
{
 ba.ViewAcct();
 cout << endl;
}
```

Первая функция передает объект посредством ссылки, а вторая — по значению.

Теперь предположим, что каждая из них используется с аргументом производного класса:

```
Overdraft buzz("Buzz Parsec", 00001111,
 4300);
show(buzz);
sloppy(buzz);
```

Вызов функции **show()** приводит к тому, что аргумент **rba** оказывается ссылкой на объект **buzz** класса **Overdraft**, поэтому **rba.ViewAcct()** интерпретируется как версия **Overdraft**, как и должно быть. Но в функции **sloppy()**, которая передает объект по значению, **ba** — это объект **BankAccount**, созданный конструктором **BankAccount(const BankAccount &)**. (Автоматическое приведение вверх позволяет аргументу конструктора ссылаться на объект **Overdraft**.) Таким образом, в функции **sloppy()** **ba.ViewAcct()** является версией **BankAccount**, и поэтому отображается только компонент **BankAccount** объекта **buzz**.

### **Деструкторы**

Как уже говорилось, деструктор базового класса должен быть виртуальным. Таким образом, при удалении производного объекта посредством указателя или ссылки базового класса на объект программы использует деструктор производного класса, за которым следует деструктор базового класса, а не один деструктор базового класса.

## Итоговый анализ функций класса

Функции класса C++ имеют много разновидностей. Одни из них могут быть унаследованы, другие — нет. Одни функции операторов могут быть как функциями-элементами, так и дружественными функциями, в то время как другие могут быть только функциями-элементами. В табл. 12.1, основывающейся на подобной таблице из ARM (*Annotated Reference Manual* — Аннотированное справочное руководство), обобщены эти свойства. В ней запись `opr=` означает, что операторы присваивания записаны в форме `+=`, `*=` и т.д. Обратите внимание, что свойства для операторов `opr=` не отличаются от свойств категории "прочие операторы". Причина для перечисления операторов `opr=` отдельно состоит в том, чтобы указать, что эти операторы ведут себя иначе, чем оператор `=`.

## Резюме

Наследование позволяет адаптировать программный код к конкретным потребностям, определяя новый класс (производный) из существующего класса (базового). Общедоступное наследование моделируется отношением *is-a*, означающее, что объект производного класса также должен быть своего рода объектом базового класса. Как часть модели *is-a*, производный класс наследует элементы данных и большинство методов базового класса. Однако производный класс не наследует конструкторы базового класса, деструктор и оператор присваивания.

Производный класс может обращаться к общедоступным и защищенным элементам базового класса непосредственно и к частным членам базового класса с помощью общедоступных и защищенных методов базового класса. Затем можно добавлять новые элементы данных и методы к классу, а производный класс можно использовать в качестве базового класса для дальнейшей разработки. Каждый производный класс требует собственных конструкторов. Когда программа создает объект производного класса, она сначала вызывает конструктор базового класса, а затем конструктор производного класса. Когда программа удаляет объект, она сначала вызывает деструктор производного класса, а затем базового.

Если предполагается, что класс будет базовым, можно выбрать использование защищенных элементов вместо частных элементов, чтобы производные классы могли обращаться к этим элементам непосредственно. Однако в общем случае использование приватных элементов позволит снизить вероятность ошибок программирования. Если нужно, чтобы производный класс мог переопределять метод базового класса, сделайте его виртуальной функцией, объявив с ключевым словом `virtual`. В результате этого объекты, к которым обращаются указатели или ссылки, могут быть обработаны в соответствии с типом объекта, а не ссылки или указателя. В частности, деструктор для базового класса должен быть виртуальным.

Таблица 12.1 Свойства функций-элементов.

| Функция                      | Наследуется | Элемент или<br>дружественная<br>конструкция | Генерируется<br>по умолчанию | Может<br>быть<br>виртуальной | Может иметь<br>возвращаемый тип |
|------------------------------|-------------|---------------------------------------------|------------------------------|------------------------------|---------------------------------|
| Конструктор                  | Нет         | Элемент                                     | Да                           | Нет                          | Нет                             |
| Деконструктор                | Нет         | Элемент                                     | Да                           | Да                           | Нет                             |
| <code>=</code>               | Нет         | Элемент                                     | Да                           | Да                           | Да                              |
| <code>&amp;</code>           | Да          | Любое                                       | Да                           | Да                           | Да                              |
| Преобразование               | Да          | Элемент                                     | Нет                          | Да                           | Нет                             |
| <code>()</code>              | Да          | Элемент                                     | Нет                          | Да                           | Да                              |
| <code>[]</code>              | Да          | Элемент                                     | Нет                          | Да                           | Да                              |
| <code>-&gt;</code>           | Да          | Элемент                                     | Нет                          | Да                           | Да                              |
| <code>opr=</code>            | Да          | Любое                                       | Нет                          | Да                           | Да                              |
| <code>new</code>             | Да          | Статический элемент                         | Нет                          | Нет                          | <code>void*</code>              |
| <code>delete</code>          | Да          | Статический элемент                         | Нет                          | Нет                          | <code>void</code>               |
| Прочие операции              | Да          | Любое                                       | Нет                          | Да                           | Да                              |
| Прочие элементы              | Да          | Элемент                                     | Нет                          | Да                           | Да                              |
| Дружественные<br>конструкции | Нет         | Дружественная<br>конструкция                | Нет                          | Нет                          | Да                              |

Может потребоваться определить ABC (абстрактный базовый класс), который определяет интерфейс. При этом можно не вдаваться в вопросы реализации. Например, можно было бы определить абстрактный класс **Shape**, производными от которого будут конкретные классы формы, такие как **Circle** и **Square**. Абстрактный базовый класс должен содержать, по крайней мере, один чисто виртуальный метод. Чисто виртуальный метод можно объявить, помещая символы = 0 перед заключительной точкой с запятой объявления.

```
virtual double area() const = 0;
```

Программист не определяет чисто виртуальные методы и не может создавать объект класса, содержащий чисто виртуальные элементы. Вместо этого они служат для определения общего интерфейса, подлежащего использованию производными классами.

## Вопросы для повторения

- Что производный класс наследует из базового класса?
- Что производный класс не наследует из базового класса?
- Предположим, что в качестве возвращаемого типа для функции **BankAccountD::operator()** был бы определен **BankAccountD**, а не **BankAccountD &**. Как бы это сказалось на программе?
- В каком порядке вызываются конструкторы и деструкторы класса при создании и удалении объекта производного класса?
- Требуются ли конструкторы производному классу, если он не добавляет к базовому классу элементы данных?
- Предположим, что и базовый, и производный классы определяют метод с одним и тем же именем и объект производного класса вызывает этот метод. Какой метод вызывается?
- Когда производный класс должен определять оператор присваивания?
- Можно ли присваивать адрес объекта производного класса указателю на базовый класс? Можно ли присваивать адрес объекта базового класса указателю на производный класс?
- Можно ли присваивать объект производного класса объекту базового класса? Можно ли присваивать объект базового класса объекту производного класса?
- Предположим, что вы определяете функцию, которая в качестве аргумента принимает ссылку на

объект базового класса. Почему эта функция может также использовать в качестве аргумента объект производного класса?

- Предположим, что определяется функция, которая в качестве аргумента принимает объект базового класса (т.е. функция передает объект базового класса по значению). Почему эта функция может также использовать в качестве аргумента объект производного класса?
- Почему лучше передавать объект по ссылке, а не по значению?
- Предположите, что **Corporation** — базовый класс, а **PublicCorporation** — производный. Предположите также, что каждый класс определяет функцию-элемент **head()**, что **ph** — указатель на тип **Corporation** и что **ph** присвоен адрес объекта **PublicCorporation**. Как интерпретируется **ph->head()**, если базовый класс определяет **head()** как:
  - Регулярную функцию
  - Виртуальную функцию
- Что неправильно в следующем коде?

```
class Kitchen
{
private:
 double kit_sq_ft;
public:
 Kitchen() {kit_sq_ft = 0.0; }
 virtual double area()
 { return kit_sq_ft * kit_sq_ft; }
};

class House : public Kitchen
{
private:
 double all_sq_ft;
public:
 House() {all_sq_ft += kit_sq_ft;}
 double area(const char *s)
 { cout << s; return all_sq_ft; }
};
```

## Упражнения по программированию

- Начните со следующего объявления класса:

```
// базовый класс
class Cd { // представляет компакт-диск
private:
 char performers[50];
 char label[20];
 int selections;// количество
 // выборок
 double playtime; // время
 // воспроизведения в минутах
public:
 Cd(char * s1, char * s2, int n,
 double x);
```

```

Cd(const Cd & d);
Cd();
~Cd();
void Report() const; // сообщает все
 // данные компакт-диска
Cd & operator=(const Cd & d);
};

}

```

Получите производный класс **Classic**, добавляющий массив элементов **char**, которые будут содержать строку, идентифицирующую первоочередную задачу на компакт-диске. Если базовый класс требует, чтобы любые функции были виртуальными, соответствующим образом измените объявление базового класса. Проверьте свою программу с помощью следующей программы:

```

#include <iostream>
using namespace std;
#include "classic.h" // который будет
 // содержать cd.h
void Bravo(const Cd & disk);
int main()
{
 Cd c1("Beatles", "Capitol", 14, 35.5);
 Classic c2 = Classic("Piano Sonata in
 B flat, Fantasia in C",
 "Alfred Brendel", "Philips", 2,
 57.17);
 Cd *pcd = &c1;

 cout << "Using object directly:\n";
 c1.Report(); // использует метод Cd
 c2.Report(); // использует метод
 // Classic

 cout << "Using type Cd * pointer to
 objects:\n";
 pcd->Report(); // использует метод
 // Cd для объекта cd
 pcd = &c2;
 pcd->Report(); // использует метод
 // Classic для объекта classic

 cout << "Calling a function with a Cd
reference argument:\n";
 Bravo(c1);
 Bravo(c2);

 cout << "Testing assignment: ";
 Classic copy;
 copy = c2;
 copy.Report()

 return 0;
}

void bravo(const Cd & disk)
{
 disk.Report();
}

```

2. Повторите упражнение 1, но на сей раз для различных строк, отслеживаемых обоими классами. Вместо массивов фиксированного размера используйте метод динамического распределения памяти.

3. Измените иерархию классов **BankAccount-Overdraft** так, чтобы оба класса были производными из абстрактного базового класса. Проверьте результат с помощью программы, подобной приведенной в листинге 12.8.

4. Орден Benevolent Order of Programmers собирает коллекцию разлитого в бутылки портвейна. Для ее описания ответственный за разлив ВОР создал класс **Port**, объявленный как показано ниже:

```

#include <iostream>
using namespace std;
class Port
{
private:
 char * brand;
 char style[20]; // т.е., желтовато-
 // коричневый, рубиновый, год
 // изготовления вина
 int bottles;
public:
 Port(const char * br = "none", const
 char * st = "none",
 eint b = 0);
 Port(const Port & p); // конструктор
 // копии
 virtual ~Port() { delete [] brand; }
 Port & operator=(const Port & p);
 Port & operator+=(int b);
 // добавляет b к bottles
 Port & operator-=(int b); // вычитает
 // b из bottles, если возможно
 int BottleCount() const
 {
 return bottles;
 }
 virtual void Show() const;
 friend ostream & operator<<(ostream &
 os, const Port & p);
};

```

Метод **Show()** представляет информацию в следующем формате:

```

Brand: Gallo
Kind: tawny
Bottles: 20

```

Функция **operator<<()** представляет информацию в следующем формате (без символа новой строки в конце строки):

```

Gallo, tawny, 20

```

Ответственный за разлив завершил определения методов для класса **Port**. Затем он (прежде чем был уволен за случайную отправку кому-то бутылки Cockburn 45-го года изготовления для экспериментального соуса барбекю) получил производный класс **VintagePort** следующим образом:

```

class VintagePort : public Port
 // используется стиль "vintage"
{
private:
 char * nickname; // т.е.
 // " Благородный " или
 // " Старый Бархат " и т.п.
 int year; // год изготовления вина
public:
 VintagePort();
 VintagePort(const char * br, int b,
 const char * nn, int y);
 VintagePort(const VintagePort & vp);
 ~VintagePort() { delete [] nickname; }
 VintagePort & operator=(const
 VintagePort & vp);
 void Show() const;
 friend ostream & operator<<(ostream &
 os, const VintagePort & vp);
};

```

Вам придется завершить создание класса `VintagePort`.

- Первоочередная задача состоит в том, чтобы воссоздать определения методов класса `Port`, поскольку упомянутый ответственный за разлив уничтожил свои определения в отместку за увольнение.
- Вторая задача состоит в том, чтобы объяснить, почему одни методы переопределены, а другие — нет.
- Третья задача — объяснить, почему функции `operator=()` и `operator<<()` не являются виртуальными.
- И наконец, четвертая задача — обеспечить определения для методов класса `VintagePort`.

# Повторное использование программного кода в C++

**В этой главе рассматривается следующее:**

- Отношения *has-a*
- Классы с объектами-элементами (включение)
- Частное и защищенное наследование
- Создание шаблонов классов
- Использование шаблонов классов
- Специализации шаблонов
- Множественное наследование
- Виртуальные базовые классы

Одна из основных целей C++ состоит в том, чтобы упростить повторное использование кода. Эту задачу позволяет решить метод публичного наследования. Существуют и другие механизмы достижения этой цели, которые будут рассмотрены в этой главе. В частности, будет описана технология использования элементов класса, которые сами являются объектами другого класса. Этот метод называется *включением, компоновкой* или *послойным представлением*. Еще одна возможность — использование методов приватного или защищенного наследования. Методы включения, приватного и защищенного наследования обычно используются для реализации отношения *has-a* (*"содержит объект"*), т.е. отношения, при котором новый класс содержит объект другого класса. Например, класс *Stereo* мог бы содержать объект *CdPlayer*. Множественное наследование позволяет создавать классы, которые наследуются из двух или более базовых классов, объединяя их функциональные возможности.

В главе 9 были представлены шаблоны функций. Теперь мы рассмотрим шаблоны классов, которые открывают еще один путь для многократного использования кода. Шаблоны классов позволяют определять класс в общих чертах. Затем шаблон можно использовать для создания конкретных классов, определенных для конкретных типов. Например, можно было бы определить

общий шаблон стека и затем использовать его, чтобы создать один класс, представляющий стек значений *int*, и другой класс, представляющий стек значений *double*. Можно было бы даже сгенерировать класс, представляющий стек стеков.

## Классы, включающие элементы объектов

Давайте начнем рассмотрение с классов, которые в качестве элементов включают объекты класса. Некоторые классы, такие как *String* из главы 11 или стандартные классы и шаблоны C++ из главы 15, обеспечивают удобные способы представления компонентов более обширного класса. Рассмотрим конкретный пример.

Кто такой студент? Некто посещающий учебное заведение? Некто занятый умственными исследованиями? Беглец от жестокой действительности реального мира или лицо, имеющее определенное имя и количество заработанных баллов? Понятно, что последнее определение — совершенно неподходящая характеристика человека, но оно хорошо подходит для простого компьютерного представления. Поэтому давайте разработаем класс *Student*, основанный на этом определении.

Упрощенное представление студента именем и количеством полученных баллов предполагает использование

объекта класса `String` (см. главу 11) для хранения имени и объекта класса массива (который вскоре будет определен) для хранения значений баллов (предполагается, что они будут иметь тип `double`). (Познакомившись с классами библиотек в главе 15, вы, вероятно, пожелаете использовать стандартные классы `string` и `vector`.) Может возникнуть искушение получить класс `Student` путем выполнения публичного наследования из этих двух классов. Это было бы примером множественного публичного наследования, допускаемого языком C++, но в данном случае это было бы неправильно. Причина в том, что отношение студента к этим классам не соответствует модели *is-a* ("является объектом"). Студент — это не имя. И это не массив баллов. В данном случае имеет место отношение *has-a*. Студент имеет имя и обладает массивом баллов. Обычная методика C++ для моделирования отношения *has-a* заключается в использовании компоновки или включения; т.е. в создании класса, содержащего элементы, которые являются объектами другого класса. Например, объявление класса `Student` можно начать так:

```
class Student
{
private:
 //использование объекта String для имени
 String name;
 //использование объекта ArrayDb
 //для хранения баллов
 ArrayDb scores;
 ...
};
```

Как обычно, класс придает элементам данных приватный характер. Это подразумевает, что функции-элементы класса `Student` могут использовать публичные интерфейсы классов `String` и `ArrayDb` (сокращение от слов *array of double* — массив значений типа `double`) для получения доступа и изменения объектов `name` и `scores`, но внешние функции не могут это выполнять. Внешний мир сможет получать доступ к объектам `name` и `scores` только посредством общедоступного интерфейса, определенного для класса `Student` (рис. 13.1). Описывая этот процесс, обычно говорят, что класс `Student` получает реализацию своих объектов, но не наследует интерфейс. Например, для хранения имени объект `Student` использует реализацию `String`, а не реализацию `char * name` или `char name[26]`. Но объект `Student` не обладает изначально способностью использовать функцию `operator==()` объекта `String`.

### ИНТЕРФЕЙСЫ И РЕАЛИЗАЦИИ

При использовании метода публичного наследования класс наследует интерфейс и, возможно, реализацию. (Чисто виртуальные функции в базовом классе обеспечивают интерфейс без реализации.) Получение интерфейса — это

часть отношения *is-a*. С другой стороны, при использовании компоновки класс приобретает реализацию без интерфейса. Отсутствие наследования интерфейса — это свойство отношения *has-a*.

То, что объект класса не приобретает автоматически интерфейс содержащегося в нем объекта, — удобное свойство для отношения *has-a*. Например, можно было расширить класс `String` для перегрузки оператора `+`, чтобы разрешить конкатенацию двух строк, но концептуально не имеет смысла объединять два объекта `Student`. Это одна из причин, по которой не следует использовать метод публичного наследования в данном случае. С другой стороны, части интерфейса для содержащегося класса могут иметь смысл для нового класса. Например, может потребоваться использовать метод `operator<()` из интерфейса `String` для сортировки объектов `Student` по имени. Это можно сделать, определяя функцию-элемент `Student::Operator<()`, которая внутренне использует функцию `String::Operator<()`. Давайте перейдем к рассмотрению некоторых деталей реализации.

### Класс ArrayDb

Прежде всего, класс `ArrayDb` должен быть разработан так, чтобы класс `Student` мог его использовать. Этот класс будет в значительной степени подобен классу `String`, поскольку последний также является массивом, в данном случае массивом переменных типа `char`. Сначала давайте перечислим некоторые необходимые и/или желательные свойства для класса `ArrayDb`. Итак, этот класс должен отвечать следующим требованиям:

- Сохранять несколько значений типа `double`.
- Обеспечивать произвольный доступ к индивидуальным значениям посредством указания индекса в квадратных скобках.

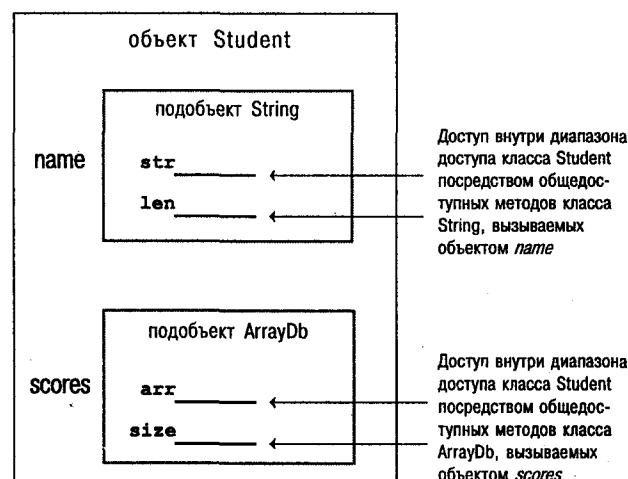


РИСУНОК 13.1 Включение.

- Позволять присваивать один массив другому.
- Выполнять проверку пределов, чтобы гарантировать корректность указанных индексов массива.

Первые два свойства присущи массиву. Третье свойство не является характерной особенностью встроенных массивов, но оно справедливо для объектов класса, поэтому создание класса массива обеспечит его реализацию. Последнее свойство не характерно для встроенных массивов, но оно может быть добавлено в качестве составной части второго свойства.

На данном этапе можно в значительной мере повторить объявление класса `String`. В этом случае можно выполнить следующее:

```
class ArrayDb
{
private:
 unsigned int size; //число элементов массива
 double * arr; //адрес первого элемента
public:
 ArrayDb(); //заданный по умолчанию конструктор
 //создание массива ArrayDb из n элементов,
 //установка каждого в значение val
 ArrayDb(unsigned int n, double val = 0.0);

 //создание массива ArrayDb из n элементов,
 //инициализация значениями массива pn
 ArrayDb(const double * pn, unsigned int n);

 ArrayDb(const ArrayDb & a); //конструктор
 //копирования
 virtual ~ArrayDb(); //деструктор
 //здесь должны помещаться другие компоненты,
 //подлежащие добавлению

 ArrayDb & operator=(const ArrayDb & a);
 friend ostream & operator<<(ostream & os,
 const ArrayDb & a);
};
```

Класс будет использовать метод динамического распределения памяти для создания массива желательного размера. Следовательно, он также будет поддерживать деструктор, конструктор копирования и оператор присваивания. Для удобства он будет включать еще несколько конструкторов.

Основное новое свойство — обеспечение произвольного доступа путем использования записи массива. Предположим, что имеется следующее объявление:

```
// пять элементов, значение каждого из
// которых установлено равным 20.0
ArrayDb scores(5, 20.0);
```

Если `scores` действительно подобен массиву, значит, можно выполнить следующее:

```
double temp = scores[3];
scores[3] = 16.5;
```

В этом случае требуется, чтобы общедоступное выражение `scores[3]` соответствовало частному представлению `scores.arr[3]`. Это можно легко выполнить, поскольку `[]` — всего лишь еще одна операция C++, подобная `<<`, и поэтому она может быть перегружена. Подобно `<<`, операция `[]` имеет два операнда. Основная особенность состоит в том, что один из operandов заключен в скобки. В любом случае выражение, подобное

```
scores[3]
```

объединяется со следующим вызовом функции перегруженного оператора:

```
scores.operator[](3)
```

Здесь имя функции — `operator[]`. Поскольку левый operand — объект класса `ArrayDb`, а правый operand имеет тип `int`, это соответствовало бы функции-элементу `ArrayDb::operator[](int)`.

Оператор

```
temp = scores[2];
```

соответствует следующему:

```
temp = scores.operator[](2);
```

Это означает, что функция должна возвратить значение `scores.arr[2]` или ссылку на этот элемент массива. Однако оператор, подобный

```
scores[2] = 19.0;
```

преобразуется в следующее выражение:

```
temp = scores.operator[](2) = 19.0;
```

Присваивание значения функции требует, чтобы функция возвращала ссылку, а не значение, поэтому это единственный доступный путь. Простейшей реализацией была бы следующая:

```
double & ArrayDb::operator[](int i)
{
 return arr[i];
}
```

В данном случае происходит преобразование `scores[i]` в `scores.arr[i]`, при этом разрешается общедоступный доступ к приватным внутренним элементам массива. Поскольку основное назначение массива — обеспечение произвольного доступа к его элементам, обеспечение публичного доступа к частным элементам массива весьма желательно. Как будет показано в следующем разделе, использование операции `[]` вместо обеспечения непосредственного доступа с помощью публичных элементов данных может гарантировать более высокий уровень защиты.

## Работа с operator[]( )

Перегруженная операция `operator[]` позволяет обращаться к отдельным элементам, но только посредством заданной операции. Это предоставляет возможность встраивать в программу некоторые проверки безопасности. В частности, метод может проверять, находится ли данный индекс массива в допустимых пределах значений, т.е. операцию можно задавать следующим образом:

```
double & ArrayDb::operator[](int i)
{
 // проверка индекса перед
 // продолжением выполнения
 if (i < 0 || i >= size)
 {
 cerr << "Error in array limits: "
 << i << " is a bad index\n";
 exit(1);
 }
 return arr[i];
}
```

Правда, при этом замедляется выполнение программы, так как требуется вычисление значения условного оператора при каждом обращении программы к элементу массива. Но зато повышается степень безопасности, поскольку предотвращается выход за границы заданного массива.

### Альтернатива с использованием спецификатора `const`

Предшествующее определение имеет слабое место. Давайте рассмотрим следующий код:

```
// пять элементов, значение каждого
// из которых установлено равным 0.4
const ArrayDb noChanges(5, 0.4);

cout << noChanges[2]; // недопустимо!
```

Проблема состоит в том, что `noChanges` имеет спецификатор `const`, а `operator[]( )` не гарантирует защиты от изменения значения, причем в этом случае нельзя вызывать метод, отличный от метода `const`, с объектом `const`. К счастью, существует простое решение. Можно перегрузить функции, сигнатуры которых идентичны, за исключением того, что одна функция использует ссылку или указатель `const`, а другая — нет. Иначе говоря, класс нуждается также в следующей функции:

```
const double & ArrayDb::operator[](int i)
{
 // проверка индекса перед продолжением
 if (i < 0 || i >= size)
 {
```

```
 cerr << "Error in array limits: "
 << i << " is a bad index\n";
 exit(1);
 }
 return arr[i];
}
```

Обратите внимание, что, поскольку возвращаемая ссылка имеет тип `const`, данную версию нельзя использовать для присвоения значения элементу массива. При этом можно реализовать доступ к массиву типа `const`. Этую версию можно использовать только для получения значения из элемента. Поэтому оператор

```
cout << noChanges[2]; //допускается для
 //объекта типа const
```

допустим, а следующий оператор — нет:

```
noChanges[2] = 300.4; //не допускается для
 //объекта типа const
```

Компилятор выберет версию `const` функции `operator[]( )` для использования с объектами типа `const` класса `ArrayDb` и будет использовать другую версию для объектов из класса `ArrayDb`, относящихся к типу `const`.

В листинге 13.1 приведен заголовочный файл для класса `ArrayDb`. Для удобства определение класса содержит метод `Average()`, который возвращает среднее значение элементов массива.

 **ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**  
Более ранние реализации не поддерживают ключевое слово `explicit`.

Интересным моментом является то, что один из конструкторов использует ключевое слово `explicit`:

```
explicit ArrayDb(unsigned int n,
 double val = 0.0);
```

Вспомните, что конструктор, который может быть вызван с одним аргументом, служит в качестве функции неявного преобразования типа аргумента в тип класса. В этом случае первый аргумент представляет количество элементов в массиве, а не значение для массива, поэтому наличие конструктора, служащего в качестве функции преобразования `Int-в-ArrayDb`, лишено смысла. Использование ключевого слова `explicit` отключает неявные преобразования. Если бы это ключевое слово было пропущено, то код, подобный следующему, был бы возможен:

```
// массив из 20 элементов, значение каждого
// из которых установлено равным 100
ArrayDb doh(20, 100);

// присвоение doh значений массива из пяти
// элементов, значение каждого из которых
// установлено равным 0
doh = 5;
```

**Листинг 13.1 Заголовочный файл arraydb.h.**

```

// arraydb.h - класс array
#ifndef _ARRAYDB_H_
#define _ARRAYDB_H_
#include <iostream>
using namespace std;

class ArrayDb
{
private:
 unsigned int size; // количество элементов массива
 double * arr; // адрес первого элемента
public:
 ArrayDb(); // заданный по умолчанию конструктор
 // создает массив ArrayDb из n элементов, значение каждого из которых установлено равным val
 explicit ArrayDb(unsigned int n, double val = 0.0);
 // создает массив ArrayDb из n элементов и инициализирует его значениями массива рп
 ArrayDb(const double * pn, unsigned int n);
 ArrayDb(const ArrayDb & a); // конструктор копирования
 virtual ~ArrayDb(); // деструктор
 unsigned int ArSize() const { return size; } // возвращает размер массива
 double Average() const; // усредненный возвращаемый массив

 // перегруженные операторы
 virtual double & operator[](int i); // индексация массива
 virtual const double & operator[](int i) const; // индексация массива
 ArrayDb & operator=(const ArrayDb & a);
 friend ostream & operator<<(ostream & os, const ArrayDb & a);
};

#endif

```

Здесь невнимательный программист ввел `doh` вместо `doh[0]`. Если бы конструктор был освобожден с помощью ключевого слова `explicit`, значение 5 преобразовывалось бы во временный объект `ArrayDb` при вызове конструктора `ArrayDb(5)` со значением, по умолчанию равным 0 и используемым для второго аргумента. Затем в результате присваивания исходный объект `doh` заменяется на временный. При использовании ключевого слова `explicit` компилятор будет воспринимать оператор присваивания как ошибку.

Описанная здесь реализация демонстрируется в листинге 13.2.

**Пример класса Student**

Теперь, имея класс `ArrayDb`, давайте приступим к созданию объявления класса `Student`. Конечно, этот класс должен содержать конструкторы и, по крайней мере, несколько функций, поддерживающих интерфейс для класса `Student`. Программа, представленная в листинге 13.3, делает это, определяя все встроенные конструкторы.

**Инициализация включенных объектов**

Обратите внимание, что все конструкторы используют уже знакомый синтаксис списка инициализации для инициализации и элементов объектов `name` и `scores`.

В некоторых ранее описанных случаях конструкторы использовали его для инициализации элементов, которые были встроенными типами:

```
// инициализирует qsize объектом qs
Queue::Queue(int qs) : qsize(qs) { ... }
```

Этот код использует имя элемента данных (`qsize`) в списке инициализации. Кроме того, конструкторы из предыдущих примеров использовали список инициализации для инициализации части базового класса производного объекта:

```
OverdraftD::OverdraftD(const OverdraftD & od) : BankAccountD(od) { ... }
```

Для унаследованных объектов конструкторы использовали имя класса в списке инициализации для вызова конкретного конструктора базового класса. Для объектов-элементов конструкторы используют имя элемента. Например, взгляните на последний конструктор в листинге 13.3:

```
Student(const char * str, const double * pd, int n) : name(str), scores(pd, n) { }
```

Поскольку он инициализирует объекты-элементы, а не унаследованные объекты, в списке инициализации используются имена элементов, а не имена классов.

## Листинг 13.2 Программа arraydb.cpp.

```

// arraydb.cpp - методы класса ArrayDb

#include <iostream>
using namespace std;
#include <cstdlib> // прототип exit()
#include "arraydb.h"

// заданный по умолчанию конструктор не содержит аргументов
ArrayDb::ArrayDb()
{
 arr = NULL;
 size = 0;
}

// формирует массив из n элементов, значение каждого из которых установлено равным val
ArrayDb::ArrayDb(unsigned int n, double val)
{
 arr = new double[n];
 size = n;
 for (int i = 0; i < size; i++)
 arr[i] = val;
}

// инициализирует объект ArrayDb не относящимся к классу массивом
ArrayDb::ArrayDb(const double *pn, unsigned int n)
{
 arr = new double[n];
 size = n;
 for (int i = 0; i < size; i++)
 arr[i] = pn[i];
}

// инициализирует объект ArrayDb другим объектом ArrayDb
ArrayDb::ArrayDb(const ArrayDb & a)
{
 size = a.size;
 arr = new double[size];
 for (int i = 0; i < size; i++)
 arr[i] = a.arr[i];
}

ArrayDb::~ArrayDb()
{
 delete [] arr;
}

double ArrayDb::Average() const
{
 double sum = 0;
 int i;
 int lim = ArSize();
 for (i = 0; i < lim; i++)
 sum += arr[i];
 if (i > 0)
 return sum / i;
 else
 {
 cerr << "No entries in score array\n";
 return 0;
 }
}

```

```
// позволяет пользователю обращаться к элементам посредством индекса (присваивание допускается)
double & ArrayDb::operator[](int i)
{
 // проверка индекса перед продолжением
 if (i < 0 || i >= size)
 {
 cerr << "Error in array limits: "
 << i << " is a bad index\n";
 exit(1);
 }
 return arr[i];
}

// позволяет пользователю обращаться к элементам посредством индекса (присваивание запрещено)
const double & ArrayDb::operator[](int i) const
{
 // проверка индекса перед продолжением
 if (i < 0 || i >= size)
 {
 cerr << "Error in array limits: "
 << i << " is a bad index\n";
 exit(1);
 }
 return arr[i];
}

// присваивание определения класса
ArrayDb & ArrayDb::operator=(const ArrayDb & a)
{
 if (this == &a) // если объект присвоен самому себе,
 return *this; // ничего не изменяйте
 delete [] arr;
 size = a.size;
 arr = new double[size];
 for (int i = 0; i < size; i++)
 arr[i] = a.arr[i];
 return *this;
}

// быстрый вывод, 5 значений в строке
ostream & operator<<(ostream & os, const ArrayDb & a)
{
 int i;
 for (i = 0; i < a.size; i++)
 {
 os << a.arr[i] << " ";
 if (i % 5 == 4)
 os << "\n";
 }
 if (i % 5 != 0)
 os << "\n";
 return os;
}
```

## Листинг 13.3 Класс student.h.

```

// student.h - определение класса Student с использованием включения

#ifndef _STUDNTC_H_
#define _STUDNTC_H_

#include <iostream>
using namespace std;
#include "arraydb.h"
#include "strng2.h" // из главы 11

class Student
{
private:
 String name;
 ArrayDb scores;
public:
 Student() : name("Null Student"), scores() { }
 Student(const String & s) : name(s), scores() { }
 Student(int n) : name("Nully"), scores(n) { }
 Student(const String & s, int n) : name(s), scores(n) { }
 Student(const String & s, const ArrayDb & a) : name(s), scores(a) { }
 Student(const char * str, const double * pd, int n) : name(str), scores(pd, n) { }
 ~Student() { }
 double & operator[](int i);
 const double & operator[](int i) const;
 double Average() const;

// дружественные конструкции
 friend ostream & operator<<(ostream & os, const Student & stu);
 friend istream & operator>>(istream & is, Student & stu);
};

#endif

```

Каждый элемент в этом списке инициализации вызывает соответствующий конструктор. Так, `name(str)` вызывает конструктор `String(const char *)`, а `scores(pd, n)` вызывает конструктор `ArrayDb (const double *, int)`.

А что происходит, если не использовать синтаксис списка инициализации? Как и в случае с унаследованными компонентами, C++ требует, чтобы все объекты-элементы были созданы прежде, чем будет создана остальная часть объекта. Так, если пропустить список инициализации, C++ использует заданные по умолчанию конструкторы, определенные для классов объектов-элементов.

### Использование интерфейса для включенного объекта

Интерфейс для включенного объекта не является общедоступным, но он может использоваться внутри методов класса. Например, вот как можно определить функцию, которая возвращает средний балл студента:

```

double Student::Average() const
{
 return scores.Average();
}

```

Это определяет функцию, которая может вызываться объектом `Student`. Внутренне он использует функцию `ArrayDb::Average()`. Дело в том, что `scores` — объект класса `ArrayDb`, поэтому он может вызывать функции-элементы класса `ArrayDb`.

Аналогично можете определить дружественную функцию, которая использует обе версии оператора `<<`: и `String`, и `ArrayDb`:

```

ostream & operator<<(ostream & os,
 const Student & stu)
{
 os << "Scores for " << stu.name << ":\n";
 os << stu.scores;
 return os;
}

```

Поскольку `stu.name` — объект `String`, он вызывает функцию `operator<<(ostream &, const String &)`. Точно так же объект `stu.scores` класса `ArrayDb` вызывает функцию `operator<<(ostream &, const ArrayDb &)`. Обратите внимание, что новая функция должна быть дружественной для класса `Student`, чтобы она могла обращаться к элементам `name` и `scores` объекта `Student`.

В листинге 13.4 показан файл методов класса для класса `Student`. Он включает методы, которые позволяют использовать оператор `[]` для обращения к отдельным значениям баллов в объекте `Student`.

**Листинг 13.4 Программа studentc.cpp.**

```

// studentc.cpp - класс Student,
// использующий включение
#include "studentc.h"

double Student::Average() const
{
 return scores.Average(); // использует
 // ArrayDb::Average()
}

double & Student::operator[](int i)
{
 return scores[i]; // использует
 // ArrayDb::operator[]()
}

const double & Student::operator[](int i) const
{
 return scores[i];
}

// дружественные функции

// использует версию String из ArrayDb
ostream & operator<<(ostream & os,
 const Student & stu)
{
 os << "Scores for " << stu.name << ":\n";
 os << stu.scores;
 return os;
}

// использует версию String
istream & operator>>(istream & is, Student & stu)
{
 is >> stu.name;
 return is;
}

```

При этом не потребовался большой объем нового кода. Использование метода включения позволяет применить программный код, который уже был создан.

**Использование нового класса**

Давайте сконструируем небольшую программу, чтобы проверить новый класс. Для простоты в ней будет использоваться массив, состоящий только из трех объектов **Student**, каждый из которых хранит пять значений полученных баллов. И в ней будет использоваться бесхитростный цикл ввода, который не проверяет результаты ввода и не позволяет прервать процесс ввода. Тестовая программа представлена в листинге 13.5. Скомпилируйте ее вместе с программами **studentc.cpp**, **strng2.cpp** и **arrayDb.cpp**.

**Листинг 13.5 Программа use\_stuc.cpp.**

```

// use_stuc.cpp - использует композитный класс
// Компиляция вместе с studentc.cpp,
// strng2.cpp, arraydb.cpp
#include <iostream>
using namespace std;
#include "studentc.h"

void set(Student & sa, int n);
const int pupils = 3;
const int quizzes = 5;

int main()
{
 Student ada[pupils] =
 { quizzes, quizzes, quizzes };

 int i;
 for (i = 0; i < pupils; i++)
 set(ada[i], quizzes);
 for (i = 0; i < pupils; i++)
 {
 cout << "\n" << ada[i];
 cout << "average: "
 << ada[i].Average() << "\n";
 }
 return 0;
}

void set(Student & sa, int n)
{
 cout << "Please enter the student's name: ";
 cin >> sa;
 cout << "Please enter " << n
 << " quiz scores:\n";
 for (int i = 0; i < n; i++)
 cin >> sa[i];
 while (cin.get() != '\n')
 continue;
}

```

Ниже приведен пример выполнения программы:

```

Please enter the student's name: Gil Bayts
Please enter 5 quiz scores:
92 94 96 93 95
Please enter the student's name: Pat Roone
Please enter 5 quiz scores:
83 89 72 78 95
Please enter the student's name: Fleur O'Day
Please enter 5 quiz scores:
92 89 96 78 64
Scores for Gil Bayts:
92 94 96 93 95
average: 94
Scores for Pat Roone:
83 89 72 78 95
average: 83.4
Scores for Fleur O'Day
92 89 96 78 64
average: 83.8

```

## Приватное наследование

C++ имеет второе средство реализации отношения *has-a* — *приватное наследование*. При использовании метода приватного наследования общедоступные и защищенные элементы базового класса становятся приватными элементами производного класса. Это означает, что методы базового класса не становятся частью общедоступного интерфейса производного объекта. Однако они могут использоваться внутри функций-элементов производного класса.

Давайте подробнее рассмотрим тему интерфейса. При использовании метода общедоступного наследования общедоступные методы базового класса становятся общедоступными методами производного класса. Одним словом, производный класс наследует интерфейс базового класса. Это часть отношения *is-a*. При использовании метода приватного наследования общедоступные методы базового класса становятся приватными методами производного класса. Иначе говоря, производный класс не наследует интерфейс базового класса. Как было показано на примере включенных объектов, это отсутствие наследования — часть отношения *has-a*.

При использовании метода приватного наследования класс наследует реализацию. Это значит, что, если класс **Student** основывается на классе **String**, класс **Student** унаследует компонент класса **String**, который может использоваться для хранения строки. Более того, методы **Student** могут внутренне использовать методы **String** для получения доступа к компоненту **String**.

Включение позволяет добавить объект к классу в качестве именованного объекта-элемента, в то время как в результате приватного наследования добавляется объект к классу в качестве неименованного унаследованного объекта. Для обозначения объекта, добавленного путем наследования или помещения, мы будем использовать термин *подобъект*.

Приватное наследование обеспечивает те же самые свойства, что и включение: выполнение реализации, но не внедрение интерфейса. Следовательно, оно также может использоваться для реализации отношения *has-a*. Давайте посмотрим, как можно использовать метод приватного наследования для изменения класса **Student**.

### Пример класса **Student** (новая версия)

Для реализации приватного наследования необходимо при определении класса использовать ключевое слово **private** вместо **public**. (Фактически **private** — это значение, принятное по умолчанию, поэтому пропуск спецификатора доступа также ведет к частному наследованию.) Класс **Student** должен наследоваться из двух классов, поэтому в объявлении будут перечислены оба:

```
class Student : private String,
 private ArrayDb
{
public:
 ...
};
```

При наличии больше чем одного базового класса используется *множественное наследование*, или **MI**. В общем случае множественное наследование, особенно общедоступное, может приводить к проблемам, которые приходится решать путем применения дополнительных синтаксических правил. Мы рассмотрим эти вопросы далее в этой главе. Но в данном конкретном случае **MI** не вызывает никаких проблем.

Обратите внимание, что новый класс не будет нуждаться в приватном разделе. Дело в том, что два унаследованных базовых класса уже обеспечивают все необходимые элементы данных. Включение поддерживает использование двух явно именованных объекта в качестве элементов. Однако приватное наследование обеспечивает использование двух неименованных подобъектов в качестве унаследованных элементов. Это первое из основных различий в двух подходах.

### Инициализация компонентов базового класса

Наличие неявно унаследованных компонентов вместо объектов-элементов повлияет на кодирование, поскольку больше нельзя будет использовать **name** и **scores** для описания объектов. Вместо этого придется вновь прибегнуть к методикам, которые использовались для общедоступного наследования. Например, рассмотрите конструкторы. При реализации включения используется следующий конструктор:

```
// использует имена объектов для включения
Student(const char * str, const double * pd,
 int n) : name(str), scores(pd, n) { }
```

В новой версии для унаследованных классов будет использоваться синтаксис инициализатора, в котором для идентификации конструкторов используется имя класса вместо имени элемента:

```
// использует имена классов для наследования
Student(const char * str, const double * pd,
 int n) : String(str), ArrayDb(pd, n) { }
```

Таким образом, в списке инициализатора используются термины, подобные **String(str)**, вместо **name(str)**. Это второе основное различие при реализации двух подходов.

В листинге 13.6 показано новое объявление класса. Единственные изменения — удаление явных имен объектов и использование имен классов вместо имен элементов во встроенных конструкторах.

**Листинг 13.6 Класс studenti.h.**

```
//studenti.h - определение класса Student с
//использованием метода приватного наследования

#ifndef _STUDENTI_H_
#define _STUDENTI_H_

#include <iostream>
using namespace std;
#include "arraydb.h"
#include "strng2.h"

class Student : private String, private ArrayDb
{
public:
 Student() : String("Null Student"), ArrayDb() { }
 Student(const String & s)
 : String(s), ArrayDb() { }
 Student(int n) : String("Nully"), ArrayDb(n) { }
 Student(const String & s, int n)
 : String(s), ArrayDb(n) { }
 Student(const String & s, const ArrayDb & a)
 : String(s), ArrayDb(a) { }
 Student(const char * str, const double * pd,
 int n) : String(str), ArrayDb(pd, n) { }
 ~Student() { }
 double & operator[](int i);
 const double & operator[](int i) const;
 double Average() const;
//дружественные конструкции
 friend ostream & operator<<(ostream & os,
 const Student & stu);
 friend istream & operator>>(istream & is,
 Student & stu);
};

#endif
```

**Использование методов базового класса**

Приватное наследование ограничивает использование методов базового класса. Это ограничение состоит в том, что их можно использовать только внутри методов производного класса. Однако иногда может потребоваться, чтобы средство базового класса стало общедоступным. Например, в результате выполнения объявления класса появляется возможность работы с функцией `Average()`. Как и при включении, методика, применяемая в этом случае, заключается в использовании приватной функции `ArrayDb::Average()` внутри общедоступной функции `Student::average()` (рис. 13.2). При включении вызывается метод с объектом:

```
double Student::Average() const
{
 return scores.Average();
 // использует ArrayDb::Average()
}
```

При этом наследование позволяет использовать имя класса и оператор определения диапазона доступа для вызова функции базового класса:

```
double Student::Average() const
{
 return ArrayDb::Average();
}
```

Пропуск спецификатора `ArrayDb::` заставил бы компилятор интерпретировать вызов функции `Average()` как `Student::Average()`, приводя к крайне нежелательному рекурсивному определению функции. Одним словом, при методе включения для вызова метода используются имена объектов, в то время как приватное наследование обуславливает использование имени класса и оператора определения диапазона доступа.

Подобная методика явного определения имени функции с помощью имени ее класса не срабатывает для дружественных функций, потому что они не принадлежат к классу. Однако для вызова нужных функций можно выполнять явное приведение типа к базовому классу. Например, давайте рассмотрим следующее определение дружественной функции:

```
ostream & operator<<(ostream & os,
 const Student & stu)
{
 os << "Scores for "
 << (const String &) stu << ":\n";
 os << (const ArrayDb &) stu;
 return os;
}
```

Если `plato` — объект класса `Student`, то оператор

```
cout << plato;
```

вызовет эту функцию вместе с ссылкой `stu`, являющейся ссылкой на `plato`, и `os`, являющейся ссылкой на `cout`. В составе программного кода приведение типа

```
os << "Scores for "
 << (const String &) stu << ":\n";
```

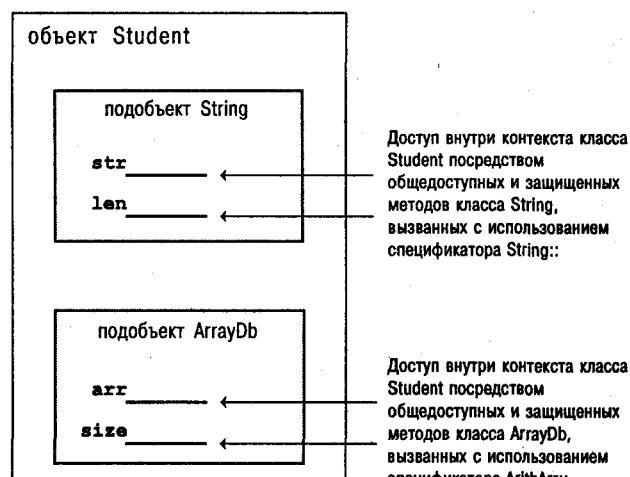


РИСУНОК 13.2 Приватное наследование.

позволяет явно преобразовать `stu` в ссылку на объект типа `String`, и это соответствует функции `operator<<(ostream &, const String &)`. Точно так же в результате приведения типа

```
os << (const ArrayDb &) stu;
```

осуществляется вызов функции `operator<<(ostream &, const ArrayDb &)`.

Ссылка `stu` не преобразовывается автоматически в ссылку `String` или `ArrayDb`. Основная причина этого заключается в следующем: при использовании метода приватного наследования ссылке или указателю на базовый класс не может быть присвоена ссылка или указатель на производный класс без явного приведения типа.

Однако, даже если бы в примере использовался метод общедоступного наследования, пришлось бы выполнять явные приведения типа. Одна из причин этого заключается в том, что без приведения типа код, подобный

```
os << stu;
```

соответствовал бы прототипу дружественной функции, приводя к рекурсивному обращению. Вторая причина заключается в том, что, поскольку класс использует метод множественного наследования, компилятор не смог бы определить, в какой базовый класс выполняется преобразование в данном случае, так как оба возможные преобразования соответствуют существующим функциям `operator<<()`.

В листинге 13.7 показаны все методы класса, отличающиеся от определенных внутри объявлений класса.

#### Листинг 13.7 Программа `studenti.cpp`.

```
// studenti.cpp - класс Student, использующий
// метод приватного наследования
#include "studenti.h"
double Student::Average() const
{
 return ArrayDb::Average();
}
double & Student::operator[](int i)
{
 return ArrayDb::operator[](i);
}
const double & Student::operator[](int i) const
{
 return ArrayDb::operator[](i);
}
// дружественные функции
ostream & operator<<(ostream & os,
 const Student & stu)
{
 os << "Scores for "
 << (const String &) stu << ":\n";
 os << (const ArrayDb &) stu;
 return os;
}
```

```
istream & operator>>(istream & is, Student & stu)
{
 is >> (String &) stu;
 return is;
}
```

#### Использование измененного класса `Student`

Снова пора проверить новый класс. Обратите внимание, что обе версии класса `Student` имеют абсолютно одинаковый общедоступный интерфейс, поэтому его можно проверить с помощью одной и той же программы. Единственное различие в том, что нужно включить `studenti.h` вместо `studentc.h` и что программу нужно компоновать со `studenti.cpp`, а не со `studentc.cpp`. Программа приведена в листинге 13.8. Скомпилируйте ее вместе с программами `studenti.cpp`, `string2.cpp` и `arrayDb.cpp`.

#### Листинг 13.8 Программа `use_stui.cpp`.

```
// use_stui.cpp - использует класс с частным
// произведением
// Компилируйте вместе с studenti.cpp,
// string2.cpp, arraydb.cpp
#include <iostream>
using namespace std;
#include "studenti.h"
void set(Student & sa, int n);
const int pupils = 3;
const int quizzes = 5;
int main()
{
 Student ada[pupils] =
 { quizzes, quizzes, quizzes };
 int i;
 for (i = 0; i < pupils; i++)
 set(ada[i], quizzes);
 for (i = 0; i < pupils; i++)
 {
 cout << "\n" << ada[i];
 cout << "average: "
 << ada[i].Average() << "\n";
 }
 return 0;
}
void set(Student & sa, int n)
{
 cout << "Please enter the student's name: ";
 cin >> sa;
 cout << "Please enter " << n
 << " quiz scores:\n";
 for (int i = 0; i < n; i++)
 cin >> sa[i];
 while (cin.get() != '\n')
 continue;
}
```

Результаты выполнения программы:

```
Please enter the student's name: Gil Bayts
Please enter 5 quiz scores:
```

**92 94 96 93 95**

Please enter the student's name: Pat Roone

Please enter 5 quiz scores:

**83 89 72 78 95**

Please enter the student's name: Fleur O'Day

Please enter 5 quiz scores:

**92 89 96 78 64**

Scores for Gil Bayts:

**92 94 96 93 95**

average: 94

Scores for Pat Roone:

**83 89 72 78 95**

average: 83.4

Scores for Fleur O'Day

**92 89 96 78 64**

average: 83.8

Те же входные данные, что и прежде, ведут к такому же результату, что и прежде.

## Включение или приватное наследование?

Предположив, что отношение *has-a* можно моделировать и с помощью метода включения, и с помощью метода приватного наследования, какой из этих методов следует использовать? Большинство программистов, пишущих на языке C++ предпочтает включение. Во-первых, его проще отслеживать. При просмотре объявления класса вы видите явно именованные объекты, представляющие включенные классы, и код может обращаться к этим объектам по именам. Использование метода наследования придает отношению более абстрактный вид. Во-вторых, наследование может порождать проблемы, особенно если класс наследуется более чем из одного базового класса. В этом случае, вероятно, придется иметь дело с такими проблемами, как наличие методов с одним и тем же именем в отдельных базовых классах или наличие отдельных базовых классов, совместно использующих общий предок. В целом вероятность возникновения проблем ниже при использовании метода включения. Кроме того, включение позволяет помещать в программу более одного подобъекта того же самого класса. Если класс нуждается в трех объектах *String*, то, используя метод включения, можно объявить три отдельных элемента *String*. Наследование же позволяет использовать только одиничный объект. (Было бы трудно различить объекты, если бы они все были безымянными.)

Однако приватное наследование предоставляет такие свойства, которые не может обеспечить включение. Например, предположите, что класс имеет защищенные элементы, которые могут быть элементами данных или функциями-элементами. Такие элементы доступны производным классам, но не миру в целом. Если включить такой класс в состав другого класса путем использования метода компоновки, новый класс будет являться частью мира в целом, а не частью производного класса.

Следовательно, он не может обращаться к защищенным элементам. В то же время использование метода наследования делает новый класс производным классом, и поэтому он может обращаться к защищенным элементам.

Еще одна ситуация, предполагающая использование метода приватного наследования, связана с необходимостью переопределять виртуальные функции. И в этом случае такая привилегия предоставляется производному классу, но не классу. При использовании метода приватного наследования переопределенные функции были бы пригодны для использования только внутри класса, а не для общего доступа.



### СОВЕТ

В общем случае метод включения следует использовать для моделирования отношения *has-a*. Метод приватного наследования нужно использовать, если новый класс должен обращаться к защищенным элементам в исходном классе или если он должен переопределять виртуальные функции.

## Защищенное наследование

Защищенное наследование — разновидность приватного наследования. Этот метод использует ключевое слово **protected** при перечислении компонентов базового класса:

```
class Student : protected String,
 protected ArrayDb { ... };
```

При использовании метода защищенного наследования общедоступные и защищенные элементы базового класса становятся защищенными элементами производного класса. Как и в случае с приватным наследованием, интерфейс для базового класса доступен производному классу, но не внешнему миру. Основное различие между приватным и защищенным наследованием проявляется при получении другого класса из производного класса. При использовании метода приватного наследования этот класс третьего поколения не получает возможности внутреннего использования интерфейса базового класса. Дело в том, что общедоступные методы базового класса становятся приватными в производном классе, а к частным элементам и методам класса нельзя непосредственно обращаться со следующего уровня произведения. При использовании метода защищенного наследования общедоступные базовые методы становятся защищенными во втором поколении и поэтому доступны внутренне для следующего уровня произведения.

Общедоступное, приватное и защищенное наследование обобщены в табл. 13.1. Термин *неявное ворхнее приведение* означает, что может существовать указатель или ссылка базового класса, ссылающаяся на объект производного класса без выполнения явного приведения типа.

Таблица 13.1 Разновидности наследования.

| Свойство                          | Общедоступное наследование                                     | Защищенное наследование                                        | Приватное наследование                                         |
|-----------------------------------|----------------------------------------------------------------|----------------------------------------------------------------|----------------------------------------------------------------|
| Общедоступные элементы становятся | Общедоступными элементами производного класса                  | Защищенными элементами производного класса                     | Приватными элементами производного класса                      |
| Защищенные элементы становятся    | Защищенными элементами производного класса                     | Защищенными элементами производного класса                     | Приватными элементами производного класса                      |
| Приватные элементы становятся     | Доступными только при использовании интерфейса базового класса | Доступными только при использовании интерфейса базового класса | Доступными только при использовании интерфейса базового класса |
| Неявное верхнее приведение        | Да                                                             | Да (но только внутри производного класса)                      | Нет                                                            |

### Переопределение доступа с помощью объявления using

Общедоступные элементы базового класса становятся защищенными или приватными при защищенном или частном создании производного класса. В последнем примере было показано, как производный класс может делать доступным метод базового класса при наличии метода производного класса, использующего метод базового класса:

```
double Student::Average() const
{
 return ArrayDb::Average();
}
```

Существует альтернатива включению одного вызова функции в другой, которая состоит в использовании объявления `using` (подобного тем, которые использовались с областями имен) для объявления использования элементов базового класса производным классом, даже если произведение является приватным. Например, в `studenti.h` можно пропустить объявление для `Student::Average()` и заменить его объявлением `using`:

```
class Student : private String,
 private ArrayDb
{
public:
 using ArrayDb::Average;
 ...
};
```

Объявление `using` делает метод `ArrayDb::Average()` доступным, как если бы он был общедоступным методом класса `Student`. Это означает, что можно пропустить определение `Student::Average()` в программе `studenti.cpp` и все же использовать метод `Average()` как прежде:

```
cout << "average: " << ada[i].Average() << "\n";
```

Различие состоит в том, что теперь метод `ArrayDb::Average()` вызывается непосредственно, а не путем промежуточного вызова метода `Student::Average()`.

Обратите внимание, что в объявлении `using` используется только имя элемента — не используются ни круглые скобки, ни сигнатуры функций, ни возвращаемые типы. Например, чтобы сделать метод `operator[]()` класса `ArrayDB` доступным классу `Student`, нужно было бы поместить следующее объявление `using` в общедоступный раздел объявления класса `Student`:

```
using ArrayDb::operator[];
```

Это сделало бы доступными обе версии (`const` и отличную от `const`). Подход с применением объявления `using` применим только для наследования, но не для включения.

Существует более старый метод повторного объявления методов базового класса в приватном производном классе, который заключается в помещении имени метода в общедоступный раздел производного класса. Вот как это могло бы быть выполнено:

```
class Student : private String,
 private ArrayDb
{
public:
 //повторное объявление в качестве
 //частного с использованием только имени
 ArrayDb::operator[][];
 ...
};
```

Это напоминает объявление `using` без использования ключевого слова `using`. Однако этот подход *не слишком хорош*, поэтому постепенно перестает применяться. Поэтому, если компилятор поддерживает объявление `using`, используйте его, чтобы сделать метод из частного базового класса доступным производному классу.

### Шаблоны классов

Наследование (общедоступное, приватное или защищенное) и включение не всегда отвечают стремлению к повторному использованию кода. Например, давайте рассмотрим классы `Stack` (см. главу 9), `Queue` (см. главу 11)

и `ArrayDb`. Все они — примеры *классов контейнеров*, которые предназначены для хранения других объектов или типов данных. Например, класс `Stack` хранил значения `unsigned long`. Так же легко можно было бы определить класс `Stack` для хранения значений `double` или объектов `String`. За исключением типа сохраненного объекта, код был бы идентичным. Однако, вместо того чтобы создавать объявления нового класса, желательно было бы иметь возможность определять стек в общем виде (т.е. независимо от типа) и затем обеспечивать конкретный тип в качестве параметра для класса. Затем этот же общий код можно было бы использовать для создания стеков различных типов значений. В главе 9 в примере `Stack` в качестве первого шага навстречу этому стремлению был использован спецификатор `typedef`. Однако тот подход имеет несколько недостатков. Во-первых, при каждом изменении типа приходится редактировать файл заголовка. Во-вторых, эту методику можно использовать для генерирования только одного вида стека на программу. Иначе говоря, `typedef` не может представлять два различных типа одновременно, и поэтому метод нельзя использовать для определения стеков `ints` и `String` в одной программе.

Шаблоны классов C++ обеспечивают лучший способ генерирования общих объявлений классов. (Первоначально C++ не поддерживал шаблоны, и с момента их появления шаблоны продолжают развиваться; поэтому существует вероятность, что используемый вами компилятор поддерживает не все описанные здесь свойства.) Шаблоны обеспечивают *параметризованные* типы, т.е. возможность передачи имени типа в качестве аргумента предписания для создания класса или функции. Например, передавая имя типа `int` шаблону `Queue`, можно заставить компилятор создать класс `Queue` для формирования очереди объектов `int`.

Библиотека Standard Template Library (Стандартная библиотека шаблонов) (STL) C++, которая была частично рассмотрена в главе 15, обеспечивает мощные и гибкие реализации шаблонов нескольких классов контейнеров. А в этой главе будут исследованы более основополагающие особенности разработки.

## Определение шаблона класса

Давайте используем класс `Stack` из главы 9 в качестве основы для создания шаблона. Вот как выглядит объявление исходного класса:

```
typedef unsigned long Item;
class Stack
{
private:
 // константа, характерная для класса
 enum { MAX = 10 };
```

```
Item items[MAX]; //содержит элементы стека
int top; //индекс для верхнего элемента стека
public:
 Stack();
 bool isempty() const;
 bool isfull() const;
 // push() возвращает значение false,
 // если стек уже заполнен, и true —
 // в противном случае
 bool push(const Item & item); //добавление
 //элемента в стек
 // pop() возвращает значение false,
 // если стек уже пуст, и true —
 // в противном случае
 bool pop(Item & item); //выталкивает
 //верхний элемент
};
```

Подход с использованием шаблона заменит определение `Stack` определением шаблона, и функции-элементы `Stack` — функциями-элементами шаблона. Как и в случае с шаблонными функциями, класс шаблона необходимо предварять кодом следующей формы:

```
template <class Type>
```

Ключевое слово `template` сообщает компилятору, что вы собираетесь определять шаблон. Конструкция в угловых скобках аналогична списку аргументов функции. Ключевое слово `class` можно считать служащим в качестве имени типа переменной, которая принимает тип в качестве значения, а `Type` — представляющим имя этой переменной.

В данном случае использование ключевого слова `class` не означает, что `Type` должен быть классом; это означает лишь то, что `Type` служит в качестве общего спецификатора типа, который при использовании шаблона будет заменяться реальным типом. Более новые реализации позволяют в этом контексте вместо ключевого слова `class` использовать более понятное ключевое слово `typename`:

```
template <typename Type> //более новый вариант
```

В позиции `Type` можно использовать свой вариант общего имени типа; при указании имени действуют те же правила, что и при указании любого другого идентификатора. Наиболее часто используются `T` и `Type`; мы будем использовать последнее. При вызове шаблона `Type` будет заменяться конкретным значением типа, таким как `int` или `String`. Внутри определения шаблона общее имя типа используется для идентификации типа, который должен быть сохранен в стеке. Для случая `Stack` это означало бы использование `Type` везде, где старое объявление прежде использовало бы идентификатор `Item` конструкции `typedef`. Например,

```
Item items[MAX]; //содержит элементы стека
 становится следующим:
```

```
Type items[MAX]; //содержит элементы стека
```

Аналогично методы класса исходного класса можно заменить функциями-элементами шаблона. Заголовок каждой функции будет предварен тем же самым объявлением шаблона:

```
template <class Type>
```

И снова идентификатор Item конструкции `typedef` заменяется общим именем типа `Type`. Еще одно изменение заключается в том, что спецификатор класса должен быть изменен со `Stack::` на `Stack<Type>::`. Например,

```
bool Stack::push(const Item & item)
{
...
}
```

становится следующим:

```
template <class Type>
// или template <typename Type>
bool Stack<Type>::push(const Type & item)
{
...
}
```

Если определять метод внутри объявления класса (встроенное определение), можно опускать префикс шаблона и спецификатор класса.

В листинге 13.9 показаны объединенные шаблоны класса и функции-элемента. Важно понимать, что эти шаблоны не являются определениями класса и функции-элемента. Скорее, они служат инструкциями для компилятора C++ относительно генерирования определений класса и функции-элемента. Конкретная актуализация шаблона, такого как класс стека для обработки объектов `String`, называется *созданием экземпляра* или *специализацией*. Если только вы не располагаете компилятором, в котором реализовано новое ключевое слово `export`, размещение шаблонной функции-элемента в отдельном файле реализации будет невозможно. Поскольку шаблоны не являются функциями, они не могут компилироваться отдельно. Шаблоны должны использоваться совместно с запросами о конкретных экземплярах шаблонов. Проще всего можно добиться успеха, если поместить всю информацию шаблона в заголовочный файл и включить его в файл, который будет использовать шаблоны.

### Листинг 13.9 Класс stacktp.h.

---

```
// stacktp.h - шаблон стека
template <class Type>
class Stack
{
private:
 enum { MAX = 10 }; //константа,
 //характерная для класса
 Type items[MAX]; //содержит элементы стека
 int top; //индекс для верхнего
 //элемента стека
```

```
public:
 Stack();
 bool isempty();
 bool isfull();
 // добавляет элемент в стек
 bool push(const Type & item);
 // выталкивает верхнюю часть
 bool pop(Type & item);
}

template <class Type>
Stack<Type>::Stack()
{
 top = 0;
}

template <class Type>
bool Stack<Type>::isempty()
{
 return top == 0;
}

template <class Type>
bool Stack<Type>::isfull()
{
 return top == MAX;
}

template <class Type>
bool Stack<Type>::push(const Type & item)
{
 if (top < MAX)
 {
 items[top++] = item;
 return true;
 }
 else
 return false;
}

template <class Type>
bool Stack<Type>::pop(Type & item)
{
 if (top > 0)
 {
 item = items[-top];
 return true;
 }
 else
 return false;
}
```

Если компилятор имеет реализацию нового ключевого слова `export`, определения методов шаблона можно поместить в отдельный файл при условии, что каждое определение предваряется ключевым словом `export`:

```
export template <class Type>
Stack<Type>::Stack()
{
 top = 0;
}
```

Тогда можно было бы следовать тем же соглашениям, которые используются для обычных классов:

1. Чтобы сделать объявление доступным для программы, следует поместить объявление класса шаблона в заголовочный файл и использовать директиву `#include`.

2. Чтобы сделать объявления доступными для программы, следует поместить определения методов класса шаблона в файл исходного кода и использовать файл проекта или аналогичный ему.

## Использование класса шаблона

При простом включении шаблона в программу класс шаблона не генерируется. Необходимо создать экземпляр. Для этого объягите объект типа класса шаблона, заменив общее имя типа конкретным желаемым типом. Вот, например, как можно было бы создать два стека: один — для накопления объектов `int` и другой — для накопления объектов `String`:

```
Stack<int> kernels; //создает стек int
Stack<String> colonels; //создает стек
 //объектов String
```

Видя эти два объявления, компилятор будет следовать инструкциям, приведенным в шаблоне `Stack<Type>`, чтобы сгенерировать два отдельных объявления класса и два отдельных набора методов класса. Объявление класса `Stack<int>` полностью заменит `Type` на `int`, в то время как объявление класса `Stack<string>` полностью заменит `Type` на `String`. Конечно, используемые алгоритмы должны соответствовать типам. Например, класс стека предполагает, что можно присваивать один элемент другому. Это предположение справедливо для основных типов, структур и классов (если только оператор присваивания не сделан частным), но не для массивов.

Общие идентификаторы типа, такие как `Type` в вышеприведенном примере, называются параметрами типа, т.е. они действуют как переменная, но вместо числового значения им присваивается тип. Так, в вышеприведенном объявлении `kernel` параметр типа `Type` имеет значение `int`.

Обратите внимание, что требуемый тип нужно поддерживать явно. Это отличается от обычных шаблонов функций, для которых компилятор может использовать типы аргументов функции, чтобы выяснить, какую функцию следует генерировать:

```
Template <class T>
void simple(T t) { cout << t << '\n'; }
...
simple(2); // генерирует пустую
 // функцию simple(int)
simple("two") // генерирует пустую
 // функцию simple(char *)
```

В листинге 13.10 приведена измененная первоначальная программа проверки стека (листинг 9.13). Здесь используются идентификаторы порядка загрузки строк вместо значений `unsigned long`. Поскольку в нем используется наш класс `String`, компилируйте его вместе с программой `strng2.cpp`.

### Листинг 13.10 Программа stacktem.cpp.

```
//stacktem.cpp - проверяет класс шаблона стека
//Компилировать вместе с strng2.cpp

#include <iostream>
using namespace std;
#include <cctype>
#include "stacktp.h"
#include "strng2.h"

int main()
{
 Stack<String> st; //создает пустой стек
 char c;
 String po;
 cout << "Please enter A to add a
 purchase order,\n"
 << "P to process a PO, or Q to quit.\n";
 while (cin >> c && toupper(c) != 'Q')
 {
 while (cin.get() != '\n')
 continue;
 if (!isalpha(c))
 {
 cout << '\a';
 continue;
 }
 switch(c)
 {
 case 'A':
 case 'a': cout << "Enter a PO number
 to add: ";
 cin >> po;
 if (st.isfull())
 cout << "stack already full\n";
 else
 st.push(po);
 break;
 case 'P':
 case 'p': if (stisempty())
 cout << "stack already empty\n";
 else {
 st.pop(po);
 cout << "PO # " << po
 << " popped\n";
 }
 break;
 }
 cout << "Please enter A to add a
 purchase order,\n"
 << "P to process a PO, or Q
 to quit.\n";
 }
 cout << "Bye\n";
 return 0;
}
```

## ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Используйте более старый заголовочный файл `cstype.h`, если применяемая реализация не поддерживает `cctype`.

Результаты выполнения программы:

```
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: red911porsche
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: green328bmw
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: silver747boing
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #silver747boing popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #green328bmw popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #red911porsche popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
stack already empty
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
Q
Bye
```

## Более подробное рассмотрение шаблона класса

В качестве типа для шаблона класса `Stack <Type>` можно использовать встроенный тип или объект класса. А как насчет указателя? Например, можно ли в листинге 13.10 использовать указатель на тип `char` вместо объекта `String`? В конце концов, такие указатели представляют собой встроенный способ обработки строк C++. Ответ будет следующим: можно создать стек указателей, но он будет работать не очень хорошо без существенных модификаций в программе. Компилятор может создать класс, но именно программист должен обеспечить его эффективное использование. Давайте посмотрим, почему такой стек указателей работает не очень хорошо с программой из листинга 13.10, а затем рассмотрим пример ситуации, когда стек указателей полезен.

### Неправильное использование стека указателей

Мы кратко рассмотрим три простые, но показательные попытки адаптации листинга 13.10 к использованию стека

указателей. Эти попытки служат хорошей иллюстрацией того, что следует учитывать структуру шаблона, а не просто использовать его вслепую. Все три начинаются со следующего совершенно правильного вызова шаблона `Stack<Type>`:

```
Stack<char *> st; // создает стек для
// указателей-на-char
```

Затем в первой версии

```
String po;
```

заменяется на

```
char * po;
```

Идея состоит в том, чтобы для реализации ввода с клавиатуры использовать указатель `char` вместо объекта `String`. Но этот подход изначально можно назвать неудачным, поскольку при создании указателя не создается область для хранения вводимых строк.

Во второй версии

```
String po;
```

заменяется на

```
char po[40];
```

В результате выделяется пространство для входной строки. Кроме того, массив `po` имеет тип `char *` и поэтому может быть помещен в стек. Но массив принципиально не согласуется с предположениями, сделанными для метода `pop()`:

```
template <class Type>
bool Stack<Type>::pop(Type & item)
{
 if (top > 0)
 {
 item = items[--top];
 return true;
 }
 else
 return false;
}
```

Во-первых, переменная ссылки `item` должна ссылаться на какое-либо значение `Lvalue`, а не на имя массива. Во-вторых, программа заключает, что можно выполнять присваивания переменной `item`. Даже если бы переменная `item` могла ссылаться на массив, для имени массива нельзя выполнять присваивание. Поэтому и данный подход также неудачен.

В третьей версии

```
String po;
```

заменяется на

```
char * po = new char[40];
```

Выделяется пространство для входной строки. Кроме того, `po` является переменной, следовательно, она

совместима с кодом метода `pop()`. Однако в этом случае мы сталкиваемся с наиболее серьезной проблемой. Имеется только одна переменная `po`, и она всегда указывает на одну и ту же ячейку памяти. Действительно, содержимое памяти изменяется при считывании каждой новой строки, но при каждой операции записи в стек помещается один и тот же адрес. Таким образом, при очистке стека вы всегда получаете обратно тот же самый адрес, и он всегда относится к последней строке, считанной в память. В частности, стек не сохраняет каждую новую строку при ее вводе и не выполняет никакой полезной функции.

### Корректное использование стека указателей

Один из способов использования стека указателей состоит в том, чтобы вызывающая программа обеспечивала массив указателей, каждый из которых указывает на другую строку. Тогда помещение этих указателей в стек имело бы смысл, поскольку каждый указатель ссылался бы на другую строку. Обратите внимание, что ответственность за создание отдельных указателей лежит на вызывающей программе, а не на стеке. Задача стека — управлять указателями, а не создавать их.

Например, предположим, что нужно смоделировать следующую ситуацию. Кто-то доставил тележку папок Плодсону (Plodson). Если корзина для входящих документов Плодсона пуста, он снимает верхнюю папку с тележки и помещает ее в свою корзину для входящих документов. Если корзина для входящих документов заполнена, Плодсон изымает верхнюю папку из корзины, обрабатывает ее, и помещает в свою корзину для исходящих документов. Если корзина для входящих документов заполнена частично, Плодсон может просматривать верхнюю папку в корзине для входящих документов или взять следующую папку с тележки и поместить ее в корзину для входящих документов. В глубине души считая это несколько эксцентричным способом самовыражения, он подбрасывает монетку, чтобы решить, какое из этих действий предпринять. Давайте рассмотрим влияние его метода на первоначальный порядок папок.

Эту ситуацию можно смоделировать с помощью массива указателей на строки, представляющие папки на тележке. Каждая строка будет содержать имя человека, заданное папкой. Стек можно использовать в целях представления корзины для входящих документов, а второй массив указателей можно использовать в целях представления корзины для исходящих документов. Добавление папки в корзину для входящих документов имитируется "выталкиванием" указателя из входного массива в стек, а просмотр папки имитируется "выталкиванием" элемента из стека и добавлением его к корзине для исходящих документов.

Учитывая важность исследования всех аспектов этой проблемы, было бы полезно иметь возможность экспериментировать с различными размерами стека. В листинге 13.11 класс `Stack<Type>` слегка переопределен, чтобы конструктор `Stack` принимал необязательный аргумент размера. При этом включается внутреннее использование динамического массива, и поэтому теперь класс нуждается в деструкторе, конструкторе копирования и операторе присваивания. Кроме того, определение сокращает объем кода, придавая нескольким методам встроенный характер.

### Листинг 13.11 Класс `stcktp1.h`.

```
// stcktp1.h - измененный шаблон Stack
template <class Type>
class Stack
{
private:
 enum { MAX = 10 } ; //константа,
 //характерная для класса
 int stacksize;
 Type * items; //содержит элементы стека
 int top; //индекс для верхнего элемента стека
public:
 explicit Stack(int ss = MAX);
 Stack(const Stack & st);
 ~Stack() { delete [] items; }
 bool isempty() { return top == 0; }
 bool isfull() { return top == stacksize; }
 bool push(const Type & item); //добавляет
 //элемент в стек
 bool pop(Type & item); // "выталкивает"
 //верхнюю часть стека
 Stack & operator=(const Stack & st);
};

template <class Type>
Stack<Type>::Stack(int ss) : stacksize(ss), top(0)
{
 items = new Type [stacksize];
}

template <class Type>
Stack<Type>::Stack(const Stack & st)
{
 stacksize = st.stacksize;
 top = st.top;
 items = new Type [stacksize];
 for (int i = 0; i < top; i++)
 items[i] = st.items[i];
}

template <class Type>
bool Stack<Type>::push(const Type & item)
{
 if (top < stacksize)
 {
 items[top++] = item;
 return true;
 }
 else
 return false;
}
```

```

template <class Type>
bool Stack<Type>::pop(Type & item)
{
 if (top > 0)
 {
 item = items[-top];
 return true;
 }
 else
 return false;
}

template <class Type>
Stack<Type> & Stack<Type>::operator=(const
 Stack<Type> & st)
{
 if (this == &st)
 return *this;
 delete [] items;
 stacksize = st.stacksize;
 top = st.top;
 items = new Type [stacksize];
 for (int i = 0; i < top; i++)
 items[i] = st.items[i];
 return *this;
}

```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Некоторые реализации языка могут не распознавать ключевое слово **explicit**.

Обратите внимание, что прототип объявляет ссылку на **Stack** как возвращаемый тип для функции оператора присваивания, в то время как при определении фактической функции-шаблона идентифицируется тип как **Stack<Type>**. Первая запись является сокращенной формой второй записи, но она может использоваться только внутри диапазона доступа класса. Иначе говоря **Stack** можно использовать внутри объявления шаблона и внутри определений функций шаблона, но снаружи класса, как при идентификации возвращаемых типов и при использовании оператора определения, нужно использовать полную форму **Stack<Type>**.

В программе, представленной в листинге 13.12, новый шаблон стека используется для моделирования ситуации с Плодсоном. Функции **rand()**, **srand()** и **time()** используются таким же образом, как и в предыдущих моделях, в которых требовалось генерировать случайные числа. Случайное генерирование чисел 0 или 1 моделирует подбрасывание монеты.

### Листинг 13.12 Программа **stkptr1.cpp**.

```

// stkptr1.cpp - проверяет стек указателей
#include <iostream>
using namespace std;
#include <cstdlib> // для rand(), srand()
#include <ctime> // для time()
#include "stackptr1.h"
const int Stacksize = 4;
const int Num = 10;

```

```

int main()
{
 srand(time(0)); // генерирует случайные
 // значения rand()
 cout << "Please enter stack size: ";
 int stacksize;
 cin >> stacksize;
 Stack<char *> st(stacksize); // создает пустой
 // стек с четырьмя ячейками
 char * in[Num] = {
 " 1: Hank Gilgamesh", " 2: Kiki Ishtar",
 " 3: Betty Rocker", " 4: Ian Flagrant",
 " 5: Wolfgang Kibble", " 6: Portia Koop",
 " 7: Joy Almondo", " 8: Xaverie Paprika",
 " 9: Juan Moore", "10: Misha Mache"
 };
 char * out[Num];
 int processed = 0;
 int nextin = 0;
 while (processed < Num)
 {
 if (st.isempty())
 st.push(in[nextin++]);
 else if (st.isfull())
 st.pop(out[processed++]);
 else if (rand() % 2 && nextin < Num)
 // фифти-фифти
 st.push(in[nextin++]);
 else
 st.pop(out[processed++]);
 }
 for (int i = 0; i < Num; i++)
 cout << out[i] << "\n";
 cout << "Bye\n";
 return 0;
}

```

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Некоторые реализации требуют использования **stdlib.h** вместо **cstdlib** и **time.h** – **ctime**.

Ниже приведены два примера выполнения программы. Обратите внимание, что конечный порядок файлов может несколько отличаться от одного тестового запуска до другого, даже когда размер стека сохраняется неизменным.

```

Please enter stack size: 5
2: Kiki Ishtar
1: Hank Gilgamesh
3: Betty Rocker
5: Wolfgang Kibble
4: Ian Flagrant
7: Joy Almondo
9: Juan Moore
8: Xaverie Paprika
6: Portia Koop
10: Misha Mache
Bye

Please enter stack size: 5
3: Betty Rocker
5: Wolfgang Kibble

```

```

6: Portia Koop
4: Ian Flagranti
8: Xaverie Paprika
9: Juan Moore
10: Misha Mache
7: Joy Almondo
2: Kiki Ishtar
1: Hank Gilgamesh
Bye

```

## Примечания к программе

Сами строки никогда не перемещаются. При помещении строки в стек в действительности создается новый указатель на существующую строку, т.е. создается указатель, значение которого — адрес существующей строки. А при удалении строки из стека копируется значение адреса в массив `out`.

А какое влияние деструктор стека оказывает на строки? Никакого. Конструктор класса использует `new` в целях создания массива для хранения указателей. Деструктор класса уничтожает этот массив, а не строки, на которые указывали элементы массива.

## Шаблон массива и аргументы, не являющиеся типами

Шаблоны часто используются для классов контейнеров, поскольку идея параметров типа хорошо согласуется с потребностью применять общий план организации хранения к ряду типов. Действительно, стремление поддержки кода многократного использования для классов контейнеров было основной побуждающей причиной для введения шаблонов, поэтому давайте рассмотрим еще один пример, в котором исследуются еще несколько аспектов разработки и использования шаблонов. В частности, мы рассмотрим аргументы, не являющиеся типом, или аргументы-выражения и использование массива для работы с функциями, реализующими наследование.

Давайте начнем с шаблона простого массива, который позволяет указывать размер массива. Одна из методик, применяемая в последней версии шаблона `Stack`, предполагает использование динамического массива внутри класса и аргумента конструктора для определения количества элементов. При другом подходе используется аргумент шаблона для определения размера обычного массива. В листинге 13.13 показано, как это может быть выполнено.

### Листинг 13.13 Класс arraytp.h.

---

```
//arraytp.h - шаблон массива
#include <iostream>
using namespace std;
#include <cstdlib>
template <class T, int n>
```

```

class ArrayTP
{
private:
 T ar[n];
public:
 ArrayTP();
 explicit ArrayTP(const T & v);
 virtual T & operator[](int i);
 virtual const T & operator[](int i) const;
};

template <class T, int n>
ArrayTP<T,n>::ArrayTP()
{
 for (int i = 0; i < n; i++)
 ar[i] = 0;
}

template <class T, int n>
ArrayTP<T,n>::ArrayTP(const T & v)
{
 for (int i = 0; i < n; i++)
 ar[i] = v;
}

template <class T, int n>
T & ArrayTP<T,n>::operator[](int i)
{
 if (i < 0 || i >= n)
 {
 cerr << "Error in array limits: " << i
 << " is out of range\n";
 exit(1);
 }
 return ar[i];
}

template <class T, int n>
const T & ArrayTP<T,n>::operator[](int i) const
{
 if (i < 0 || i >= n)
 {
 cerr << "Error in array limits: " << i
 << " is out of range\n";
 exit(1);
 }
 return ar[i];
}

```

---

Обратите внимание на заголовок шаблона:

```
template <class T, int n>
```

Ключевое слово `class` определяет `T` как параметр или аргумент типа. Ключевое слово `int` определяет переменную `n` как имеющую тип `int`. Этот второй вид параметра, который указывает конкретный тип вместо того, чтобы действовать в качестве общего имени типа, называется *аргументом, не имеющим тип* или *аргументом-выражением*. Предположим, что имеется следующее объявление:

```
ArrayTP<double, 12> eggweights;
```

Это заставляет компилятор определить класс `ArrayTP<double, 12>` и создать объект `eggweights` этого класса. При определении класса компилятор заменяет `T` на `double` и `n` на `12`.

Аргументы-выражения имеют некоторые ограничения. Аргумент-выражение может быть интегральным типом, типом перечисления, ссылкой или указателем. Таким образом, аргумент `double m` исключается, но аргументы `double & rm` и `double * pm` допустимы. Кроме того, код шаблона не может изменять значение аргумента или принимать его адрес. Таким образом, в шаблоне `ArrayTP` такие выражения, как `n++` или `&n`, не допускаются. И еще, при создании экземпляра шаблона значение, используемое для аргумента-выражения, должны быть постоянным выражением.

Этот подход к определению размеров массива имеет одно преимущество по сравнению с подходом с применением конструктора, используемым в `Stack`. Подход с применением конструктора использует динамическую память, управляемую операторами `new` и `delete`, в то время как подход с применением аргумента-выражения использует стек памяти, поддерживаемый для автоматических переменных. Это обеспечивает более быстрое выполнение, особенно при наличии множества маленьких массивов.

Основной недостаток подхода с применением аргумента-выражения состоит в том, что каждое значение размера массива генерирует собственный шаблон. Иначе говоря, объявления

```
ArrayTP<double, 12> eggweights;
ArrayTP<double, 13> donuts;
```

генерируют два отдельных объявления класса. Но объявления

```
Stack<int> eggs(12);
Stack<int> dunkers(13);
```

генерирует только одно объявление класса, и информация о размере передается конструктору для этого класса.

Еще одно различие — то, что подход с применением конструктора является более универсальным, поскольку размер массива сохраняется в качестве элемента класса, а не жестко кодируется в определении. Это делает возможным, например, определять присваивание из массива одного размера в массив другого размера или создавать класс, который допускает массивы переменных размеров.

## Использование шаблона вместе с семейством классов

Допустим, что имеется семейство классов. В частности, предположим, что существует базовый класс наряду с несколькими производными классами. Теперь предположим, что существует несколько объектов этих классов, которые желательно сохранить в массиве. Одна из про-

блем состоит в том, что все элементы массива должны иметь тот же самый тип, но тип объекта производного класса отличается от типа объекта из базового класса или другого производного класса. Он может даже отличаться по размеру от объекта базового класса, и поэтому ввести компилятор в заблуждение относительно типа не получится. Таким образом, массив представляет инструмент, который подходит для однородных элементов (элементов одного и того же вида), но в случае с семейством класса приходится иметь дело с гетерогенной совокупностью объектов (объектов различных видов).

Именно в подобных ситуациях полезны отношения *is-a* и виртуальные функции. Как рассматривается в листинге 13.8, можно создать массив указателей базового класса. Общедоступное наследование определяет, что адрес любого объекта производного класса можно присвоить указателю базового класса, и поэтому такой массив может содержать адреса ряда типов. Размеры типов могут различаться, но все указатели имеют одинаковый размер. Более того, если эти указатели используются для вызова методов класса, виртуальные функции гарантируют, что указатель базового класса на производный объект вызовет методы класса производного объекта.

Давайте проверим это, используя шаблон `ArrayTP`. Прежде всего, вы нуждаетесь в семействе классов. Для наглядности давайте сохраним классы простыми. В листинге 13.14 определены классы `Worker`, `Waiter`, `Singer` и `Greeter`. Класс `Worker` включает имя и идентификационный номер. Класс `Waiter`, производный из `Worker`, добавляет оценку степени изящности. Класс `Singer`, производный из `Worker`, добавляет описатель вокального диапазона. И наконец, класс `Greeter`, производный из `Worker`, добавляет оценку искренности.

При такой организации каждый человек является рабочим какой-либо конкретной профессии (официантом, певцом, поздравителем), но не рабочим вообще. Таким образом, `Worker` может быть сделан абстрактным базовым классом. Вспомните, что это требует, по крайней мере, одной чистой виртуальной функции. Этот класс позволяет деструктору принять эту роль. (Поскольку `Worker` — базовый класс, он должно иметь виртуальный деструктор, даже если тот не выполняет никаких функций. Если `Worker` должен также быть абстрактным базовым классом, деструктор можно также сделать чистой виртуальной функцией.)

Существует одна особенность создания чистого деструктора. Обычно чисто виртуальная функция реализуется в производном классе, но деструктор должен быть реализован в собственном классе. Так, в случае наличия чисто виртуального деструктора, все же нужно обеспечить реализацию базового класса, что и будет сделано в листинге 13.15.

## Листинг 13.14 Класс worker.h.

```

// worker.h - рабочие классы
#include "strng2.h"

class Worker // абстрактный базовый класс
{
private:
 String fullname;
 long id;
public:
 Worker() : fullname("no one"), id(0L) { }
 Worker(const String & s, long n)
 : fullname(s), id(n) { }
 virtual ~Worker() = 0; // чисто виртуальный деструктор
 virtual void Set();
 virtual void Show() const;
};

class Waiter : public Worker
{
private:
 int panache;
public:
 Waiter() : Worker(), panache(0) { }
 Waiter(const String & s, long n, int p = 0)
 : Worker(s, n), panache(p) { }
 Waiter(const Worker & wk, int p = 0)
 : Worker(wk), panache(p) { }
 void Set();
 void Show() const;
};

class Singer : public Worker
{
protected:
 enum { other, alto, contralto, soprano,
 bass, baritone, tenor} ;
 enum { Vtypes = 7} ;
private:
 static char *pv[Vtypes]; // строковые эквиваленты типов голосов
 int voice;
public:
 Singer() : Worker(), voice(other) { }
 Singer(const String & s, long n, int v = other)
 : Worker(s, n), voice(v) { }
 Singer(const Worker & wk, int v = other)
 : Worker(wk), voice(v) { }
 void Set();
 void Show() const;
};

class Greeter : public Worker
{
private:
 int cheer;
public:
 Greeter() : Worker(), cheer(0) { }
 Greeter(const String & s, long n, int c = 0)
 : Worker(s, n), cheer(c) { }
 Greeter(const Worker & wk, int c = 0)
 : Worker(wk), cheer(c) { }
 void Set();
 void Show() const;
};

```

## Листинг 13.15 Программа worker.cpp.

```

// worker.cpp - методы рабочего класса
#include "worker.h"
#include <iostream>
using namespace std;
// методы класса Worker
// необходимо реализовать виртуальный деструктор, даже если он "чистый"
Worker::~Worker() {}
void Worker::Set()
{
 cout << "Enter worker's name: ";
 cin >> fullname;
 cout << "Enter worker's ID: ";
 cin >> id;
 while (cin.get() != '\n')
 continue;
}
void Worker::Show() const
{
 cout << "Name: " << fullname << "\n";
 cout << "Employee ID: " << id << "\n";
}
// методы класса Waiter
void Waiter::Set()
{
 Worker::Set();
 cout << "Enter waiter's panache rating: ";
 cin >> panache;
 while (cin.get() != '\n')
 continue;
}
void Waiter::Show() const
{
 cout << "Category: waiter\n";
 Worker::Show();
 cout << "Panache rating: " << panache << "\n";
}
// методы класса Singer
char * Singer::pv[] = { "other", "alto", "contralto", "soprano", "bass", "baritone", "tenor" };
void Singer::Set()
{
 Worker::Set();
 cout << "Enter number for singer's vocal range:\n";
 int i;
 for (i = 0; i < Vtypes; i++)
 {
 cout << i << ": " << pv[i] << " ";
 if (i % 4 == 3)
 cout << '\n';
 }
 if (i % 4 != 0)
 cout << '\n';
 cin >> voice;
 while (cin.get() != '\n')
 continue;
}
void Singer::Show() const
{
 cout << "Category: singer\n";
 Worker::Show();
 cout << "Vocal range: " << pv[voice] << "\n";
}
// методы класса Greeter
void Greeter::Set()
{
 Worker::Set();
 cout << "Enter greeter's cheerfulness rating: ";
 cin >> cheer;
 while (cin.get() != '\n')
 continue;
}
void Greeter::Show() const
{
 cout << "Category: greeter\n";
 Worker::Show();
 cout << "Cheerfulness rating: " << cheer << "\n";
}

```

Однако объявления деструктора как чисто виртуальной функции достаточно, чтобы сделать базовый класс абстрактным и предотвратить создание объектов класса **Worker**.

Еще один момент, на который следует обратить внимание, состоит в том, что некоторые конструкторы производного класса обращаются к конструктору **Worker(const Worker &)**, даже если класс не определял его. Однако вспомните, что компилятор генерирует этот конструктор (конструктор копирования), если мы не делаем этого. Заданный по умолчанию конструктор прекрасно подходит для этого конкретного описания.

Затем нужно определить те функции, которые еще не имеют встроенных определений. Эта информация поддерживается в листинге 13.15. В нем также инициализируется массив указателей на данные, используемые классом **Singer**, для диапазона доступа. Ради краткости в примере не предпринимаются серьезные усилия для проверки корректности всей вводимой информации.

В заключение в листинге 13.16 приведена программа, предназначенная для проверки схемы использования массива указателей базового класса для управления семейством классов. В этом примере кафе "Лола" нанимает первых служащих.

### Листинг 13.16 Программа workarr.cpp.

```
// workarr.cpp - массив рабочих
// Компилировать вместе с программами worker.cpp, strng2.cpp
#include <iostream>
using namespace std;
#include <cstring>
#include "worker.h"
#include "arraytp.h" // пропустите, если шаблоны не реализованы
const int SIZE = 5;
int main()
{
 ArrayTP<Worker *, SIZE> lolas;
 // если никаких шаблонов нет, пропустите вышеприведенную строку и используйте ту,
 // которая приведена ниже Worker * lolas[SIZE];
 cout << "Enter staff for Lola's: \n";
 int ct;
 for (ct = 0; ct < SIZE; ct++)
 {
 char choice;
 cout << "Enter the employee category:\n"
 << "g: greeter w: waiter s: singer " << "q: quit\n";
 cin >> choice;
 while (strchr("gwsq", choice) == NULL)
 {
 cout << "Please enter an g, w, s, or q: ";
 cin >> choice;
 }
 if (choice == 'q')
 break;
 switch(choice)
 {
 case 'g': lolas[ct] = new Greeter; break;
 case 'w': lolas[ct] = new Waiter; break;
 case 's': lolas[ct] = new Singer; break;
 }
 cin.get();
 lolas[ct]->Set();
 }
 cout << "\nHere is your staff:\n";
 int i;
 for (i = 0; i < ct; i++)
 {
 cout << '\n';
 lolas[i]->Show();
 }
 for (i = 0; i < ct; i++)
 delete lolas[i];
 return 0;
}
```

## ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если используемая система не поддерживает шаблоны, вместо этого можно использовать обычный массив указателей на класс `Worker`. Дополнительные инструкции приведены в листинге.

Результаты выполнения программы:

```
Enter staff for Lola's:
Enter the employee category:
g: greeter w: waiter s: singer q: quit
w
Enter worker's name: Fran Godot
Enter worker's ID: 1004
Enter waiter's panache rating: 6
Enter the employee category:
g: greeter w: waiter s: singer q: quit
s
Enter worker's name: Igor Tunefree
Enter worker's ID: 1009
Enter number for singer's vocal range:
0: other 1: alto 2: contralto 3: soprano
4: bass 5: baritone 6: tenor
4
Enter the employee category:
g: greeter w: waiter s: singer q: quit
q
Enter worker's name: Hap Gladfoote
Enter worker's ID: 1003
Enter greeter's cheerfulness rating: 8
Enter the employee category:
g: greeter w: waiter s: singer q: quit
w
Enter worker's name: Walt Wiltcress
Enter worker's ID: 1022
Enter waiter's panache rating: 5
Enter the employee category:
g: greeter w: waiter s: singer q: quit
q

Here is your staff:

Category: waiter
Name: Fran Godot
Employee ID: 1004
Panache rating: 6

Category: singer
Name: Igor Tunefree
Employee ID: 1009
Vocal range: bass

Category: greeter
Name: Hap Gladfoote
Employee ID: 1003
Cheerfulness rating: 8

Category: waiter
Name: Walt Wiltcress
Employee ID: 1022
Panache rating: 5
```

## Примечание к программе

Стандартная функция ANSI C `strchr(const char * str, char ch)` осуществляет поиск символа `ch` в строке `str`. Она возвращает адрес первого местонахождения `ch`, если находит его; в противном случае функция возвращает нулевой указатель. Таким образом, код

```
while (strchr("ewsq", choice) == NULL)
```

обеспечивает удобный способ представления того, что введенный символ является допустимым.

Оператор `switch` присваивает адрес нового объекта одному из элементов указателя-на-`Worker` массива:

```
switch(choice)
{
 case 'g': lolas[ct] = new Greeter;
 break;
 case 'w': lolas[ct] = new Waiter;
 break;
 case 's': lolas[ct] = new Singer;
 break;
}
```

В зависимости от вводимых пользовательских данных указатель может указывать на объекты `Worker`, `Waiter` или `Greeter`. Поскольку базовый указатель может указывать на объект любого производного класса, приведение типа не требуется.

`Set()` и `Show()` определены как виртуальные функции, поэтому при вызове этих функций с указателем-на-`Worker` вызывается функция, соответствующая указанному классу:

```
lolas[ct]->Set();
...
lolas[i]->Show();
```

Таким образом, однородную совокупность указателей можно использовать для управления разнородной совокупностью объектов. Это пример полиморфизма в действии: один вызов функции в зависимости от контекста может активизировать различные функции.

Обратите внимание на следующий программный код:

```
for (i = 0; i < ct; i++)
 delete lolas[i];
```

Память, выделенная с помощью оператора `new`, должна быть освобождена с помощью оператора `delete`. Это не входит в задачу класса `ArrayTP`, поскольку объект `lolas` не создает, и не уничтожает объекты семейства `Worker`. Он просто хранит адреса объектов.

## Многосторонность шаблона

К классам шаблона можно применять те же самые методы, что и к обычным классам. Классы шаблонов могут служить в качестве базовых классов и могут быть компонентными классами. Они сами могут быть аргу-

ментами типа для других шаблонов. Например, шаблон стека можно реализовать, используя шаблон массива. Кроме того, можно иметь шаблон массива, используемый для создания массива, элементы которого — стеки, основанные на шаблоне стека. Иначе говоря, можно использовать код со следующими строками:

```
template <class T>
class Array
{
private:
 T entry;
 ...
};

template <class Type>
class GrowArray : public Array<Type> {...} ;
 // наследование
template <class Tp>
class Stack
{
 Array<Tp> ar; // использование Array<>
 // в качестве компонента
 ...
 ...
 Array < Stack<int> > asi; // массив стеков
 // объектов типа int
```

В последнем операторе два символа `>` должны быть разделены, по крайней мере, одним символом пробела во избежание путаницы с оператором `>>`.

Другой пример многосторонности шаблонов — то, что шаблоны могут использоваться рекурсивно. Например, располагая ранее приведенным определением шаблона массива, его можно использовать следующим образом:

```
ArrayTP< ArrayTP<int,20>, 10> twodee;
```

При этом создается массив из 10 элементов, `twodee`, каждый из элементов которого, в свою очередь, создает массив из 20 элементов типа `int`.

Могут существовать шаблоны более чем с одним параметром типа. Например, предположим, что требуется класс, который содержит два вида значений. Можно создать и использовать класс шаблона `Pair` для хранения двух различных значений. (Совершенно случайно Standard Template Library содержит аналогичный шаблон, именуемый `pair`.) Короткий пример программы приведен в листинге 13.17.

### Листинг 13.17 Программа pairs.cpp.

```
//pairs.cpp—определяет и использует шаблон
Pair

#include <iostream>
using namespace std;

template <class T1, class T2>
class Pair
{
```

```
private:
 T1 a;
 T2 b;
public:
 T1 & first(const T1 & f);
 T2 & second(const T2 & s);
 T1 first() const { return a; }
 T2 second() const { return b; }
 Pair(const T1 & f, const T2 & s) : a(f),
 b(s) { }

template<class T1, class T2>
T1 & Pair<T1,T2>::first(const T1 & f)
{
 a = f;
 return a;
}
template<class T1, class T2>
T2 & Pair<T1,T2>::second(const T2 & s)
{
 b = s;
 return b;
}

int main()
{
 Pair<char *, int> ratings[4] =
 {
 Pair<char *, int>("The Purple Duke", 5),
 Pair<char *, int>("Jake's Frisco Cafe", 4),
 Pair<char *, int>("Mont Souffle", 5),
 Pair<char *, int>("Gertie's Eats", 3)
 };

 int joints = sizeof(ratings) /
 sizeof (Pair<char *, int>);
 cout << "Rating:\t Eatery\n";
 for (int i = 0; i < joints; i++)
 cout << ratings[i].second() << ":\t "
 << ratings[i].first() << "\n";

 ratings[3].second(6);
 cout << "Oops! Revised rating:\n";
 cout << ratings[3].second() << ":\t "
 << ratings[3].first() << "\n";
 return 0;
}
```

Следует обратить внимание, что внутри `main()` можно использовать `Pair<char *,int>` для вызова конструкторов и в качестве аргумента для `sizeof`. Дело в том, что именем класса является `Pair<char *,int>`, а не `Pair`. Кроме того, `Pair<String,ArrayDb>` может быть именем совершенно другого класса.

Результаты выполнения программы:

|         |                    |
|---------|--------------------|
| Rating: | Eatery             |
| 5:      | The Purple Duke    |
| 4:      | Jake's Frisco Cafe |
| 5:      | Mont Souffle       |
| 3:      | Gertie's Eats      |
| Oops!   | Revised rating:    |
| 6:      | Gertie's Eats      |

Еще одно новое свойство шаблона класса — возможность обеспечения определенных по умолчанию значений для параметров-типов:

```
template <class T1, class T2 = int>
class Map {...};
```

Это заставляет компилятор использовать `int` в качестве типа `T2`, если значение для `T2` опущено:

```
Map<double, double> m1; // T1 - double,
 // T2 - double
Map<double> m2; // T1 - double, T2 - int
```

Стандартная библиотека шаблонов (см. главу 15) часто использует это свойство, определяя класс в качестве заданного по умолчанию типа.

Хотя можно обеспечивать значения, принятые по умолчанию, для параметров-типов шаблона класса, этого нельзя делать для параметров шаблонов функций. Однако можно обеспечивать значения, принятые по умолчанию, которые не относятся к типам параметров как для шаблонов классов, так и для шаблонов функций.

## Специализации шаблонов

Шаблоны классов подобны шаблонам функций тем, что позволяют использовать неявные образования экземпляров, явные образования экземпляров и явные специализации, которые обобщенно называются *специализациями*. Другими словами, шаблон описывает класс, используя термины общего типа, в то время как специализация представляет собой объявление класса, сгенерированное с помощью конкретного типа.

## Неявные образования экземпляров

В приводившихся до сих пор примерах использовались *неявные образования экземпляров*. В них объявляется один или более объектов с указанием желательного типа, и компилятор генерирует специализированное определение класса, используя предписания, которые обеспечиваются общим шаблоном:

```
ArrayTb<int, 100> stuff; // неявное
 // образование экземпляра
```

Компилятор не генерирует неявный экземпляр класса до тех пор, пока не потребуется объект:

```
ArrayTb<double, 30> * pt; //указатель,
 //никакой объект пока не требуется
pt = new ArrayTb<double, 30>; //теперь
 //объект необходим
```

Второй оператор заставляет компилятор генерировать определение класса и создавать объект в соответствии с этим определением.

## Явные образования экземпляров

Компилятор генерирует *явный экземпляр* объявления класса при объявлении класса с использованием ключевого слова `template` и указанием желательного типа или типов:

```
// генерирует класс ArrayTB<String, 100>
template ArrayTb<String, 100>;
```

В этой строке `ArrayTb<String, 100>` объявляется классом. В этом случае компилятор генерирует определение класса независимо от того, что никакой объект класса еще не был создан или упомянут. Однако, как и при неявном создании экземпляра, общий шаблон используется в качестве руководства для генерирования специализации.

## Явные специализации

*Явная специализация* представляет собой определение для конкретного типа или типов, который должен использоваться вместо общего шаблона. Иногда может потребоваться изменить шаблон, чтобы он вел себя по-другому при создании экземпляра для конкретного типа; в этом случае можно задать явную специализацию. Предположим, например, что был определен шаблон для класса, представляющего упорядоченный массив, для которого элементы сортируются по мере их добавления в массив:

```
template <class T>
class SortedArray
{
 ... // подробности опущены
};
```

Предположим также, что шаблон использует оператор `>` для сравнения значений. Это успешно срабатывает не только в том случае, когда речь идет о числах, но и когда `T` представляет тип класса — при условии, что был определен метод `T::operator>()`. Но такой подход не приемлем, если `T` — строка, представляемая типом `char *`. Фактически шаблон будет работать, но строки окажутся отсортированными по адресам, а не в алфавитном порядке. В данном случае требуется определение класса, в котором `strcmp()` используется вместо `>`. В подобной ситуации можно обеспечить явную специализацию шаблона. Функция принимает форму шаблона, определенного для одного конкретного типа, а не шаблона для общего типа. Сталкиваясь с необходимостью выбора между специализированным шаблоном и общим шаблоном, которые оба соответствуют запросу на создание экземпляра, компилятор использует специализированную версию.

Специализированное определение шаблона класса имеет следующую форму:

```
template <> class
 Имя_класса<имя_специализированного_типа>
{ ... } ;
```

Устаревшие компиляторы могут распознавать только более старую форму, в которой отсутствует последовательность `template <>`:

```
class Имя_класса<имя_специализированного_типа>
{ ... } ;
```

Для поддержки шаблона `SortedArray`, специализированного для типа `char *`, при новой форме записи нужно было бы использовать программный код, подобный следующему:

```
template <> class SortedArray<char *>
{
 ... // подробности опущены
};
```

Здесь код реализации использовал бы функцию `strcmp()` вместо операции `>` для сравнения значений массива. Теперь запросы `char *` относительно `SortedArray` будут использовать следующее специализированное определение вместо более общего определения шаблона:

```
SortedArray<int> scores; // использует общее
 // определение
SortedArray<char *> dates; // использует
 // специализированное определение
```

### Частичные специализации

C++ допускает также *частичные специализации*, которые частично ограничивают общность шаблона. Например, частичная специализация может обеспечить конкретный тип для одного из параметров-типов:

```
// общий шаблон
template <class T1, class T2> class Pair {...};
// специализация, где для T2 установлен тип int
template <class T1> class Pair<T1, int> {...};
```

Угловые скобки `<>`, следующие за ключевым словом `template`, определяют объявление параметров типа, которые еще не специализированы. Так, второе объявление специализирует `T2` как `int`, но оставляет `T1` открытым. Обратите внимание, что указание всех типов задается пустой парой скобок и полной явной специализацией:

```
// специализация со значениями T1 и T2,
// имеющими тип int
template <> class Pair<int, int> { ... } ;
```

При наличии выбора компилятор использует наиболее специализированный шаблон:

```
// использует общий шаблон Pair
Pair<double, double> p1;
```

```
// использует частичную специализацию
// Pair<T1, int>
Pair<double, int> p2;
// использует явную специализацию
// Pair<int, int>
Pair<int, int> p3;
```

Можно также частично специализировать существующий шаблон, обеспечив специальную версию для указателей:

```
template<class T> // общая версия
class Feeb { ... };
template<class T*> // частичная
 // специализация указателя
class Feeb { ... }; // измененный код
```

При обеспечении отличающегося от указателя типа компилятор использует общую версию; при обеспечении указателя компилятор использует специализацию указателя:

```
// использует общий шаблон Feeb, T - char
Feeb<char> fb1;
// использует специализацию Feeb T*, T - char
Feeb<char *> fb2;
```

Без частичной специализации второе объявление использовало бы общий шаблон, интерпретируя `T` как тип `char *`. С частичной специализацией используется специализированный шаблон, интерпретируя `T` как `char`.

Свойство частичной специализации позволяет задавать ряд ограничений. Например, можете сделать следующее:

```
// общий шаблон
template <class T1, class T2, class T3>
class Trio{...};
// специализация с T3, имеющим тип T2
template <class T1, class T2> class
Trio<T1, T2, T2> {...};
// специализация с T3 и T2, также имеющими
// тип T1*
template <class T1> class Trio<T1, T1*, T1*> {...};
```

При этих заданных объявлениях компилятор осуществлял бы следующий выбор:

```
// использует общий шаблон
Trio<int, short, char *> t1;
// использует Trio<T1, T2, T2>
Trio<int, short> t2;
Trio<char, char *, char *> t3;
use Trio<T1, T1*, T1*>
```

### Множественное наследование

*Множественное наследование* (*Multiple inheritance — MI*) описывает класс, который включает более одного непосредственного базового класса. Как и в случае с одиноч-

ным наследованием, общедоступное множественное наследование должно выражать отношение *is-a*. Например, если имеются классы *Waiter* и *Singer*, из них можно было бы получить класс *SingingWaiter*:

```
class SingingWaiter : public Waiter,
 public Singer {...};
```

Обратите внимание, что каждый базовый класс необходимо квалифицировать с помощью ключевого слова *public*. Дело в том, что компилятор предполагает приватное произведение производного объекта, если не указано иное:

```
class SingingWaiter : public Waiter, Singer
 {...}; //Singer — приватный базовый класс
```

Как было рассмотрено ранее в этой главе, приватное и защищенное MI может выражать отношение *has-a*. Теперь же мы сосредоточим свое внимание на общедоступном наследовании.

Множественное наследование может представлять новые проблемы для программиста. Двумя главными проблемами являются наследование различных методов с одним и тем же именем из двух различных базовых классов и наследование нескольких экземпляров класса посредством двух или более связанных непосредственных базовых классов. Решение этих проблем включает представление нескольких новых правил и вариаций синтаксиса. Таким образом, использование метода множественного наследования может быть более трудным и сопряженным с проблемами, чем использование метода одиночного наследования. По этой причине в сообществе C++ ведутся горячие споры по поводу MI; некоторые настаивают на его удалении из языка. Другим MI

нравится, и они доказывают, что оно очень полезно, даже необходимо для конкретных проектов. Третий предлагают использовать MI осторожно и умеренно.

Давайте исследуем конкретный пример и посмотрим, каковы проблемы и их решения. Мы используем вариацию примера класса *Worker* (листинги 13.14–13.16). Пример включает классы *Waiter* и *Singer*, полученные из абстрактного базового класса *Worker*. Таким образом, MI можно использовать для получения класса *SingingWaiter* из классов *Waiter* и *Singer* (рис. 13.3). Это как раз тот случай, когда базовый класс (*Worker*) наследуется через два отдельных получения производных классов — обстоятельство, вызывающее большинство трудностей, связанных с MI. Ради сокращения рассматриваемого примера давайте опустим класс *Greeter*. Далее предположим, что создание класса *SingingWaiter*, который будет служить для иллюстрации некоторых проблем, мы начинаем с листинга 13.14 (*worker.h*) после удаления из него раздела *Greeter*. В частности, придется столкнуться со следующими вопросами:

- Сколько рабочих?
- Какой метод?

## Определение количества рабочих

Предположите, что вы начинаете с общедоступного получения *SingingWaiter* из *Singer* и *Waiter*:

```
class SingingWaiter: public Singer,
 public Waiter {...};
```

Поскольку и *Singer*, и *Waiter* наследуют компонент *Worker*, *SingingWaiter* окажется с двумя компонентами *Worker* (рис. 13.4).

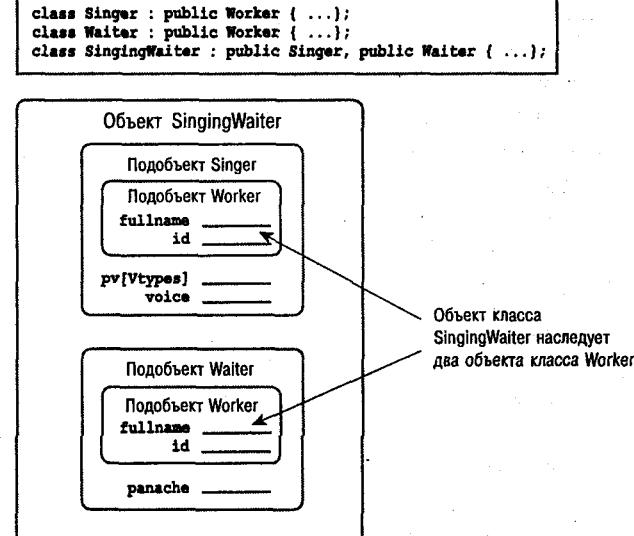
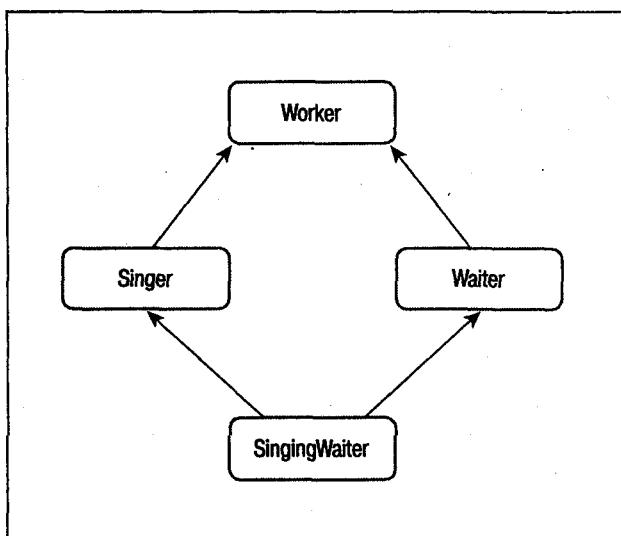


РИС. 13.3 Множественное наследование с общим предком.

РИС. 13.4 Наследование двух базовых объектов.

Как и можно было ожидать, это порождает проблемы. Например, обычно можно присваивать адрес объекта производного класса указателю базового класса, но теперь это становится неоднозначным:

```
SingingWaiter ed;
Worker * pw = &ed; // неоднозначно
```

Обычно такое присваивание устанавливает указатель базового класса на адрес объекта базового класса внутри производного объекта. Но `ed` содержит два объекта класса `Worker`, следовательно, имеется два адреса, из которых приходится выбирать. Объект можно было бы указать, выполняя приведение типа:

```
// Worker в Waiter
Worker * pw1 = (Waiter *) &ed;

// Worker в Singer
Worker * pw2 = (Singer *) &ed;
```

Это, конечно, усложняет методику использования массива указателей базового класса для обращения к ряду объектов (полиморфизм).

Наличие двух копий объекта `Worker` вызывает также и другие проблемы. Однако действительный вопрос заключается в том, зачем вообще иметь две копии объекта `Worker`. Поющий офицант, подобно любому другому рабочему, должен иметь только одно имя и один идентификатор. Когда C++ добавил множественное наследование к своему набору приемов, он добавил и новую методику, *виртуальный базовый класс*, делающий это возможным.

### Виртуальные базовые классы

Виртуальные базовые классы позволяют объекту, производному из нескольких базовых объектов, которые сами совместно используют общий базовый класс, наследовать только один объект этого общего базового класса. Для данного примера нужно было бы сделать `Worker` виртуальным базовым классом для `Singer` и `Waiter`, используя ключевое слово `virtual` в объявлениях класса (ключевые слова `virtual` и `public` могут появляться в любом порядке):

```
class Singer : virtual public Worker { ... };
class Waiter : public virtual Worker { ... };
```

Затем нужно было бы определить `SingingWaiter` как прежде:

```
class SingingWaiter: public Singer,
 public Waiter { ... };
```

Теперь объект `SingingWaiter` будет содержать единственную копию объекта `Worker`. В сущности, унаследованные объекты `Singer` и `Waiter` совместно используют общий объект `Worker` вместо поддержки собственной копии каждым из них (рис. 13.5). Поскольку теперь `SingingWaiter` содержит только один подобъект `Worker`, можно снова использовать полиморфизм.

Давайте рассмотрим вопросы, которые могут возникнуть:

- Почему используется термин "виртуальный"?
- Почему не обходятся без объявления базовых классов в качестве виртуальных и не делают виртуальное поведение нормой для многократного наследования?
- Существуют ли какие-либо скрытые опасности?

Прежде всего, почему выбран термин "виртуальный"? В конце концов, на первый взгляд не существует очевидной связи между концепциями виртуальных функций и виртуальных базовых классов. Просто со стороны сообщества пользователей C++ оказывается сильное сопротивление введению новых ключевых слов. Было бы неудобно, например, если бы новое ключевое слово соответствовало имени какой-либо важной функции или переменной в главной программе. Поэтому в C++ ключевое слово `virtual` просто используется повторно для нового средства — ключевое слово оказывается слегка перегруженным.

Далее, почему бы не обойтись без объявления базовых классов в качестве виртуальных и не сделать виртуальное поведение нормой для множественного наследования? Во-первых, бывают случаи, когда могут потребоваться несколько копий базового объекта. Во-вторых, превращение базового класса в виртуальный требует, чтобы программа выполняла некоторые дополнительные расчеты, а пользователь не должен оплачивать то средство, в котором он не нуждается. В-третьих, существуют еще некоторые вопросы, поднятые в следующем разделе.

```
class Singer : virtual public Worker { ... };
class Waiter : virtual public Worker { ... };
class SingingWaiter : public Singer, public Waiter { ... };
```

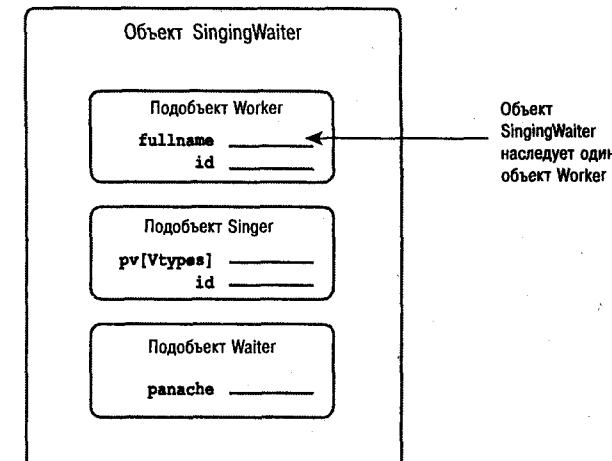


РИСУНОК 13.5 Наследование с виртуальным базовым классом.

И наконец, существуют ли какие-либо скрытые проблемы? Да. Обеспечение работы виртуальных базовых классов требует корректировок в правилах C++, и некоторые фрагменты программы должны быть закодированы по-другому. Кроме того, использование виртуальных базовых классов может требовать изменения существующего кода. Например, добавление класса **SingingWaiter** в иерархию **Worker** потребовало добавления ключевого слова **virtual** в классы **Singer** и **Waiter**.

### Новые правила конструктора

Наличие виртуальных базовых классов требует нового подхода к конструкторам классов. В случае с невиртуальными базовыми классами **единственные конструкторы**, которые могут появляться в списке инициализации, — это конструкторы для непосредственных базовых классов. Но эти конструкторы могут, в свою очередь, передавать информацию своим базовым классам. Например, можете существовать следующая организация конструкторов:

```
class A
{
 int a;
public:
 A(int n = 0) { a = n; }

};

class B : public A
{
 int b;
public:
 B(int m = 0, int n = 0) : A(n) : {b = m; }

};

class C : public B
{
 int c;
public:
 C(int q = 0, int m = 0, int n = 0) :
 B(m, n) { c = q; }

};
```

Конструктор **C** может вызывать конструкторы только из класса **B**, а конструктор **B** может вызывать конструкторы только из класса **A**. Здесь конструктор **C** использует значение **q** и возвращает значения **m** и **n** конструктору **B**. Конструктор **B** использует значение **m** и возвращает значение **n** конструктору **A**.

Эта автоматическая передача информации может быть несуществима, если **Worker** является виртуальным базовым классом. Например, рассмотрим следующий возможный конструктор для примера множественного наследования:

```
SingingWaiter(const Worker & wk, int p = 0,
 int v = Singer::other) :
 Waiter(wk, p), Singer(wk, v) { }
```

Проблема состоит в том, что при автоматической передаче информации аргумент **wk** передавался бы объекту **Worker** по двум отдельным путям (посредством **Waiter** и посредством **Singer**). Во избежание этого потенциального конфликта C++ отключает автоматическую передачу информации через промежуточный класс к базовому классу, *если* базовый класс виртуален. Таким образом, вышеупомянутый конструктор будет инициализировать элементы **panache** и **voice**, но информация аргумента **wk** не будет попадать в подобъект **Waiter**. Однако компилятор должен создать компонент базового объекта перед созданием производных объектов; в рассматриваемом случае он будет использовать заданный по умолчанию конструктор **Worker**.

Если для виртуального базового класса желательно использовать конструктор, отличающийся от заданного по умолчанию конструктора, нужно вызвать базовый конструктор явно. Поэтому конструктор должен выглядеть следующим образом:

```
SingingWaiter(const Worker & wk, int p = 0,
 int v = Singer::other) : Worker(wk),
 Waiter(wk, p), Singer(wk, v) { }
```

Здесь в программном коде явно вызывается конструктор **Worker(const Worker &)**. Обратите внимание, что такое использование допустимо и часто необходимо для виртуальных базовых классов и запрещено для невиртуальных базовых классов.

### ПРЕДОСТЕРЕЖЕНИЕ

Если класс имеет косвенный виртуальный базовый класс, конструктор для этого класса должен явно вызвать конструктор для виртуального базового класса, если только он не нуждается исключительно в заданном по умолчанию конструкторе для виртуального базового класса.

### Выбор метода

Кроме введения изменений в правила конструктора класса, MI часто требует и других изменений, выполненных для кода. Давайте рассмотрим проблему распространения метода **Show()** на класс **SingingWaiter**. Поскольку объект **SingingWaiter** не включает новых элементов данных, можно было бы думать, что класс использует только унаследованные методы. Это порождает первую проблему. Предположим, что вы не указываете новую версию **Show()** и пытаетесь использовать объект **SingingWaiter** для вызова унаследованного метода **Show()**:

```
SingingWaiter newhire("Elise Hawks", 2005,
 6, soprano);
newhire.Show(); // неоднозначно
```

При одиночном наследовании отсутствие переопределения метода **Show()** приводит к использованию са-

мого последнего наследственного определения. В этом случае каждый прямой предок имеет функцию `Show()`, что делает это обращение неоднозначным.

### ПРЕДОСТЕРЕЖЕНИЕ

Множественное наследование может приводить к неоднозначным обращениям к функциям. Например, класс `BadDude` мог бы наследовать два совершенно различных метода `Draw()` из класса `Gunslinger` и класса `PokerPlayer`.

Для прояснения того, что имеется в виду, можно использовать оператор определения диапазона доступа:

```
SingingWaiter newhire("Elise Hawks", 2005,
 6, soprano);
newhire.Singer::Show(); // использует
 // версию Singer
```

Однако лучший подход — переопределение метода `Show()` для `SingingWaiter` с условием, чтобы он указывал, какой метод `Show()` следует использовать. Например, если нужно, чтобы объект `SingingWaiter` использовал версию `Singer`, выполните следующее:

```
void SingingWaiter::Show()
{
 Singer::Show();
}
```

Этот подход, при котором производный метод вызывает базовый метод, работает достаточно хорошо для одиночного наследования. Например, предположим, что класс `HeadWaiter` является производным от класса `Waiter`. Тогда можно было бы использовать последовательность определений, подобную следующей, в которой каждый производный класс расширяет информацию, отображаемую его базовым классом:

```
void Worker::Show() const
{
 cout << "Name: " << fullname << "\n";
 cout << "Employee ID: " << id << "\n";
}

void Waiter::Show() const
{
 Worker::Show();
 cout << "Panache rating: " << panache << "\n";
}

void HeadWaiter::Show() const
{
 Waiter::Show();
 cout << "Presence rating: "
 << presence << "\n";
}
```

Однако этот инкрементный подход терпит неудачу в случае с `SingingWaiter`. Метод

```
void SingingWaiter::Show()
{
 Singer::Show();
}
```

терпит неудачу потому, что он игнорирует компонент `Waiter`. Это можно исправить, вызывая также версию `Waiter`:

```
void SingingWaiter::Show()
{
 Singer::Show();
 Waiter::Show();
}
```

В результате имя и идентификатор данного лица будут отображены дважды, поскольку и `Singer::Show()`, и `Waiter::Show()` вызывают `Worker::Show()`.

Как это может быть исправлено? Один из способов состоит в использовании модульного подхода вместо инкрементного. Иначе говоря, необходимо обеспечить один метод, который отображает только компоненты `Worker`, другой метод, который отображает только компоненты `Waiter` (вместо компонентов `Waiter` плюс `Worker`), и третий метод, который отображает только компоненты `Singer`. Затем метод `SingingWaiter::Show()` может собрать эти компоненты воедино. Например, можно выполнить следующее:

```
void Worker::Data() const
{
 cout << "Name: " << fullname << "\n";
 cout << "Employee ID: " << id << "\n";
}

void Waiter::Data() const
{
 cout << "Panache rating: " << panache << "\n";
}

void Singer::Data() const
{
 cout << "Vocal range: " << pv[voice] << "\n";
}

void SingingWaiter::Data() const
{
 Singer::Data();
 Waiter::Data();
}

void SingingWaiter::Show() const
{
 cout << "Category: singing waiter\n";
 Worker::Data();
 Data();
}
```

Аналогично другие методы `Show()` были бы сформированы из соответствующих компонентов `Data()`.

При этом подходе объекты по-прежнему использовали бы метод `Show()` в общедоступном режиме. С другой стороны, методы `Data()` должны быть внутренними для классов, вспомогательными методами, используемыми в поддержку общедоступного интерфейса. Однако приданье методам `Data()` приватного характеру помешало бы, скажем, коду `Waiter` использовать метод `Worker::Data()`. Существует только одна ситуация, в

которой класс защищенного доступа является полезным. Если методы `Data()` защищены, они могут использоваться внутренне всеми классами в иерархии, в тоже время оставаясь скрытыми от внешнего мира.

Другим возможным подходом было бы превращение всех компонентов данных в защищенные вместо приватных, но использование защищенных методов вместо защищенных данных обеспечивает более жесткое управление разрешениями доступа к данным.

Методы `Set()`, которые обеспечивают данные для установки значений объектов, создают аналогичную проблему. Например, `SingingWaiter::Set()` должен запрашивать информацию `Worker` только один раз, а не дважды. В этом случае подходит то же самое решение. Можно обеспечить защищенные методы `Get()`, которые обеспечивают информацию только для одного класса, а затем собрать вместе методы `Set()`, которые используют методы `Get()` в качестве строительных блоков.

Таким образом, использование множественного наследования с общедоступным предком требует введения виртуальных базовых классов, изменения правил для списков инициализации конструкторов и, возможно, перепрограммирования классов, если они не были сохранены при многократном наследовании. Модифицированные объявления класса, представляющие эти изменения, показаны в листинге 13.18, а реализация — в листинге 13.19.

Конечно, простое любопытство требует, чтобы мы проверили эти классы, и код для выполнения этой проверки приведен в листинге 13.20, основанном на листинге 13.16. Обратите внимание, что программа использует полиморфное свойство, присваивая адреса различных видов классов указателям базового класса. Эту программу следует компилировать с `workermi.cpp` и `strng2.cpp`. Убедитесь также в наличии заголовочного файла `arraytp.h` (в котором содержится наше определение шаблона массива).

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если используемая система не поддерживает шаблоны, вместо этого можно использовать простой массив. Только внесите небольшие изменения, предложенные в листинге.

Результаты выполнения программы из листинга 13.20:

```
Enter the employee category:
w: waiter s: singer t: singing waiter q: quit
w
Enter waiter's name: Wally Snipeside
Enter worker's ID: 1020
Enter waiter's panache rating: 5
Enter the employee category:
w: waiter s: singer t: singing waiter q: quit
t
Enter singing waiter's name: Natasha Gargalova
```

Enter worker's ID: 1021

Enter waiter's panache rating: 6

Enter number for singer's vocal range:

0: other 1: alto 2: contralto 3: soprano

4: bass 5: baritone 6: tenor

3

Enter the employee category:

w: waiter s: singer t: singing waiter q: quit

q

Here is your staff:

Category: waiter

Name: Wally Snipeside

Employee ID: 1020

Panache rating: 5

Category: singing waiter

Name: Natasha Gargalova

Employee ID: 1021

Vocal range: soprano

Panache rating: 6

Давайте рассмотрим еще несколько вопросов, связанных с множественным наследованием.

#### Смешанные виртуальные и невиртуальные базовые классы

Давайте снова рассмотрим случай производного класса, который наследует базовый класс более чем по одному маршруту. Если базовый класс виртуален, производный класс содержит один подобъект базового класса. Если базовый класс невиртуален, производный класс содержит несколько подобъектов. А что, если имеет место смешанный случай? Например, предположим, что класс `B` — виртуальный базовый класс для классов `C` и `D` и невиртуальный базовый класс для классов `X` и `Y`. Кроме того, предположим, что класс `M` является производным от классов `C`, `D`, `X` и `Y`. В этом случае класс `M` содержит один подобъект класса `B` для всех виртуальных предков (т.е. для классов `C` и `D`) и отдельный подобъект класса `B` для каждого невиртуального предка (т.е. для классов `X` и `Y`). Таким образом, всего он будет содержать три подобъекта класса `B`. Когда класс наследует конкретный базовый класс посредством нескольких виртуальных и нескольких невиртуальных путей, класс содержит один подобъект базового класса для представления всех виртуальных путей и отдельный подобъект базового класса для представления каждого невиртуального пути.

#### Виртуальные базовые классы и доминирование

Использование виртуальных базовых классов изменяет способ решения неоднозначности программой C++. Правила, действующие для невиртуальных базовых классов, просты. Если класс наследует два или больше элементов (данных или методов) с одним и тем же именем из различных классов, использование этого имени без указания имени класса будет неоднозначным.

## Листинг 13.18 Класс worker.h.

```

// Worker.h - рабочие классы с MI
#include "strng2.h"

class Worker // абстрактный базовый класс
{
private:
 String fullname;
 long id;
protected:
 virtual void Data() const;
 virtual void Get();
public:
 Worker() : fullname("no one"), id(0L) { }
 Worker(const String & s, long n)
 : fullname(s), id(n) { }
 virtual ~Worker() = 0;
 virtual void Set() = 0;
 virtual void Show() const = 0;
};

class Waiter : virtual public Worker
{
private:
 int panache;
protected:
 void Data() const;
 void Get();
public:
 Waiter() : Worker(), panache(0) { }
 Waiter(const String & s, long n, int p = 0)
 : Worker(s, n), panache(p) { }
 Waiter(const Worker & wk, int p = 0)
 : Worker(wk), panache(p) { }
 void Set();
 void Show() const;
};

class Singer : virtual public Worker
{
protected:
enum { other, alto, contralto, soprano,
 bass, baritone, tenor} ;
enum { Vtypes = 7} ;
void Data() const;
void Get();
private:
 static char *pv[Vtypes]; // строковые эквиваленты типов голоса
 int voice;
public:
 Singer() : Worker(), voice(other) { }
 Singer(const String & s, long n, int v = other)
 : Worker(s, n), voice(v) { }
 Singer(const Worker & wk, int v = other)
 : Worker(wk), voice(v) { }
 void Set();
 void Show() const;
};

// множественное наследование
class SingingWaiter : public Singer, public Waiter
{
protected:
 void Data() const;
 void Get();
}

```

```

public:
 SingingWaiter() { }
 SingingWaiter(const String & s, long n, int p = 0,
 int v = Singer::other)
 : Worker(s,n), Waiter(s, n, p), Singer(s, n, v) { }
 SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
 : Worker(wk), Waiter(wk,p), Singer(wk,v) { }
 SingingWaiter(const Waiter & wt, int v = other)
 : Worker(wt),Waiter(wt), Singer(wt,v) { }
 SingingWaiter(const Singer & wt, int p = 0)
 : Worker(wt),Waiter(wt,p), Singer(wt) { }
 void Set();
 void Show() const;
};

}

```

### Листинг 13.19 Программа workermi.cpp.

```

// workermi.cpp - методы рабочего класса с МИ
#include "workermi.h"
#include <iostream>
using namespace std;
// методы Worker
Worker::~Worker() { }

// защищенные методы
void Worker::Data() const
{
 cout << "Name: " << fullname << "\n";
 cout << "Employee ID: " << id << "\n";
}

void Worker::Get()
{
 cin >> fullname;
 cout << "Enter worker's ID: ";
 cin >> id;
 while (cin.get() != '\n')
 continue;
}

// методы Waiter
void Waiter::Set()
{
 cout << "Enter waiter's name: ";
 Worker::Get();
 Get();
}

void Waiter::Show() const
{
 cout << "Category: waiter\n";
 Worker::Data();
 Data();
}

// защищенные методы
void Waiter::Data() const
{
 cout << "Panache rating: " << panache << "\n";
}

void Waiter::Get()
{
}

```

```

cout << "Enter waiter's panache rating: ";
cin >> panache;
while (cin.get() != '\n')
 continue;
}

// методы Singer

char * Singer::pv[Singer::Vtypes] = { "other", "alto", "contralto",
 "soprano", "bass", "baritone", "tenor" };

void Singer::Set()
{
 cout << "Enter singer's name: ";
 Worker::Get();
 Get();
}

void Singer::Show() const
{
 cout << "Category: singer\n";
 Worker::Data();
 Data();
}

// замещенные методы
void Singer::Data() const
{
 cout << "Vocal range: " << pv[voice] << "\n";
}

void Singer::Get()
{
 cout << "Enter number for singer's vocal range:\n";
 int i;
 for (i = 0; i < Vtypes; i++)
 {
 cout << i << ": " << pv[i] << " ";
 if (i % 4 == 3)
 cout << '\n';
 }
 if (i % 4 != 0)
 cout << '\n';
 cin >> voice;
 while (cin.get() != '\n')
 continue;
}

// методы SingingWaiter
void SingingWaiter::Data() const
{
 Singer::Data();
 Waiter::Data();
}

void SingingWaiter::Get()
{
 Waiter::Get();
 Singer::Get();
}

void SingingWaiter::Set()
{
 cout << "Enter singing waiter's name: ";
}

```

```

Worker::Get();
Get();
}

void SingingWaiter::Show() const
{
 cout << "Category: singing waiter\n";
 Worker::Data();
 Data();
}

```

### Листинг 13.20 Программа workmi.cpp.

```

// workmi.cpp - множественное наследование
// Компилируйте с программами workermi.cpp, strng2.cpp

#include <iostream>
using namespace std;
#include <cstring>
#include "worker.h"
#include "arraytp.h" // пропустить, если никакой шаблон не поддерживается
const int SIZE = 5;
int main()
{
 ArrayTP<Worker *, SIZE> lolas;
 // если никакой шаблон не поддерживается, опустите вышеприведенную
 // строку и используйте следующую: Worker * lolas[SIZE];
 int ct;
 for (ct = 0; ct < SIZE; ct++)
 {
 char choice;
 cout << "Enter the employee category:\n"
 << "w: waiter s: singer " << "t: singing waiter q: quit\n";
 cin >> choice;
 while (strchr("ewstq", choice) == NULL)
 {
 cout << "Please enter a w, s, t, or q: ";
 cin >> choice;
 }
 if (choice == 'q')
 break;
 switch(choice)
 {
 case 'w': lolas[ct] = new Waiter;
 break;
 case 's': lolas[ct] = new Singer;
 break;
 case 't': lolas[ct] = new SingingWaiter;
 break;
 }
 cin.get();
 lolas[ct]->Set();
 }

 cout << "\nHere is your staff:\n";
 int i;
 for (i = 0; i < ct; i++)
 {
 cout << '\n';
 lolas[i]->Show();
 }
 for (i = 0; i < ct; i++)
 delete lolas[i];
 return 0;
}

```

Однако, если применяются виртуальные базовые классы, такое использование может быть как неоднозначным, так и неоднозначным. В этом случае, если одно имя **доминирует** над всеми другими, оно может однозначно использоваться без спецификатора.

Так каким же образом одно имя элемента доминирует над другим? Имя в производном классе доминирует над этим же именем в любом классе предка, прямом или косвенном. Например, рассмотрите следующие определения:

```
class B
{
public:
 short q();
 ...
};

class C : virtual public B
{
public:
 long q();
 int omb()
 ...
};

class D : public C
{
 ...
};

class E : virtual public B
{
private:
 int omb();
 ...
};

class F: public D, public E
{
 ...
};
```

Здесь определение **q()** в классе **C** доминирует над определением в классе **B**, поскольку **C** получен из **B**. Таким образом, в методах класса **F** имя **q()** может использоваться для обозначения **C::q()**. С другой стороны, ни одно определение **omb()** не доминирует над другим, потому что ни **C**, ни **E** не являются базовыми классами друг для друга. Следовательно, предпринятая со стороны класса **F** попытка использовать функцию **omb()** без указания класса имела бы неоднозначные последствия.

Правила исключения виртуальной неоднозначности не охватывают правила обеспечения доступа. Это значит, что, даже если метод **E::omb()** является частным и, следовательно, недоступен непосредственно для класса **F**, использование функции **omb()** дает неоднозначные результаты.

Аналогично, даже если бы метод **C::q()** был частным, он бы доминировал над **D::q()**. В таком случае можно

было бы вызывать **B::q()** в классе **F**, но использование вместо этого функции **q()** без указания класса относилось бы к недоступному методу **C::q()**.

## Некоторые итоги по теме множественного наследования

Вначале давайте сделаем обзор по теме множественного наследования без виртуальных базовых классов. Эта форма MI не налагает никаких новых правил. Однако, если класс наследует два элемента с одним и тем же именем, но из различных классов, в производном классе нужно использовать спецификаторы класса для различия двух элементов. Иначе говоря, методы в классе **BadDude**, полученном из **Gunslinger** и **PokerPlayer**, использовали бы **Gunslinger::draw()** и **PokerPlayer::draw()** для различия методов **draw()**, унаследованных из двух классов. В противном случае компилятор должен был бы выводить сообщение о неоднозначном использовании.

Если один класс наследуется из базового класса, не являющегося виртуальным, более чем по одному маршруту, то класс наследует по одному объекту базового класса для каждого невиртуального экземпляра базового класса. Иногда это может быть желательным, но чаще несколько экземпляров базового класса представляют проблему.

Далее давайте рассмотрим MI совместно с виртуальными базовыми классами. Класс становится виртуальным базовым классом, когда при указании получения производный класс использует ключевое слово **virtual**:

```
class marketing : public virtual reality {...};
```

Основное отличие и причина применения виртуальных базовых классов состоит в том, что класс, который наследуется из одного или более экземпляров виртуального базового класса, наследует только один объект базового класса. Реализация этого свойства влечет за собой и другие требования:

- Производный класс с косвенным виртуальным базовым классом должен иметь конструкторы, которые вызывают косвенные конструкторы базового класса непосредственно, что запрещено для косвенных невиртуальных базовых классов.
- Неоднозначность имени решается правилом доминирования.

Как видите, многократное наследование может создавать сложности при программировании. Однако большинство этих сложностей возникает, когда производный класс наследуется из одного и того же самого базового класса более чем по одному маршруту. Во избежание этой ситуации придется заботиться только о полном указании унаследованных имен при их использовании.

## Резюме

Язык C++ обеспечивает несколько средств для многократного использования кода. Общедоступное наследование, описанное в главе 12, позволяет моделировать отношения *is-a* с производными классами, которые способны повторно использовать код базовых классов. Приватное и защищенное наследование также позволяют повторно использовать код базового класса, на сей раз моделируя отношения *has-a*. При использовании метода приватного наследования общедоступные и защищенные элементы базового класса становятся приватными элементами производного класса. При использовании метода защищенного наследования общедоступные и защищенные элементы базового класса становятся защищенными элементами производного класса. Таким образом, в любом случае общедоступный интерфейс базового класса становится внутренним интерфейсом для производного класса. Иногда это описывается как наследование реализации, но не интерфейса, поскольку производный объект не может явно использовать интерфейс базового класса. Таким образом, производный объект нельзя рассматривать как своего рода базовый объект. Из-за этого указателю или ссылке базового класса не разрешается ссылаться на производный объект без явного приведения типа.

Код класса можно также повторно использовать, разработав класс с элементами, которые сами являются объектами. Этот подход, называемый включением, иерархическим представлением или композицией, также моделирует отношение *has-a*. Метод включения проще реализовать и использовать, чем метод приватного или защищенного наследования, поэтому обычно предпочтитаю его. Однако приватное и защищенное наследование обеспечивает более широкие возможности. Например, наследование позволяет производному классу получать доступ к защищенным элементам базового класса, а также переопределять виртуальную функцию, унаследованную из базового класса. Поскольку включение не является формой наследования, ни одна из этих возможностей недоступна при повторном использовании кода класса посредством включения. С другой стороны, включение больше подходит, если нужны несколько объектов данного класса. Например, класс *State* мог бы включать массив объектов *County*.

Множественное наследование (MI) позволяет в проекте класса повторно использовать код более чем одного класса. Приватное или защищенное MI моделируют отношение *has-a*, в то время как общедоступное MI моделирует отношение *is-a*. Множественное наследование может создавать проблемы с многократно определенными именами и многократно унаследованными базами.

Для разрешения неоднозначности имен можно использовать спецификаторы класса, а для избежания многократно унаследованных баз — виртуальные базовые классы. Однако использование виртуальных базовых классов представляет новые правила с целью записи списков инициализации для конструкторов и при разрешении неоднозначностей.

Шаблоны класса позволяют создать структуру общего класса, в котором тип (обычно тип элемента) представляется параметром-типов. Типичный шаблон выглядит следующим образом:

```
template <class T>
class Ic
{
 T v;
 ...
public:
 Ic(const T & val) : v(val) { }
 ...
};
```

Здесь *T* — параметр-тип, и он действует в качестве заместителя для реального типа, который будет определен позднее. (Этот параметр может иметь любое допустимое имя C++, но *T* и *Type* — общепринятые идентификаторы.) Можно также использовать *typename* вместо *class* в следующем контексте:

```
template <typename T> //то же, что и
//template <class T>
class Rev {...};
```

Определения класса (экземпляры) генерируются при объявлении объекта класса с определением конкретного типа. Например, объявление

```
// неявное создание экземпляра
class Ic<short> sic;
```

заставляет компилятор генерировать объявление класса, в котором каждый случай присутствия параметра-типа *T* в шаблоне заменяется фактическим типом *short* в объявлении класса. В этом случае имя класса — *Ic<short>*, а не *Ic*. *Ic<short>* называется специализацией шаблона. В данном конкретном случае это неявное объявление экземпляра.

Явное создание экземпляра происходит при объявлении конкретной специализации класса с использованием ключевого слова *template*:

```
// явное создание экземпляра
template class IC<int>;
```

В этой ситуации для генерирования специализации *int* класса *Ic<int>* компилятор использует общий шаблон даже при том, что никакие объекты этого класса еще не были запрошены.

Можно обеспечить явные специализации, которые являются специализированными объявлениями класса,

отменяющими определение шаблона. Для этого достаточно определить класс, начав определение с `template<>`, вслед за чем должно следовать имя класса шаблона, сопровождаемое угловыми скобками, содержащими тип, для которого требуется специализация. Например, специализированный класс `Ic` для символьных указателей можно было бы задать следующим образом:

```
template <> class Ic<char *>.
{
 char * str;
 ...
public:
 Ic(const char * s) : str(s) { }
 ...
};
```

Затем объявление в форме

```
class Ic<char *> chic;
```

использовало бы для `chic` специализированное определение, а не общий шаблон.

Шаблон класса может определять более одного общего типа и может также иметь параметры, не являющиеся типом:

```
template <class T, class TT, int n>
class Pals {...};
```

Объявление

```
Pals<double, String, 6> mix;
```

генерировало бы неявное создание экземпляра, используя `double` для `T`, `String` для `TT` и `6` для `n`.

Шаблоны классов могут быть частично специализированными:

```
template <class T> Pals<T, T, 10> { ... };
template <class T, class TT> Pals<T, TT, 100>{...};
template <class T, int n> Pals <T, T*, n> { ... };
```

Первый шаблон задает специализацию, в которой оба типа совпадают, а `n` имеет значение, равное `6`. Аналогично второй создает специализацию для `n`, равного `100`, а третий — специализацию, в которой второй тип является указателем на первый тип.

Цель всех этих методов — обеспечить возможность повторного использования проверенного кода, не прибегая к его копированию вручную. Это упрощает задачу программирования и делает программы более надежными.

## Вопросы для повторения

- Для каждого из следующих наборов классов укажите, какое произведение больше подходит для второго столбца: общедоступное или приватное:

Класс Bear

Класс PolarBear

Класс Kitchen

Класс Home

Класс Person

Класс Programmer

Класс Person

Класс HorseAndJockey

Класс Person, класс Automobile

Класс Driver

- Предположим, что имеются следующие определения:

```
class Frabjous {
private:
 char fab[20];
public:
 Frabjous(const char * s = "C++") :
 fab(s) { }
 virtual void tell() { cout << fab; }
};

class Gloam {
private:
 int glip;
 Frabjous fb;
public:
 Gloam(int g = 0,const char * s = "C++");
 Gloam(int g, const Frabjous & f);
 void tell();
};
```

Считая, что версия `Gloam` функции `tell()` должна отображать значения `glip` и `fb`, обеспечьте определения для трех методов `Gloam`.

- Предположим, что имеются следующие определения:

```
class Frabjous {
private:
 char fab[20];
public:
 Frabjous(const char * s = "C++") :
 fab(s) { }
 virtual void tell() { cout << fab; }
};

class Gloam : private Frabjous{
private:
 int glip;
public:
 Gloam(int g = 0,const char * s = "C++");
 Gloam(int g, const Frabjous & f);
 void tell();
};
```

Считая, что версия `Gloam` функции `tell()` должна отображать значения `glip` и `fab`, обеспечьте определения для трех методов `Gloam`.

- Предположим, что существует следующее определение, основывающееся на шаблоне `Stack` из листинга 13.9 и классе `Worker` из листинга 13.18:

```
Stack<Worker *> sw;
```

Создайте объявление класса, которое будет сгенерировано. Создайте только объявление класса, но не методы класса, являющиеся невстроеннымы.

5. Используйте определения шаблонов, приведенные в этой главе, чтобы определить следующее:

- Массив объектов **String**
- Стек массивов элементов типа **double**
- Массив стеков указателей на объекты **Worker**

6. Описать различия между виртуальными и невиртуальными базовыми классами.

## Упражнения по программированию

1. Класс **Wine** имеет объект класса **String** (см. главу 11), содержащий название вина, и объект класса **ArrayType**, содержащий сведения о количестве доступных бутылок в течение каждого из нескольких последовательных лет. Реализуйте класс **Wine** путем использования метода включения и проверьте его с помощью простой программы. Программа должна запрашивать ввод названия вина, размера массива, первого года для массива и числа бутылок для каждого года. Программа должна использовать эти данные для создания объекта **Wine**, а затем отображать информацию, сохраненную в объекте.

2. Определите шаблон **QueueTp**. Проверьте его, создав очередь указателей-на-**Worker** (как определено в листинге 13.18) и использовав очередь в программе, подобной приведенной в листинге 13.20.

3. Класс **Person** содержит имя и фамилию человека. Кроме конструкторов он включает метод **Show()**, который отображает оба имени. Класс **Gunslinger** получен виртуально из класса **Person**. Он содержит элемент **Draw()**, который возвращает значение типа **double**, представляющее время выхода стрелка на рубеж, выпавшее ему по жребию. Класс также имеет элемент **int**, представляющий количество стволов на ружье стрелка. И наконец, он имеет функцию **Show()**, которая отображает всю эту информацию.

Класс **PokerPlayer** получен виртуально из класса **Person**. Он включает элемент **Draw()**, который возвращает произвольное число в диапазоне от 1 до 52, представляющее значение карты. (Факультативно можно было бы определить класс **Card** с элементами значений масти и достоинства и использовать возвращаемое значение **Card** для **Draw()**). Класс **PokerPlayer** использует функцию **show()** класса **Person**. Класс **BadDude** получен общедоступно из классов **PokerPlayer** и **Gunslinger**. Он имеет элемент **Gdraw()**, который возвращает время, выпавшее по

жребию нежелательному типу, и элемент **Cdraw()**, который возвращает следующую выпавшую карту. Класс имеет соответствующую функцию **Show()**. Определите все эти классы и методы наряду с любыми другими необходимыми методами (типа методов для установки значений объектов) и проверьте их в простой программе, подобной приведенной в листинге 13.20.

4. Имеются некоторые объявления класса:

```
// emp.h - заголовочный файл для класса
// employee и его дочерних объектов
#include <cstring>
#include <iostream>
using namespace std;

const int SLEN = 20;
class employee
{
protected:
 char fname[SLEN];
 char lname[SLEN];
 char job[SLEN];
public:
 employee();
 employee(char * fn, char * ln, char * j);
 employee(const employee & e);
 virtual void ShowAll() const;
 virtual void SetAll(); // запрашивает
 // пользователя о значениях
 friend ostream & operator<<(ostream & os,
 const employee & e);
};

class manager: virtual public employee
{
protected:
 int inchargeof;
public:
 manager();
 manager(char * fn, char * ln,
 char * j, int ico = 0);
 manager(const employee & e, int ico);
 manager(const manager & m);
 void ShowAll() const;
 void SetAll();
};

class fink: virtual public employee
{
protected:
 char reportsto[SLEN];
public:
 fink();
 fink(char * fn, char * ln,
 char * j, char * rpo);
 fink(const employee & e, char * rpo);
 fink(const fink & e);
 void ShowAll() const;
 void SetAll();
};

class highfink: public manager,
 public fink
```

```
{
public:
 highfink();
 highfink(char * fn, char * ln, char * j,
 char * rpo, int ico);
 highfink(const employee & e,
 char * rpo, int ico);
 highfink(const fink & f, int ico);
 highfink(const manager & m, char * rpo);
 highfink(const highfink & h);
 void ShowAll() const;
 void SetAll();
};
```

Обратите внимание, что иерархия класса использует метод множественного наследования с виртуальным базовым классом; поэтому помните о специальных правилах для списков инициализации конструкторов, действующих в этом случае. Заметьте также, что элементы данных объявлены защищенными, а не частными. Это упрощает код для некоторых из методов `highfink`. (Например, обратите внимание, что, если метод `highfink::ShowAll()` просто вызывает методы `fink::ShowAll()` и `manager::ShowAll()`, это приводит к повторному вызову метода `employee::ShowAll()` дважды). Однако можно использовать приватные данные и обеспечить дополнительные защищенные методы в стиле класса `Worker` с использованием MI. Обеспечьте реализации метода класса и проверьте классы в программе. Ниже приведена минимальная тестовая программа. В нее необходимо добавить, по крайней мере, одну проверку функции-элемента `SetAll()`.

```
// useemp1.cpp - использует классы employee
#include <iostream>
using namespace std;
#include "emp.h"

int main()
{
 employee th("Trip", "Harris", "Thumper");
 cout << th << '\n';
 th.ShowAll();
```

```
manager db("Debbie", "Bolt", "Twigger", 5);
cout << db << '\n';
db.ShowAll();

cout << "Press a key for next
 batch of output:\n";
cin.get();

fink mo("Matt", "Oggs", "Oiler",
 "Debbie Bolt");
cout << mo << '\n';
mo.ShowAll();
highfink hf(db, "Curly Kew");
hf.ShowAll();
cout << "Using an employee * pointer:\n";
employee * tri[4] = {&th, &db, &mo, &hf};
for (int i = 0; i < 4; i++)
 tri[i]->ShowAll();

return 0;
}
```

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Symantec C++ требует, чтобы адреса объектов были присвоены элементам массива `tri` индивидуально, а не с помощью оператора инициализации.

Почему никакой оператор присваивания не определен?

Почему `showall()` и `setall()` являются виртуальными?

Почему `employee` — виртуальный базовый класс?

Почему класс `highfink` не имеет никакого раздела данных?

Почему требуется только одна версия `operator<<()`?

Что произойдет, если конец программы будет заменен следующим кодом?

```
employee tri[4] = {th, db, mo, hf};
for (int i = 0; i < 4; i++)
 tri[i].showall();
```

# Дружественные конструкции, исключения и прочее

**В этой главе рассматривается следующее:**

- Дружественные классы
- Методы дружественных классов
- Вложенные классы
- Генерирование исключений, блоки try и catch
- Классы исключений
- Библиотека RTTI
- Операторы dynamic\_cast и typeid
- Операторы static\_cast, const\_cast и reinterpret\_cast

В этой главе подробно рассматриваются дружественные классы, дружественные функции-элементы и вложенные классы. Кроме того, вниманию читателя предлагается описание некоторых новейших дополнений к языку C++, в частности, к ним относятся исключения, RTTI и усовершенствованное управление приведением типов. Обработка исключений в C++ поддерживает механизм преодоления нестандартных ситуаций, которые могут привести к останову выполняющейся программы. Библиотека RTTI (Библиотека типов информации времени выполнения) является механизмом, предназначенным для идентификации объектных типов. Новый тип операторов приведения позволяет осуществлять более безопасное приведение типов. Эти три последние возможности являются совершенно новыми для C++, и многие компиляторы даже не поддерживают их.

## Дружественные структуры

В некоторых примерах, представленных в этой книге, дружественные функции использовались как часть расширенного интерфейса для класса. Подобные функции не являются единственной разновидностью дружественных конструкций, которыми может располагать класс. Класс также может быть дружественной конструкцией. В этом случае любой метод дружественного класса может получить доступ к приватным и защищенным эле-

ментам исходного класса. Следует также учитывать определенные ограничения и разрабатывать только особые функции-элементы для класса. Тогда они могли бы быть дружественными для другого класса. Класс определяет, какие функции, функции-элементы или классы являются дружественными; отношения дружественности не могут накладываться извне. Поэтому, несмотря на то что дружественные конструкции передают вовне доступ к частному разделу класса, они действительно не наносят вреда общему делу объектно-ориентированного программирования. Более того, реализуется более гибкий подход к общедоступному интерфейсу.

## Дружественные классы

Когда же возникает необходимость в том, чтобы один класс являлся дружественным для другого? Рассмотрим пример. Предположим, что вам необходимо смоделировать несложный процесс, связанный с телевидением и дистанционным управлением. При этом определяется класс `Tv`, который представляет телевизоры, и класс `Remote`, включающий устройства дистанционного управления. Ясно, что между этими классами должны быть установлены определенные взаимоотношения, но какие именно? Дистанционное управление не связано непосредственно с телевидением, и, наоборот, не все телевизоры снабжены подобными устройствами, поэтому

нельзя установить взаимоотношения публичного наследования типа *is-a* (*является*). Предполагаемые элементы не являются компонентами друг друга, поэтому нельзя воспользоваться и отношением включения *has-a* (*включает*), а также методом приватного или защищенного наследования. Истина заключается в том, что наличие дистанционного управления может изменить статус телевизора. Поэтому класс **Remote** должен быть дружественной конструкцией для класса **Tv**.

Сначала определим класс **Tv**. Телевизор можно представить с помощью набора элементов состояния, т.е. переменных, описывающих его различные аспекты. Ниже указаны некоторые из возможных состояний:

- Включено/отключено
- Настройка канала
- Установка уровня громкости
- Режим настройки кабеля или антенны
- ТВ-тюнер или вход видеомагнитофона

Режим настройки отражает тот факт, что в США полоса частот между каналами, начиная от 14, неодинакова, что в значительной степени зависит от пропускной способности кабеля и в меньшей степени — от полосы частот УКВ. При выборе устройства ввода доступен вход ТВ (TV) и Видео (VCR). Вход ТВ может быть как кабельным, так и антенным. Некоторые настройки могут поддерживать большее количество вариантов, но этот список является достаточным для целей нашей дальнейшей работы.

Телевизор также имеет такие параметры, которые не описываются переменными состояния. Например, изменяется количество каналов, которые можно принимать с помощью телевизионного приемника, кроме того, можно предусмотреть включение элемента для отслеживания этой величины.

Затем необходимо поддерживать класс с помощью методов, влияющих на эти установки. У многих современных телевизионных приемников элементы управления скрыты за панелями. Однако при работе с большинством телевизионных приемников можно переключать каналы и выполнять подобные операции без использования пульта дистанционного управления. Тем не менее, часто используется возможность одновременной настройки путем плавного перемещения по каналам, но нельзя наугад выбрать канал. Аналогично обычно имеется одна кнопка для увеличения, а другая — для уменьшения громкости звучания.

Дистанционное управление призвано дублировать элементы управления, встроенные в телевизионный приемник. С помощью методов класса **Tv** можно реализовать большое количество методов контроля. Кроме того, дистанционное управление обычно поддер-

живает выделение канала удаленного доступа, т.е. появляется возможность переключаться от канала 2 прямо к каналу 20, минуя промежуточные каналы. Многие элементы дистанционного управления могут функционировать в двух режимах — в качестве телевизионного контроллера и в качестве контроллера VCR.

В результате этих соображений появляется определение, напоминающее приведенное в листинге 14.1. Определение включает несколько констант, которые определены в результате перечислений. Ниже показан оператор, который создает класс **Remote**, используемый в качестве дружественного:

```
friend class Remote;
```

Объявление дружественного класса может осуществляться в общедоступном, приватном или защищенном разделе, нахождение которого не имеет значения. Поскольку класс **Remote** ссылается на класс **Tv**, компилятору должно быть известно о классе **Tv** еще до того, как он приступит к работе с классом **Remote**. Для этого необходимо сначала определить класс **Tv**. В качестве альтернативы можно воспользоваться объявлением, которое указано ниже; эта возможность будет рассмотрена далее.

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Если ваш компилятор не поддерживает тип данных **bool**, воспользуйтесь типом данных **int** (0 и 1) вместо типа данных **bool** (**false** и **true**).

Большая часть методов класса определяются как встроенные. Обратите внимание, что каждый метод **Remote**, в отличие от конструктора, имеет ссылку на объект **Tv** в качестве аргумента. Отсюда следует, что дистанционное управление следует использовать для определенного телевизора. Листинг 14.2 иллюстрирует остальные введенные определения. Функции по установке уровня громкости пошагово изменяют уровень громкости, причем подобное изменение вносится независимо от того, достигла ли громкость минимального либо максимального уровня. Функции выбора канала применяют произвольную выборку. При этом выбор канала с наименьшим номером (1) следует за выбором наибольшего канала (**maxchannel**).

Во многих этих методах при переключении состояний с использованием двух установок применяется условный оператор:

```
void onoff() {state = (state==On)? Off: On;}
```

Поскольку состояния определяются величинами 0 и 1, можно добиться компактности расположения элементов управления, комбинируя поразрядное исключающее ИЛИ и оператор присваивания (^=), который рассматривается в приложении Е:

```
void onoff() {state ^= 1;}
```

## Листинг 14.1 Класс tv.h.

```

// tv.h - классы Tv и Remote

#ifndef _TV_H_
#define _TV_H_

class Tv
{
public:
 friend class Remote; // Remote может получить доступ к приватным разделам Tv
 enum { Off, On } ;
 enum { MinVal, MaxVal = 20 } ;
 enum { Antenna, Cable } ;
 enum { TV, VCR } ;

 Tv(int s = Off, int mc = 100) : state(s), volume(5), maxchannel(mc), channel(2),
 mode(Cable), input(TV) { }
 void onoff() { state = (state == On) ? Off : On; }
 bool ison() const { return state == On; }
 bool volup();
 bool voldown();
 void chanup();
 void chardown();
 void set_mode() { mode = (mode == Antenna) ? Cable : Antenna; }
 void set_input() { input = (input == TV) ? VCR : TV; }
 void settings() const; // отображает все настройки

private:
 int state; // подключено или отключено
 int volume; // требуется численное значение
 int maxchannel; // максимальное число каналов
 int channel; // текущая установка канала
 int mode; // широковещательное или кабельное телевидение
 int input; // TV или VCR
};

class Remote
{
private:
 int mode; // элементы управления TV или VCR
public:
 Remote(int m = Tv::TV) : mode(m) { }
 bool volup(Tv & t) { return t.volup(); }
 bool voldown(Tv & t) { return t.voldown(); }
 void onoff(Tv & t) { t.onoff(); }
 void chanup(Tv & t) { t.chanup(); }
 void chardown(Tv & t) { t.chardown(); }
 void set_chan(Tv & t, int c) { t.channel = c; }
 void set_mode(Tv & t) { t.set_mode(); }
 void set_input(Tv & t) { t.set_input(); }
};

#endif

```

**Листинг 14.2 Программа tv.cpp.**

```
// tv.cpp - методы для класса Tv (методы Remote являются встроенными)
#include <iostream>
using namespace std;
#include "tv.h"

bool Tv::volup()
{
 if (volume < MaxVal)
 {
 volume++;
 return true;
 }
 else
 return false;
}
bool Tv::voldown()
{
 if (volume > MinVal)
 {
 volume--;
 return true;
 }
 else
 return false;
}

void Tv::chanup()
{
 if (channel < maxchannel)
 channel++;
 else
 channel = 1;
}

void Tv::chardown()
{
 if (channel > 1)
 channel--;
 else
 channel = maxchannel;
}

void Tv::settings() const
{
 cout << "TV is " << (state == Off? "Off" : "On") << '\n';
 if (state == On)
 {
 cout << "Volume setting = " << volume << '\n';
 cout << "Channel setting = " << channel << '\n';
 cout << "Mode = "
 << (mode == Antenna? "antenna" : "cable") << '\n';
 cout << "Input = "
 << (input == TV? "TV" : "VCR") << '\n';
 }
}
```

Фактически отдельная переменная без знака типа `char` может хранить до восьми двоичных параметров состояния. Но это уже другая тема, в данном случае возможно использование поразрядных операторов, которые рассматриваются в приложении Е.

Листинг 14.3 включает небольшую программу по тестированию некоторых из описанных выше возможностей. Для управления двумя отдельными телевизорами применяется аналоговый контроллер.

### Листинг 14.3 Программа use\_tv.cpp.

```
//use_tv.cpp
#include <iostream>
using namespace std;
#include "tv.h"

int main()
{
 Tv s20;
 cout << "Initial settings for 20\" TV:\n";
 s20.settings();
 s20.onoff();
 s20.chanup();
 cout << "\nAdjusted settings for 20\" TV:\n";
 s20.settings();

 Remote grey;
 grey.set_chan(s20, 10);
 grey.volup(s20);
 grey.volup(s20);
 cout << "\n20\" settings after
 using remote:\n";
 s20.settings();

 Tv s27(Tv::On);
 s27.set_mode();
 grey.set_chan(s27, 28);
 cout << "\n27\" settings:\n";
 s27.settings();

 return 0;
}
```

Результаты выполнения программы:

```
Initial settings for 20" TV:
TV is Off

Adjusted settings for 20" TV:
TV is On
Volume setting = 5
Channel setting = 3
Mode = cable
Input = TV

20" settings after using remote:
TV is On
Volume setting = 7
Channel setting = 10
Mode = cable
Input = TV

27" settings:
TV is On
Volume setting = 5
```

```
Channel setting = 28
Mode = antenna
Input = TV
```

Ключевым пунктом данного упражнения являются дружественные классы, которые представляют собой естественную конструкцию. В дружественности классов отражаются определенные взаимоотношения. Если не применять дружественность в какой-либо форме, следует создавать приватные разделы общедоступного класса `Tv`. Можно также пойти по пути конструирования более сложного и крупного класса. Этот класс будет включать телевизоры и устройства дистанционного управления. Но подобное решение не отражает того факта, что один и тот же пульт дистанционного управления может применяться при работе с несколькими телевизорами.

### Дружественные функции-элементы

Посмотрев на код для последнего примера, вы заметите, что большинство методов `Remote` реализовано с помощью общедоступного интерфейса для класса `Tv`. Это означает, что данные методы действительно не нуждаются в дружественном статусе. Действительно, единственный метод `Remote`, который обращается непосредственно к приватному элементу класса `Tv`, — `Remote::set_chan()`, поэтому в данном случае мы имеем дело с единственным методом, который нуждается в том, чтобы быть дружественным. Создание выбранных дружественных элементов классов для другого класса является возможным, но это немного неудобно. Необходимо обращать внимание на порядок, в котором упорядочиваются различные объявления и определения. Давайте поговорим о причинах создавшегося положения.

Для того чтобы метод `Remote::set_chan()` стал дружественным для класса `Tv`, необходимо объявить его дружественным в разделе объявления класса `Tv`:

```
class Tv
{
 friend void Remote::set_chan(tv & t, int c);
 ...
};
```

Однако компилятору для обработки этого оператора потребуется уже указанное выше определение `Remote`. Иначе он не будет знать, что `Remote` — это класс и что `set_chan()` является методом данного класса. Это говорит о том, что нужно вставить определение `Remote` раньше определения `Tv`. Но фактически то, что методы `Remote` ссылаются на объекты `Tv`, означает, что определение `Tv` должно задаваться ранее определения `Remote`. В данном разделе используется *опережающее определение*. Обратите внимание, что оператор

```
class Tv; // опережающее определение
```

вставляется раньше определения **Remote**. При этом реализуется следующая структура:

```
class Tv; // опережающее определение
class Remote { ... };
class Tv { ... };
```

Можно ли вместо данной структуры использовать следующую структуру?

```
class Remote; // опережающее определение
class Tv { ... };
class Remote { ... };
```

На этот вопрос будет ответ: "Нет". Дело в том, что, когда компилятор обнаруживает, что метод **Remote** объявляется как дружественный в объявлении класса **Tv**, он должен располагать уже просматриваемым объявлением класса **Remote** вообще и метода **set\_chan()** — в частности.

Другие трудности остаются. В листинге 14.1 объявление **Remote** содержало встроенный код, подобный приведенному ниже:

```
void onoff(Tv & t) { t.onoff(); }
```

Поскольку при этом вызывается метод **Tv**, компилятор должен видеть объявление класса **Tv** на данный момент времени, чтобы знать, какой метод **Tv** имеется в наличии. Но, как вы видели, это объявление обязательно следует за объявлением **Remote**. Решением подобной проблемы должно быть ограничение **Remote** до метода **деклараций** и размещение фактических определений после класса **Tv**. В результате появляется следующая структура:

```
class Tv; //опережающее объявление
class Remote { ... }; //Методы, использующие
 //Tv только в качестве прототипов
class Tv { ... };

// размещение определений метода Remote
```

Прототипы выглядят следующим образом:

```
void onoff(Tv & t);
```

Любой компилятор при просмотре указанного прототипа нуждается в информации о том, что **Tv** — это класс, и опережающие определения предоставляют подобные сведения. К тому времени компилятор достигает раздела фактических определений метода, он уже считывает объявление класса **Tv** и добавляет сведения, которые требуются для выполнения компиляции методов. Используя ключевое слово **inline** в определениях методов, можно создавать встроенные методы. Листинг 14.4 включает скорректированный заголовочный файл.

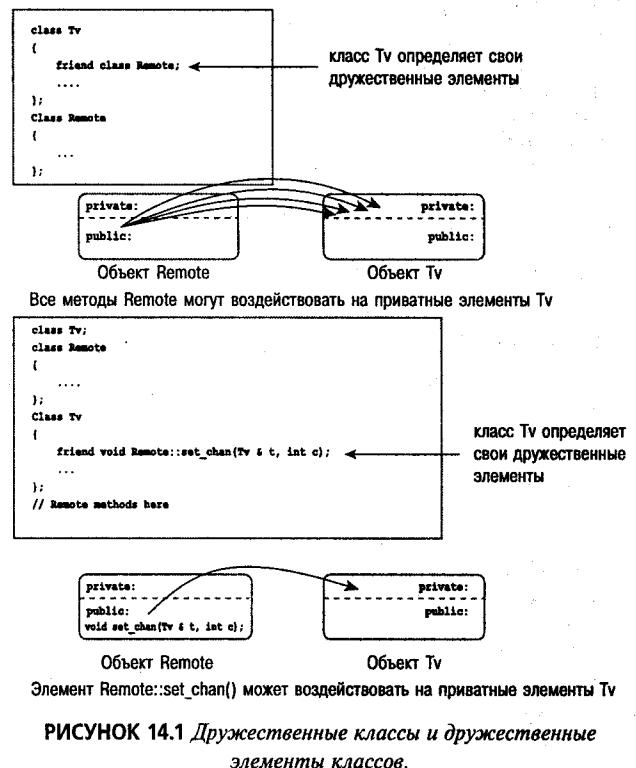
Эта версия ведет себя подобно оригиналу. Разница заключается лишь в том, что метод **Remote** является дружественным по отношению к классу **Tv** в отличие от методов **Remote**. На рис. 14.1 демонстрируется эта разница.

Между прочим, создание целого класса **Remote** в виде дружественного элемента не нуждается в опережающем определении, поскольку дружественный оператор сам идентифицирует **Remote** как класс:

```
friend class Remote;
```

## Другие дружественные отношения

Возможны другие комбинации дружественных элементов и классов. Давайте вкратце рассмотрим некоторые из них. Предположим, что в результате технологического прогресса появились интерактивные устройства дистанционного управления. Например, интерактивное устройство дистанционного управления могло бы позволить вам отвечать на некий вопрос, изложенный в телевизионной программе, и телевизор мог бы генерировать звуковой сигнал в случае неверного ответа. Не останавливаясь на возможностях телевизоров, таких как средства для программирования просмотра телепередачи, давайте рассмотрим аспекты программирования на C++. Новые методы позволили бы получить выгоду из "взаимной дружбы" с некоторыми методами **Remote**, оказывающими влияние на объект **Tv**, как прежде, и с некоторыми методами **Tv**, которые указывающими на объект **Remote**. Это может быть выполнено путем установки взаимных дружественных классов, т.е. класс **Tv** будет дружественным для класса **Remote**, а класс **Remote**



## Листинг 14.4 Класс tvfm.h.

```

// tvfm.h - классы Tv и Remote, использующие дружественный элемент
#ifndef _TVFM_H_
#define _TVFM_H_

class Tv; // опережающее определение
class Remote
{
public:
 enum State{ Off, On};
 enum { MinVal, MaxVal = 20};
 enum { Antenna, Cable};
 enum { TV, VCR};

private:
 int mode;
public:
 Remote(int m = TV) : mode(m) { }
 bool volup(Tv & t); // только прототипы
 bool voldown(Tv & t);
 void onoff(Tv & t);
 void chanup(Tv & t);
 void chardown(Tv & t);
 void set_mode(Tv & t);
 void set_input(Tv & t);
 void set_chan(Tv & t, int c);
};

class Tv
{
public:
 friend void Remote::set_chan(Tv & t, int c);
 enum State{ Off, On};
 enum { MinVal, MaxVal = 20 };
 enum { Antenna, Cable };
 enum { TV, VCR };

 Tv(int s = Off, int mc = 100) : state(s), volume(5),
 maxchannel(mc), channel(2), mode(Cable), input(TV) { }
 void onoff() { state = (state == On) ? Off : On; }
 bool ison() const { return state == On; }
 bool volup();
 bool voldown();
 void chanup();
 void chardown();
 void set_mode() { mode = (mode == Antenna) ? Cable : Antenna; }
 void set_input() { input = (input == TV) ? VCR : TV; }
 void settings() const;
private:
 int state;
 int volume;
 int maxchannel;
 int channel;
 int mode;
 int input;
};

// удаленные методы в качестве встроенных функций
inline bool Remote::volup(Tv & t) { return t.volup(); }
inline bool Remote::voldown(Tv & t) { return t.voldown(); }
inline void Remote::onoff(Tv & t) { t.onoff(); }
inline void Remote::chanup(Tv & t) { t.chanup(); }
inline void Remote::chardown(Tv & t) { t.chardown(); }
inline void Remote::set_mode(Tv & t) { t.set_mode(); }
inline void Remote::set_input(Tv & t) { t.set_input(); }
inline void Remote::set_chan(Tv & t, int c) { t.channel = c; }
#endif

```

будет дружественным для класса `Tv`. Имейте в виду, что метод `Tv`, который использует объект `Remote`, может быть прототипирован перед объявлением класса `Remote`, но должен быть объявлен после объявления так, чтобы компилятор имел достаточно информации для выполнения компиляции метода. При этом мы получаем следующую структуру:

```
class Tv
{
 friend class Remote;
public:
 void buzz(Remote & r);
 ...
};

class Remote
{
 friend class Tv;
public:
 void Bool volup(Tv & t) { t.volup(); }
 ...
};

inline void Tv::buzz(Remote & r)
{
 ...
}
```

Поскольку объявление класса `Remote` следует за объявлением класса `Tv`, метод `Remote::volup()` может быть определен в объявлении класса. Однако метод `tv::buzz()` должен быть определен вне объявления `Tv` так, чтобы определение могло следовать за объявлением класса `Tv`. Если вы не хотите, чтобы `buzz()` был встроенным, определите его в отдельном файле определений метода.

### Общедоступные дружественные элементы

Дружественные элементы применяются и в том случае, когда функция нуждается в доступе к приватным данным в двух различных классах. Логически такая функция должна быть функцией-элементом каждого класса, но это невозможно. Это может быть элемент одного класса и дружественный элемент для другого класса, но иногда более разумно создавать дружественные функции для обоих классов. Например, предположим, что у вас имеется класс `Probe`, представляющий некоторый вид программируемого измерительного устройства, и класс `Analyzer`, представляющий собой некий вид программируемого анализатора. Каждое устройство имеет внутренние часы, и вам потребуется синхронизировать показания часов. В этом случае вы можете реализовать все сказанное в следующих строках:

```
class Analyzer; // опережающее определение
class Probe
{
 friend void sync(Analyzer & a,
 const Probe & p); //синхронизация а и р
```

```
friend void sync(Probe & p,
 const Analyzer & a); //синхронизация р и а
 ...
};

class Analyzer
{
 friend void sync(Analyzer & a,
 const Probe & p); //синхронизация а и р
 friend void sync(Probe & p,
 const Analyzer & a); //синхронизация р и а
 ...
};

// определение дружественных функций
inline void sync(Analyzer & a, const Probe & p)
{
 ...
}
inline void sync(Probe & p, const Analyzer & a)
{
 ...
}
```

Опережающее определение дает возможность компилятору знать, что `Analyzer` — тип, появляющийся при достижении дружественных объявлений в объявлении класса `Probe`.

### Шаблоны и дружественные элементы

Шаблоны объявлений класса также могут включать дружественные элементы. Мы можем разбить дружественные элементы по шаблонам на три категории:

- Нешаблонные дружественные элементы
- Связанные шаблоны дружественных элементов. При этом подразумевается, что тип дружественного элемента определяется типом класса при образовании класса
- Несвязанные шаблоны дружественных элементов. При этом подразумевается, что все реализации дружественного элемента являются дружественными для каждой реализации класса.

Давайте рассмотрим примеры. Вот пример шаблона с нешаблонными дружественными элементами:

```
template <class Type>
class HasFriend
{
 friend void date(); //дружественный
 //элемент для всех реализаций HasFriend
};
```

Функция `date()` будет дружественным элементом для всех возможных реализаций шаблона. Например, она может быть дружественным элементом для класса `HasFriend<int>` и `HasFriend<String>`.

Предположим, что требуется поддерживать аргумент шаблона класса для дружественного элемента. Можете

ли вы видоизменять объявление дружественного элемента подобно следующему:

```
friend void date(HasFriend &); // возможно?
```

Ответ на этот вопрос будет звучать отрицательно. Причина заключается в том, что не существует такой вещи, как объект **HasFriend**. Имеются только специфические реализации типа **HasFriend<short>**. Чтобы обеспечивать шаблон аргумента класса, нужно указать реализацию. Например, можно сделать следующее:

```
template <class Type>
class HasFriend
{
 //связанный шаблон дружественного элемента
 friend void date(HasFriend<Type> &);
 ...
};
```

При этом потребуется определение шаблона функции:

```
template <class A>
void date (A &) {...};
```

Эта комбинация кода образует функцию **date<short>()**, которая является дружественной по отношению к **HasFriend<short>**. Каждая реализация класса имеет одну соответствующую дружескую функциональную реализацию. Это пример связанных дружественных отношений между шаблонами. Та же самая методика может использоваться для того, чтобы создать связанные шаблоны дружественных классов.

Теперь предположим, что немного изменяется объявление класса:

```
template <class Type>
class HasFriend
{
 //несвязанный шаблон дружественного элемента
 friend void date(HasFriend<T> &);
 ...
};
```

Оригинал использует то же самое обобщенное название типа (**Type**) в заголовке класса и в прототипе дружественного элемента. Эта версия использует различное обобщенное название типа (**T**) для прототипа. Эффект от использования этого формата проявляется в том, что все реализации из **date()** будут дружественными элементами для каждой реализации **HasFriend**, т.е. **date<int>()**, **date<double>()** и **date<HasFriend<String>>()** являются дружественными для **HasFriend <char \*>**. Это пример несвязанной шаблонной дружбы. Та же самая методика может использоваться при создании несвязанных шаблонов дружественных классов.

Могут ли шаблоны быть дружественными элементами по отношению к нешаблонным классам? Нет, не могут, но определенные реализации имеют право на существование:

```
class Pal
{
 friend class HasFriend<long>; // верно
 friend class HasFriend; // не допускается
};
```

Причина заключается в том, что не производные имена шаблонов, подобные **HasFriend**, могут отображаться только в шаблонах. Обычный программный код может использовать определенные реализации типа **HasFriend<Complex>**.

## Вложенные классы

В C++ можно размещать объявление класса внутри другого класса. Класс, объявленный внутри другого, называется **вложенным классом**. Это помогает избегать беспорядка в названиях при предоставлении диапазона доступа новым типам классов. Функции-элементы класса, содержащего объявление, могут создавать и использовать объекты вложенного класса. Внешний мир может использовать вложенный класс только в том случае, если объявление находится в общедоступном разделе и если используется оператор определения диапазона доступа. (Однако устаревшие версии C++ не допускают использования вложенных классов или обеспечивают выполнение описанной выше концепции не в полной мере.)

Реализация вложения классов неэквивалентна выполнению включения. Как вы помните, включение означает наличие объекта класса в качестве элемента другого класса. С другой стороны, в процессе вложения классов не создается элемент класса. Вместо этого определяется тип, который известен локально классу, содержащему вложенное объявление класса.

Обычные причины для вложения класса состоят в том, чтобы помочь реализации другого класса и избегать конфликтов названий. Например, класс **Queue** (см. главу 11, листинг 11.11) представляет собой замаскированный случай вложенных классов, когда вкладывается определение структуры:

```
class Queue
{//определения диапазона класса
//Node — это определение вложенной
//структурой, локальной для данного класса
struct Node {Item item; struct Node * next;};
...};
```

Поскольку структура — это класс, элементы которого являются общими по умолчанию, **Node** на самом деле является вложенным классом. Однако в таком определении не учитываются преимущества возможностей класса. В частности, испытывается недостаток явного конструктора. Давайте исправим этот недостаток.

Сначала посмотрим, где создаются объекты **Node** в примере **Queue**. Исследование объявления класса (лис-

тинг 11.11) и определений методов (листинг 11.12) показывает, что объекты **Node** создаются методом **enqueue()**:

```
bool Queue::enqueue(const Item & item)
{
 if (isfull())
 return false;
 Node * add = new Node;//инициализация узла
 if (add == NULL)
 return false; //выход, если нет
 //доступного конструктора
 add->item = item; //установка указателей
 //узла
 add->next = NULL;
 ...
}
```

Здесь в программном коде явно присваиваются значения элементам **Node** после инициализации этого узла. Этот вид работы лучше выполняется конструктором.

Зная теперь, где и как конструктор должен использоваться, можно обеспечивать соответствующее определение конструктора:

```
class Queue
{ // определения диапазона класса
// Node является определением вложенного
// класса, локального для данного класса
class Node
{
public:
 Item item;
 Node * next;
 Node(const Item & i) :
 item(i), next(0) { }
 ...
};
```

Этот конструктор инициализирует элемент **item** узла значением **i**, а затем присваивает указателю **next** значение **0**, что представляет собой еще один способ задания нулевого указателя в C++. (Использование указателя **NULL** потребовало бы включения заголовочного файла, который его определяет). Поскольку все узлы, созданные классом **Queue**, включают указатель **next**, которому изначально присвоено нулевое значение, это единственный конструктор, в котором нуждается класс.

Затем перепишите метод **enqueue()**, используя конструктор:

```
bool Queue::enqueue(const Item & item)
{
 if (isfull())
 return false;
 // инициализация узла
 Node * add = new Node(item);
 if (add == 0)
 return false; //выход, если
 //обработка завершена
 ...
}
```

Благодаря этому уменьшается объем программного кода для **enqueue()**, который становится более безопасным (поскольку автоматическая инициализация гораздо лучше, чем инициализация вручную).

В этом примере определяется конструктор в объявлении класса. Предположим, что вместо этого конструктор определяется в файле методов. Определение должно отразить то, что класс **Node** определен в классе **Queue**. Это реализуется путем двукратного использования оператора определения диапазона доступа:

```
Queue::Node::Node(const Item & i) :
 item(i), next(0) { }
```

## Вложенные классы и доступ

Вложенным классам соответствуют два типа доступа. Сначала в том месте, где объявлен вложенный класс, проверяется диапазон доступа вложенного класса, т.е. устанавливается, какие части программы могут создавать объекты этого класса. Затем, как и в случае с любым другим классом, общедоступные, защищенные и приватные разделы вложенного класса обеспечивают управление доступом к элементам класса. Где и как вложенный класс может использоваться, зависит и от диапазона доступа, и от управления доступом. Давайте исследуем эти вопросы более подробно.

## Диапазон доступа

Если вложенный класс объявлен в приватном разделе второго класса, об этом известно только данному классу. Это справедливо, например, в случае с классом **Node**, который вложен в раздел объявления **Queue** в последнем примере. (Может показаться, что класс **Node** был определен перед приватным разделом, но вспомните о том, что приватный доступ установлен по умолчанию для классов.) Следовательно, элементы класса **Queue** могут использовать объекты класса **Node** и указатели на объекты **Node**, но другие части программы не будут даже знать о том, что класс **Node** существует. Если вам нужно получить производный класс из класса **Queue**, класс **Node** будет также невидим для полученного класса, поскольку производный класс не может непосредственно обращаться к приватным разделам базового класса.

Если вложенный класс объявлен в защищенном разделе второго класса, он является видимым для этого класса, но невидимым для внешнего мира. Однако в этом случае производный класс знал бы о наличии вложенного класса и мог бы непосредственно создавать объекты этого типа.

Если вложенный класс объявлен в общедоступном разделе второго класса, он доступен второму классу, классам, производным от второго класса, и, поскольку

он является общедоступным, — также и внешнему миру. Однако, поскольку вложенный класс располагает определенным диапазоном доступа, он должен использоваться со спецификатором класса, благодаря чему обеспечивается доступ извне. Например, предположим, что вы располагаете следующим объявлением:

```
class Team
{
public:
 class Coach { ... };
 ...
};
```

Теперь предположим, что имеется свободный тренер, который не входит в состав ни одной команды. Чтобы создать объект `Coach` вне класса `Team`, можно выполнить следующее:

```
Team::Coach forhire; //создание объекта
 //Coach вне класса Team
```

Те же самые соображения относительно диапазонов доступа используются при рассмотрении вложенных структур, а также перечислений. Действительно, многие программисты используют общие перечисления для поддержки констант класса, которые могут использоваться программистами клиентских приложений. Например, многие реализации классов, определенные с учетом поддержки возможности `iostream`, используют эту методику, чтобы обеспечить различные опции форматирования, которых мы коснулись ранее, а более подробно рассмотрим в главе 16. В табл. 14.1 перечислены свойства диапазонов доступа для вложенных классов, структур и перечислений.

### Управление доступом

После того как класс находится в диапазоне доступа, вступает в игру управление доступом. Доступ ко вложенному классу регламентируют те же самые правила, которые регулируют доступ к обычному классу. Объявление класса `Node` в объявлении класса `Queue` не предоставляет классу `Queue` никаких специальных привилегий доступа к классу `Node`. Также классу `Node` не предоставляются специальные привилегии доступа к классу `Queue`. Таким образом, объект класса `Queue` мо-

жет обращаться только к общедоступным элементам объекта `Node` явным образом. По этой причине, например, `Queue` может создать все элементы из общедоступного класса `Node`. Это нарушает обычную практику создания приватных компонентов данных, но класс `Node` является внутренней реализацией свойства класса `Queue` и невидим для внешнего мира. Дело в том, что класс `Node` объявлен в приватном разделе класса `Queue`. Таким образом, хотя методы `Queue` могут обращаться к элементам `Node` непосредственно, клиент, использующий класс `Queue`, этого делать не может.

Одним словом, место нахождения объявления класса определяет диапазон доступа (область видимости). Учитывая, что специфический класс находится в диапазоне доступа, обычные правила управления доступом (общий, защищенный, частный, дружественный) определяют доступ к программе, содержащей элементы вложенного класса.

### Вложение в шаблоне

Как вы уже видели раньше, шаблоны предлагают неплохой метод для реализации контейнерных классов типа класса `Queue`. У вас может появиться вопрос, возникают ли проблемы при преобразовании определения класса `Queue` в шаблон из-за наличия вложенного класса? Нет, проблем в этом случае не возникает. На примере листинга 14.5 можно увидеть, каким образом выполняется упомянутое преобразование. Как обычно в случае с шаблонами класса, заголовочный файл включает шаблон класса наряду с шаблонами методов функций.

Относительно данного шаблона можно выделить один интересный момент, который заключается в том, что `Node` определен с помощью универсального типа `Item`. Таким образом, объявление

```
QueueTp<double> dq;
```

приводит к тому, что `Node` определяется для хранения типов значений `double`, в то время как

```
QueueTp<char> cq;
```

приводит к тому, что `Node` определяется для хранения типа значений `char`. Эти два класса `Node` определены в

**Таблица 14.1 Свойства диапазонов доступа для вложенных классов, структур и перечислений.**

| Место объявления<br>во вложенном классе | Доступность для<br>вложенного класса | Доступность для классов,<br>которые производны<br>от вложенного класса | Доступность для<br>внешнего мира          |
|-----------------------------------------|--------------------------------------|------------------------------------------------------------------------|-------------------------------------------|
| Приватный раздел                        | Да                                   | Нет                                                                    | Нет                                       |
| Защищенный раздел                       | Да                                   | Да                                                                     | Нет                                       |
| Общедоступный раздел                    | Да                                   | Да                                                                     | Да, совместно с классификатором<br>класса |

## Листинг 14.5 Класс queueTP.h.

```

// queueTP.h - шаблон очереди вместе с вложенным классом
template <class Item>
class QueueTP
{
private:
 enum { Q_SIZE = 10} ;
 // Node - это определение вложенного класса
 class Node
 {
public:
 Item item;
 Node * next;
 Node(const Item & i):item(i), next(0){ }
 };
 Node * front; // указатель на начало Queue
 Node * rear; // указатель на конец Queue
 int items; // текущее количество элементов в Queue
 const int qsize; // максимальное число элементов в Queue
 QueueTP(const QueueTP & q) : qsize(0) { }
 QueueTP & operator=(const QueueTP & q) { return *this; }
public:
 QueueTP(int qs = Q_SIZE);
 ~QueueTP();
 bool isempty() const
 {
 return items == 0;
 }
 bool isfull() const
 {
 return items == qsize;
 }
 int queuecount() const
 {
 return items;
 }
 bool enqueue(const Item &item); // добавить элемент в конец
 bool dequeue(Item &item); // удалить элемент из начала
};

// методы QueueTP
template <class Item>
QueueTP<Item>::QueueTP(int qs) : qsize(qs)
{
 front = rear = 0;
 items = 0;
}

template <class Item>
QueueTP<Item>::~QueueTP()
{
 Node * temp;
 while (front != 0) // пока очередь еще не пустая
 {
 temp = front; // сохранение адреса первого элемента
 front = front->next; // переустановка указателя к следующему элементу
 delete temp; // удаление прежнего первого элемента
 }
}

// добавление элемента в очередь
template <class Item>
bool QueueTP<Item>::enqueue(const Item & item)

```

```

 if (isfull())
 return false;
 Node * add = new Node(item); // инициализация узла
 if (add == NULL)
 return false; // выход, если недоступно
 items++;
 if (front == 0) // если очередь пустая
 front = add; // элемент размещается спереди
 else
 rear->next = add; // иначе элемент размещается в конце
 rear = add; // имеется точка в конце для нового узла
 return true;
}

// Размещение первого элемента в переменной item и устранение из очереди
template <class Item>
bool QueueTP<Item>::dequeue(Item & item)
{
 if (front == 0)
 return false;
 item = front->item; // присваивание item первого элемента очереди
 items--;
 Node * temp = front; // сохранение первого элемента в прежнем местоположении
 front = front->next; // переустановка начала к следующему элементу
 delete temp; // удаление прежнего первого элемента
 if (items == 0)
 rear = 0;
 return true;
}

```

#### Листинг 14.6 Программа nested.cpp.

```

// nested.cpp - использование очереди, имеющей вложенный класс
// Компиляция вместе с strng2.cpp

#include <iostream>
using namespace std;
#include "strng2.h"
#include "queuetp.h"

int main()
{
 QueueTP<String> cs(5);
 String temp;

 while(!cs.isfull())
 {
 cout << "Please enter your name. You will be "
 "served in the order of arrival.\n"
 "name: ";
 cin >> temp;
 cs.enqueue(temp);
 }
 cout << "The queue is full. Processing begins!\n";

 while (!cs.isEmpty())
 {
 cs.dequeue(temp);
 cout << "Now processing " << temp << "... \n";
 }
 return 0;
}

```

двух различных классах `QueueTP`, благодаря чему не возникает конфликтов наименований. Иначе говоря, один узел будет иметь тип `QueueTP<double>::Node`, а другой — `QueueTP<char>::Node`.

Листинг 14.6 предлагает небольшую программу для тестирования нового класса. Создается очередь объектов типа `String`, поэтому нужно выполнить компиляцию вместе с программой `strng2.cpp` (см. главу 11).

В результате выполнения программы из листинга 14.6 получаются следующие результаты:

```
Please enter your name. You will be served
in the order of arrival.
name: Kinsey Millhone
Please enter your name. You will be served
in the order of arrival.
name: Adam Dalgliesh
Please enter your name. You will be served
in the order of arrival.
name: Andrew Dalziel
Please enter your name. You will be served
in the order of arrival.
name: Kay Scarpetta
Please enter your name. You will be served
in the order of arrival.
name: Richard Jury
The queue is full. Processing begins!
Now processing Kinsey Millhone...
Now processing Adam Dalgliesh...
Now processing Andrew Dalziel...
Now processing Kay Scarpetta...
Now processing Richard Jury...
```

## Исключения

Программы во время выполнения иногда сталкиваются с проблемами, препятствующими их нормальному функционированию. Например, программа может пытаться открыть недоступный файл или запрашивать больший объем памяти, чем тот, который доступен, или может сталкиваться со значениями, которые она не может обработать. Обычно программисты стараются предупредить такие неприятности. Исключения C++ обеспечивают мощный и гибкий инструмент для программистов, имеющих дело с такими ситуациями. Исключения были добавлены к C++ недавно, и еще не все компиляторы успели их реализовать.

Перед исследованием исключений давайте посмотрим на некоторые более элементарные возможности, доступные программисту. В качестве теста рассмотрим функцию, которая вычисляет гармоническое среднее двух чисел. Гармоническое среднее двух чисел определяется как обратное от среднего арифметического обратных этим числам значений. Все это может быть сведено к следующему выражению:

$$2.0 * x * y / (x + y)$$

Обратите внимание, что, если у является отрицанием `x`, эта формула приводит к ситуации деления на нуль. Один из способов обработки этой ситуации — использование функции аварийного завершения выполнения программы `abort()` (в случае, если один параметр является отрицанием другого параметра). Прототип функции `abort()` находится в заголовочном файле `cstdlib` (или `stdlib.h`). Как правило, в случае вызова эта функция помещает сообщение типа "abnormal program termination" ("аномальное завершение программы") в поток стандартных ошибок (этот поток аналогичен потоку, используемому при обработке ошибок C) и завершает выполнение программы. Она также возвращает зависящее от выполнения значение, указывающее на имеющиеся проблемы, операционной системе или, если программа была инициализирована другой программой, основному процессу. Очищает ли функция `abort()` буферы файловых потоков (области памяти, обычно хранящие данные для передачи файлам и обратно), зависит от ее реализации в данной системе. Если хотите, можете использовать функцию `exit()`, которая очищает буферы файлов, но не отображает сообщения. Листинг 14.7 демонстрирует пример небольшой программы, использующей функцию `abort()`.

### Листинг 14.7 Программа error1.cpp.

---

```
//error1.cpp - использование функции abort()
#include <iostream>
using namespace std;
#include <cstdlib>
double hmean(double a, double b);

int main()
{
 double x, y, z;
 cout << "Enter two numbers: ";
 while (cin >> x >> y)
 {
 z = hmean(x,y);
 cout << "Harmonic mean of " << x
 << " and " << y
 << " is " << z << "\n";
 cout << "Enter next set of numbers
 ^q to quit: ";
 }
 cout << "Bye!\n";
 return 0;
}

double hmean(double a, double b)
{
 if (a == -b)
 {
 cout << "untenable arguments
 ^to hmean()\n";
 abort();
 }
 return 2.0 * a * b / (a + b);
}
```

---

Результаты выполнения программы из листинга 14.7:

```
Enter two numbers: 3 6
Harmonic mean of 3 and 6 is 4
Enter next set of numbers <q to quit>: 10 -10
untenable arguments to hmean()
abnormal program termination
```

Обратите внимание, что при вызове функции `abort()` из функции `hmean()` завершение работы программы происходит непосредственно без возврата к функции `main()`.

Программа могла бы избежать прерывания, проверяя значения `x` и `y` перед вызовом функции `hmean()`. Однако недостаточно возложить на программу заботу о выполнении подобной проверки.

Более гибкий подход, чем прерывание выполнения, должен состоять в использовании возвращаемого значения функции, чтобы определить проблему. Например, элемент класса `ostream`, `get(void)` обычно возвращает код ASCII для следующего символа ввода. Он возвращает специальное значение `EOF`, если встречает конец файла.

ла. Этот подход неприменим в случае с функцией `hmean()`.

Любое численное значение может быть допустимым возвращаемым значением, поскольку нет специального значения, используемого для локализации проблемы. В такой ситуации можно использовать аргумент указателя или аргумент ссылки, чтобы вернуть значение вызывающей программе и использовать возвращаемое значение функции для указания на успешное или неудачное завершение выполнения. Семейство перегруженных операторов `istream` использует один из вариантов этой методики. Сообщая вызывающей программе об успехе или неудаче, вы даете программе возможность не прерывать выполнение, а выбирать иные действия. Листинг 14.8 демонстрирует пример реализации этого подхода. Здесь переопределяется `hmean()` как булевская функция, возвращающее значение которой указывает на успех или неудачу. Для получения ответа используется третий аргумент.

#### Листинг 14.8 Программа error2.cpp.

```
//error2.cpp - возврат кода ошибки
#include <iostream>
using namespace std;
#include <cfloat> // (или float.h) для DBL_MAX
bool hmean(double a, double b, double * ans);
int main()
{
 double x, y, z;

 cout << "Enter two numbers: ";
 while (cin >> x >> y)
 {
 if (hmean(x,y,&z))
 cout << "Harmonic mean of " << x << " and " << y << " is " << z << "\n";
 else
 cout << "One value should not be the negative " << "of the other - try again.\n";
 cout << "Enter next set of numbers <q to quit>: ";
 }
 cout << "Bye!\n";
 return 0;
}

bool hmean(double a, double b, double * ans)
{
 if (a == -b)
 {
 *ans = DBL_MAX;
 return false;
 }
 else
 {
 *ans = 2.0 * a * b / (a + b);
 return true;
 }
}
```

Результаты выполнения программы из листинга 14.8:

```
Enter two numbers: 3 6
Harmonic mean of 3 and 6 is 4
Enter next set of numbers <q to quit>: 10 -10
One value should not be the negative of the
other - try again.
Enter next set of numbers <q to quit>: 1 19
Harmonic mean of 1 and 19 is 1.9
Enter next set of numbers <q to quit>: q
Bye!
```

### Примечания к программе

Таким образом, проект программы обеспечивает пользователю возможность продолжения работы, даже несмотря на ввод некорректных результатов. Конечно, проект предполагает, что пользователь проверит значение возвращаемого результата функции, правда, пользователи не всегда это делают. Например, чтобы не усложнять примеры программ, в большинстве листингов этой книги не предполагается проверка того, что оператор `new` возвращает нулевой указатель либо оператор `cout` успешно обрабатывает выводимые результаты.

Можно использовать либо указатель, либо ссылку на посторонние аргументы. Многие программисты для аргументов встроенных типов предпочитают пользоваться указателями, поскольку это делает очевидным использование какого-либо аргумента для ответа.

### Механизм исключений

Теперь давайте посмотрим, как вы можете решить проблемы с помощью механизма исключений. Исключение C++ генерируется в результате нестандартных ситуаций, возникающих при выполнении программ (например, попытка деления на нуль). С помощью исключений можно передавать управление от одной части программы к другой. Обработка исключений состоит из трех этапов:

- Генерация исключения
- Перехват исключения обработчиком исключений
- Использование блока `try`

Исключение вызывается при возникновении проблемы. Например, вы можете изменить функцию `hmean()` в листинге 14.7, вызвав исключение вместо функции `abort()`. Оператор генерации исключений, по сути, является оператором перехода, т.е. он дает команду программе перейти к операторам, находящимся в другом месте программы. Ключевое слово `throw` указывает на генерацию исключения. За ним следует значение, такое как символьная строка или объект, указывающее на природу исключения. Обработка исключений осуществляется с помощью *обработчика исключений* в том месте программы, где возникает проблема. Ключевое слово

`catch` указывает на перехват исключения. Код обработчика начинается ключевым словом `catch`, за которым следует в круглых скобках тип декларации, указывающий на соответствующий ему тип исключения. Далее следует заключенный в фигурные скобки код, указывающий на предпринимаемые действия. Ключевое слово `catch` наряду с типом исключения служит меткой, обозначающим точку в программе, которой должно передаваться управление при вызове исключения. Обработчик исключений называется еще *блоком catch*.

Блок `try` определяет блок кода, при выполнении которого активизируются некоторые исключения. За ним следует один или несколько блоков `catch`. На блок `try` указывает ключевое слово `try`, за которым следует заключенный в фигурные скобки фрагмент кода, указывающий на код, к которому относятся исключения.

Проще всего продемонстрировать взаимодействие трех описанных элементов с помощью небольшого примера, показанного в листинге 14.9.

Результаты выполнения программы из листинга 14.9:

```
Enter two numbers: 3 6
Harmonic mean of 3 and 6 is 4
Enter next set of numbers <q to quit>: 10 -10
bad hmean() arguments: a = -b not allowed
Enter a new pair of numbers: 1 19
Harmonic mean of 1 and 19 is 1.9
Enter next set of numbers <q to quit>: q
Bye!
```

### Примечания к программе

Блок `try` выглядит так:

```
try { // начало блока try
 z = hmean(x,y);
} // конец блока try
```

Если какой-либо из операторов в этом блоке выполняет генерацию исключения, то блоки `catch`, размещенные после этого блока, будут выполнять обработку исключений. Если программа вызвала функцию `hmean()` где-нибудь вне этого (или любого другого) блока `try`, она не сможет обработать исключение.

Генерация исключений выглядит следующим образом:

```
if (a == -b)
 throw "bad hmean() arguments:
 \"a = -b not allowed\"";
```

В этом случае генерация исключения выглядит как строка `"bad hmean() arguments: a = -b not allowed"`. Генерация исключений напоминает выполнение оператора возврата тем, что завершает выполнение функции. Однако, вместо того чтобы вернуть управление вызывающей программе, оператор генерирования приводит к тому, что программа дублируется через последовательность текущих вызовов функций до тех пор, пока не обнаружит функцию, содержащую блок `try`.

## Листинг 14.9 Программа error3.cpp

```

//error3.cpp
#include <iostream>
using namespace std;
double hmean(double a, double b);

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

 cout << "Enter two numbers: ";
 while (cin >> x >> y)
 {
 try { // начало блока try
 z = hmean(x,y);
 } // конец блока try
 catch (char * s) // начало обработчика исключений
 {
 cout << s << "\n";
 cout << "Enter a new pair of numbers: ";
 continue; // конец обработчика исключений
 }
 cout << "Harmonic mean of " << x << " and " << y
 << " is " << z << "\n";
 cout << "Enter next set of numbers <q to quit>: ";
 }
 cout << "Bye!\n";
 return 0;
}

double hmean(double a, double b)
{
 if (a == -b)
 throw "bad hmean() arguments: a = -b not allowed";
 return 2.0 * a * b / (a + b);
}

```

В листинге, 14.9 эта функция аналогична функции вызова. Далее вы найдете пример, демонстрирующий дублирование более чем одной функции. Между тем, в данном случае генерация возвращает управление программой обратно функции `main()`. Теперь программе требуется обработчик исключений (следующий за блоком `try`), подходящий для типа генерируемого исключения.

Обработчик исключений, или блок `catch`, выглядит так:

```

catch (char * s)//начало обработчика исключений
{
 cout " s " "\n";
 cout " Enter a new pair of numbers: ";
 continue;
} // конец обработчика

```

Это немного похоже на определение функции, но это только на первый взгляд. Ключевое слово `catch` определяет эту конструкцию как обработчик исключений, а символ `* s` обозначает, что этот обработчик соответствует генерируемому исключению, т.е. строке. Данное объявление `s` во многом напоминает определение аргумента функции (схожесть заключается в методе присваивания

для `s` подходящего генерируемого исключения). Следует также отметить, что, если исключение соответствует данному обработчику, программа выполняет код в скобках.

Если программа завершает выполнение операторов в блоке `try` без генерации каких-либо исключений, она пропускает блок `catch` или блоки, следующие за блоком `try`, и вызывает первый оператор, который следует за обработчиками. Поэтому, когда в программном примере обрабатываются величины 3 и 6, выполнение программы передается непосредственно заключительному оператору, содержащему результат.

Давайте проследим, как результаты в примере, следующие за величинами 10 и -10, передаются функции `hmean()`. В результате выполнения оператора проверки `if` функция `hmean()` генерирует исключение, — и выполнение этой функции завершается. При обратном поиске программы определяет, что функция `hmean()` была вызвана из блока `try` функцией `main()`. Затем программа ищет блок `catch` такого типа, который бы подходил к типу исключения. Единственный имеющийся блок `catch` располагает параметром `char *`, поэтому он и подходит в этой ситуации. Обнаружив соответствие, программа

присваивает строку "bad hmean() arguments: a = -b not allowed" переменной s. Далее программа выполняет код обработчика. Сначала она выводит значение переменной s, соответствующее перехваченному исключению. Затем она выводит пользовательские инструкции, содержащие информацию о том, как следует вводить новые данные. В итоге программа выполняет оператор `continue`, в результате чего пропускаются оставшиеся операторы цикла `while` и осуществляется переход к началу программы. То, что оператор `continue` возвращает программу к началу цикла, свидетельствует, что операторы обработчика являются частью цикла и что перехватываемая строка ведет себя как метка, указывающая на процесс выполнения программы (рис. 14.2).

Вам, должно быть, интересно, что происходит, если функция генерирует исключение и если нет блока `try` или еще какого-нибудь подходящего обработчика исключений. По умолчанию программа в конечном итоге вызывает функцию `abort()`, но этот процесс можно изменить. К этой теме мы вернемся позже.

## Разносторонность исключений

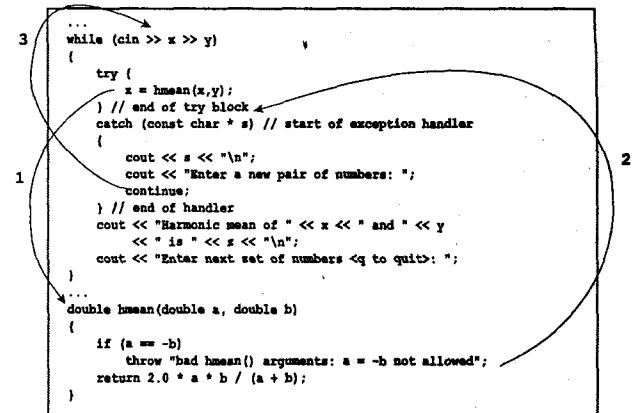
Исключения C++ являются весьма разнообразными, а потому с помощью блока `try` можно определить, какой именно код проверяется для исключений, а с помощью обработчика исключений можно установить, что сделано. Например, в листинге 14.9 блок `try` находился внутри цикла, поэтому программа продолжает выполнять цикл после обработки исключения. Если разместить цикл внутри блока `try`, можно добиться того, что исключение будет выполняться за пределами цикла, завершая, таким образом, цикл. Это демонстрируется в листинге 14.10. Там же указывается на два дополнительных момента:

- Вы можете уточнить определение функции определением исключения, чтобы указать, какие виды исключений эта функция генерирует.
- Блок `catch` может обработать более одного источника исключений.

Для уточнения прототипа функции в целях указания видов исключений, генерируемых этой функцией, добавьте определение исключений, состоящее из ключевого слова `throw` и следующего за ним через запятую в скобках списка типов исключений:

```
double hmean(double a, double b) throw(char *);
```

Этим самым выполняются два действия. Во-первых, компилятору передается информация о том, какого рода исключение или исключения генерируются функцией. Если функция генерирует какой-либо другой вид исключений, реакцией программы на подобное произошедшее



1. Программа вызывает `hmean()` в составе блока `try`.
2. Программа `hmean()` вызывает исключение, передает исключение блоку `catch` и присваивает строку исключения переменной `s`.
3. Блок `catch` передает исключение обратно в цикл `while`.

РИСУНОК 14.2 Выполнение программы с исключениями.

будет (в конечном итоге) вызов функции `abort()`. (Более детально этот вариант действий и способы его изменения мы рассмотрим далее). Во-вторых, использование спецификации исключений позволяет любому пользователю, изучающему прототип, сделать вывод, что именно эта функция генерирует исключения. При этом читатель получает напоминание о том, что он или она может предусмотреть использование блока `try` или обработчика исключений. Функции, генерирующие более одного вида исключений, могут предоставить выделенный запятыми список типов исключений; синтаксис функции имитирует его как список аргументов для прототипа функции. Например, следующий прототип указывает на функцию, которая может генерировать как исключение `* char`, так и исключение `double`:

```
double multi_err(double z) throw(char *, double);
```

Как видно из листинга 14.10, такая же информация, отображающаяся в прототипе, должна выводиться и в определении функции. Использование пустых скобок в спецификации исключений указывает на то, что функция не генерирует исключения:

```
double simple(double z) throw();
// не генерирует исключение
```

Как указывалось ранее, в листинге 14.10 все операторы цикла `while` размещены внутри блока `try`. Здесь была также добавлена вторая функция генерирования исключений, `gmean()`. Эта функция возвращает среднее геометрическое двух величин, которое определяется как корень квадратный их произведения. Эта функция не определяется для отрицательных аргументов, что создает условия для генерирования исключений.

## Листинг 14.10 Программа error4.cpp.

```

//error4.cpp
#include <iostream>
using namespace std;
#include <cmath> // или math.h, пользователи unix могут нуждаться во флаге -lm
double hmean(double a, double b) throw(char *);
double gmean(double a, double b) throw(char *);

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

 cout << "Enter two numbers: ";
 try { // начало блока try
 while (cin >> x >> y)
 {
 z = hmean(x,y);
 cout << "Harmonic mean of " << x << " and " << y << " is " << z << "\n";
 cout << "Geometric mean of " << x << " and " << y
 << " is " << gmean(x,y) << "\n";
 cout << "Enter next set of numbers <q to quit>: ";
 }
 } // конец блока try
 catch (char * s) // начало блока catch
 {
 cout << s << "\n";
 cout << "Sorry, you don't get to play any more. ";
 } // конец блока catch
 cout << "Bye!\n";
 return 0;
}

double hmean(double a, double b) throw(char *)
{
 if (a == -b)
 throw "bad hmean() arguments: a = -b not allowed";
 return 2.0 * a * b / (a + b);
}

double gmean(double a, double b) throw(char *)
{
 if (a < 0 || b < 0)
 throw "bad gmean() arguments: negative values not allowed";
 return sqrt(a * b);
}

```

Подобно функции **hmean()**, функция **gmean()** генерирует исключение строкового типа, поэтому один и тот же блок **catch** перехватывает исключения для одной из этих двух функций.

Вот результат выполнения программы, которое было прервано из-за неправильного ввода данных для функции **hmean()**:

```

Enter two numbers: 1_100
Harmonic mean of 1 and 100 is 1.9802
Geometric mean of 1 and 100 is 10
Enter next set of numbers <q to quit>: 10_-10
bad hmean() arguments: a = -b not allowed
Sorry, you don't get to play any more. Bye!

```

Поскольку обработчик исключений находится вне цикла, некорректный ввод прерывает выполнение цикла. После того как программа заканчивает выполнение

кода обработчика, она переходит к следующей строке в программе, которая выводит на экран слово "Bye!".

Для сравнения приводим пример, когда из-за некорректного ввода прерывается выполнение функции **gmean()**:

```

Enter two numbers: 1_100
Harmonic mean of 1 and 100 is 1.9802
Geometric mean of 1 and 100 is 10
Enter next set of numbers <q to quit>: 3_-15
Harmonic mean of 3 and -15 is 7.5
bad gmean() arguments: negative values not
allowed
Sorry, you don't get to play any more. Bye!

```

**Многочисленные блоки try**

При использовании блоков **try** у вас есть много вариантов выбора. Например, можно было бы обработать два

функциональных вызова в индивидуальном порядке, разместив каждый из них в отдельном блоке `try`. Это позволит программировать различные ответы на два возможных исключения, как показано в следующем примере кода:

```

while (cin >> x >> y)
{
 try { //блок try #1
 z = hmean(x,y);
 } //конец блока try #1
 catch (char * s) //начало блока catch #1
 {
 cout << s << "\n";
 cout << "Enter a new pair of numbers: ";
 continue;
 } //конец блока catch #1
 cout << "Harmonic mean of " << x
 << " and " << y
 << " is " << z << "\n";
 try { //блок try #2
 z = gmean(x,y);
 } //конец блока try #2
 catch (char * s) //начало блока catch #2
 {
 cout << s << "\n";
 cout << "Data entry terminated!\n";
 break;
 } //конец блока catch #2
 cout << "Enter next set of
 numbers <q to quit>: ";
}

```

Другой возможный вариант — использовать вложенные блоки `try`, как показано в следующем примере:

```

try { //внешний блок try
 while (cin >> x >> y)
 {
 try { //внутренний блок try
 z = hmean(x,y);
 } //конец внутреннего блока try
 catch (char * s) //внутренний блок catch
 {
 cout << s << "\n";
 cout << "Enter a new pair of numbers: ";
 continue;
 } //конец внутреннего блока catch
 cout << "Harmonic mean of " << x
 << " and " << y
 << " is " << z << "\n";
 cout << "Geometric mean of " << x
 << " and " << y
 << " is " << gmean(x,y) << "\n";
 cout << "Enter next set of numbers
 <q to quit>: ";
 } //конец внешнего блока try
 catch (char * s) //внешний блок catch
 {
 cout << s << "\n";
 cout << "Sorry, you don't get to
 play any more. ";
 } //конец внешнего блока catch
}

```

В этом примере исключение, генерируемое функцией `hmean()`, перехватывается обработчиком внутренних исключений, что способствует продолжению выполнения цикла. Однако исключение, генерируемое функцией `gmean()`, перехватывается обработчиком внешних исключений, вследствие чего выполнение цикла завершается.

### Разворачивание стека

Предположим, блок `try` не выполняет непосредственно вызов функции, генерирующей исключение, но вызывает функцию, которая генерирует исключение. Выполнение программы передается от функции, в которой генерируется исключение, функции, содержащей блок `try` и обработчики исключений. Чтобы достичь этого, необходимо выполнить процесс, который называется *разворачиванием стека* и который мы рассмотрим сейчас.

Во-первых, давайте рассмотрим, как C++ обычно обрабатывает вызовы и возвраты функции. Обычно C++ обрабатывает вызов функции, размещая информацию в стеке (см. главу 8). В частности, программа размещает адрес инструкции по вызову функции (*адрес возврата*) в стеке. При завершении вызываемой функции программа использует указанный адрес для обозначения того места, с которого продолжается выполнение программы. В процессе вызова функции также размещаются любые ее аргументы в стеке, где им придается статус переменных с автоматическим выделением памяти. Если вызываемая функция создает любые новые переменные с автоматическим выделением памяти, они, в свою очередь, также добавляются в стек. Если вызываемая функция вызывает другую функцию, то результат выполнения добавляется в стек и т.д. Когда функция завершает свое выполнение, выполнение программы передается адресу, сохранившемуся после вызова функции, и верхняя часть стека освобождается. Таким образом, функция обычно возвращается к той функции, которая ее вызвала, и т.д., причем каждая из них при завершении высвобождает переменные с автоматическим выделением памяти. Если переменные с автоматическим выделением памяти являются объектом класса, то вызывается деструктор класса (если таковой имеется).

Теперь предположим, что выполнение функции прерывается не с помощью обратного вызова, а с помощью вызова исключения. Программа снова освобождает память стека. Но вместо того, чтобы остановиться на первом адресе возврата в стеке, программа продолжает освобождать стек, пока не достигнет адреса возврата, постоянно находящегося в блоке `try` (рис. 14.3). Затем управление передается обработчикам исключений в конце блока первому оператору, следующему за вызовом функции. Этот процесс называется *разворачиванием стека*.

ка. Одной из важных особенностей механизма генерации исключений является то, что с возвратом функций вызываются деструкторы классов для любых автоматических объектов класса в стеке. Тем не менее, при возврате функции всего лишь обрабатываются объекты, размещаемые в стеке этой функцией, в то время как оператор генерирования исключений обрабатывает объекты, размещаемые в стеке при выполнении цепи вызовов функции между блоком `try` и генерацией исключений. Без свойства разворачивания стека при генерации исключений останутся невызванными деструкторы для автоматических объектов классов, размещенных в стеке при реализации промежуточных вызовов функции.

В листинге 14.11 приводится пример разворачивания стека. В нем функция `main()` вызывает функцию `details()`, а функция `details()` вызывает функцию `hmean()`. Когда функция `hmean()` генерирует исключение, управление передается функции `main()`, где оно перехватывается. При осуществлении этого процесса освобождаются переменные с автоматическим выделением памяти, представляющие аргументы функциям `hmean()` и `details()`.

#### Листинг 14.11 Программа errors.cpp.

```
//error5.cpp
#include <iostream>
using namespace std;
double hmean(double a, double b) throw(char *);
void details(double a, double b) throw(char *);
int main()
{
 double x, y;
 cout << "Enter two numbers: ";
 try {
 while (cin >> x >> y)
 details(x,y);
 }
 catch (char * s)
 {
 cout << s << "\n";
 cout << "Sorry, you can't play anymore. ";
 }
 cout << "Bye!\n";
 return 0;
}

void details(double a, double b) throw(char *)
{
 cout << "Harmonic mean of " << a
 << " and " << b
 << " is " << hmean(a,b) << "\n";
 cout << "Enter next set of numbers
 << q to quit>: ";
}

double hmean(double a, double b) throw(char *)
{
 if (a == -b)
 throw "bad hmean() arguments:
 -a = -b not allowed";
 return 2.0 * a * b / (a + b);
}
```

Это образец выполнения программы; обратите внимание на то, что генерирование исключений передается непосредственно функции `main()`, что не позволяет функции `details()` показывать текст, который в другом случае отображался бы функцией `hmean()`.

```
Enter two numbers: 3 15
Harmonic mean of 3 and 15 is 5
Enter next set of numbers <q to quit>: 20 -20
bad hmean() arguments: a = -b not allowed
Sorry, you don't get to play any more. Bye!
```

#### Дополнительные опции

Чтобы перехватить любой вид исключения, можно установить обработчик исключений. Кроме того, если блок `try` вложен, можно сделать так, что его обработчики исключений передадут управление обработчикам исключений для внешнего блока `try`. Чтобы перехватить любое исключение, используйте многоточие для указания типа исключений:

```
catch (...) { // операторы }
```

Чтобы передать управление внешнему блоку `try`, используйте `throw` без указания последующего исключения:

```
catch (char * s)
{
 cout << "Exception caught in inner loop.\n";
 throw; //отсылка к включающему блоку try
}
```

Этот образец программного кода выводит на печать сообщение, затем передает управление содержащему блоку `try`, где программа снова будет искать обработчик исключений, соответствующий первоначально генерируемому исключению.

Обратите внимание на то, что существует не единственный способ включения одного блока `try` в другой.

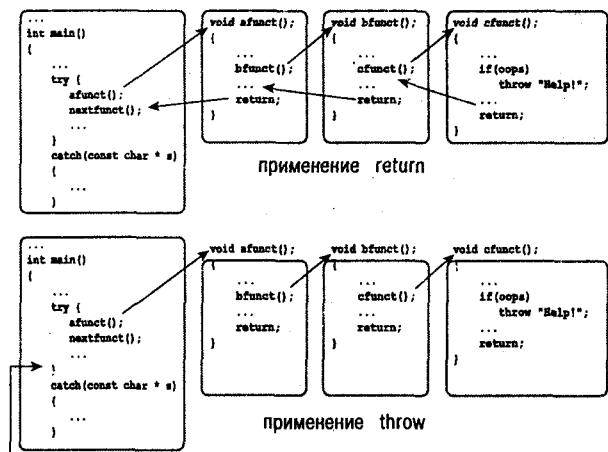


РИСУНОК 14.3 Операторы `throw` и `return`.

Один из способов — вложить один блок в другой, о чём говорилось выше. Другой способ состоит в том, что один блок `try` содержит вызов функции, которая активизирует функцию, содержащую блок `try`. В первом случае вышеупомянутый образец кода передаст управление внешнему блоку `try`. Во втором случае образец кода вызовет разворачивание стека для поиска следующего блока `try`.

## Исключения и классы

Исключения служат не только для того, чтобы обрабатывать условия ошибок в обычных функциях. Они также могут быть частью проекта класса. Например, конструктор мог бы генерировать исключение в случае, если не удастся осуществить вызов оператора `new`. Или же перегруженный оператор `[]` для класса массива может генерировать исключение, если индекс находится за пределами диапазона. Иногда интересными бывают случаи, когда исключение может привносить информацию, например, о величине неверного индекса. Говорят, в такой ситуации можно было бы использовать тип исключения `int`, но более рациональным будет генерирование исключения, являющегося объектом. Тип объекта поможет установить источник исключения. В примере с функциями `hmean()` и `gmean()`, приводившемся ранее, появлялась проблема, суть которой в том, что обе функции генерируют один и тот же тип исключения (`char *`). Это привело к дополнительным сложностям при установке блоков `catch`, создающих различие между этими двумя функциями. Используя объекты, можно создать другой тип объектов для каждого исключения, которое вы бы хотели перехватить. К тому же объект сам по себе может содержать полезную информацию.



### СОВЕТ

Если функция генерирует исключение, определите класс исключения, используемого как тип генерируемого исключения.

Действительно, обычной практикой использования исключений является вызов объектов как исключений и их перехват с помощью ссылки:

```
class problem {...} ;
...
void super()
{
 ...
 if (oh_no)
 {
 problem oops(); //конструирование объекта
 throw oops; //генерировать ошибки
 ...
 }
 ...
}
```

```
try {
 super();
}
catch(problem & p)
{
 ...
}
```

Кстати заметим, что, хотя механизм вызова/перехвата во многом похож на передачу аргумента функции, все же между этими двумя процессами есть некоторые отличия. Например, компилятор всегда при генерации исключения создает резервную копию, поэтому в приведенном выше примере кода `p` больше относится к копии `oops`, чем к самому `oops`. Это неплохо, поскольку, после того как завершается выполнение функции `super()`, `oops` уже не существует. Часто случается так, что проще совместить конструирование и вызов:

```
throw oops(); //конструирование и вызов объекта
```

Если исключения относятся к процессам класса, зачастую целесообразнее определять тип исключения как вложенный класс. Это не только позволяет определять класс, генерирующий исключения, но и помогает предотвратить конфликт названий. Предположим, у вас есть класс, называемый `ArrayDbE`, в котором публично объявлен другой класс, названный `BadIndex`. Если оператор `[]` находит неподходящую величину индекса, он может генерировать исключение типа `BadIndex`. В таком случае обработчик исключений для этого типа исключений будет выглядеть так:

```
catch (const ArrayDbE::BadIndex &) {...}
```

Классификатор `ArrayDbE` определяет `BadIndex` как объявленный в классе `ArrayDbE`. Он также говорит читателю о том, что этот обработчик исключений предназначается для исключений, генерируемых объектами `ArrayDbE`. Название `BadIndex` дает читателю неплохое представление о происхождении исключений. Звучит довольно привлекательно, поэтому давайте подробно изучим этот вопрос. В частности, давайте добавим исключения в класс `ArrayDbE`, впервые появившийся в главе 13.

К заголовочному файлу добавьте класс исключений `BadIndex`, т.е. класс, определяющий объекты, которые должны генерироваться как исключения. Он будет применяться для хранения некорректных величин индексов и будет содержать величину некорректного индекса. Обратите внимание на то, что объявление вложенного класса — это всего лишь описание класса, оно не создаёт объекты. Методы класса создадут объекты этого класса, если они генерируют исключения типа `BadIndex`. Обратите также внимание на то, что вложенный класс общедоступен. Это позволяет блокам `catch` иметь доступ

к данному типу класса. Листинг 14.12 демонстрирует новый заголовочный файл. В остальном определение такое же (кроме имени класса), как и определение класса `ArrayDbE`, описанное в главе 13, за исключением того, что оно уточняет прототипы методов для указания тех типов исключений, которые они могут генерировать. Другими словами,

```
virtual double & operator[](int i);
```

заменяется на

```
virtual double & operator[](int i)
throw(BadIndex &);
```

и т.д. (Как и в случае с обычными аргументами функции, при генерировании исключений лучше передать ссылки вместо объектов).

Теперь нужно реализовать методы класса. Это те же методы, которые были описаны в главе 12, с добавлением нескольких генераций исключений. Поскольку перегруженные операторы [] генерируют исключения вместо вызова функции `exit()`, программе далее нет необходимости включать файл `cstdlib`. В листинге 14.13 показан наглядный пример.

Обратите внимание, что в роли исключений теперь используются не строки, а объекты. Заметьте еще, что в этом случае для создания и инициализации объектов исключений используется конструктор класса исключений:

```
if (i < 0 || i >= size)
throw BadIndex(i); //создание,
//инициализация объекта BadIndex
```

Что можно сказать относительно перехвата этого типа исключений? Исключения — это объекты, а не строки символов, поэтому блок `catch` должен отражать этот факт. Кроме того, поскольку исключение является вложенным типом, в код нужно включить оператор определения диапазона доступа:

```
try {
...
}
catch(ArrayDbE::BadIndex &) {
...
}
```

В листинге 14.14 приведена небольшая программа, демонстрирующая этот процесс.

Обратите внимание на то, что второй цикл `for` умышленно выходит за рамки массива, вызывая исключение.

Результаты выполнения программы из листинга 14.14:

```
Enter free-throw percentages for your 5 top
players as a decimal fraction:
Player 1: % = 0.923
Player 2: % = 0.858
Player 3: % = 0.821
Player 4: % = 0.744
Player 5: % = 0.697
```

```
Recapitulating, here are the percentages:
Player #1: 92.3%
Player #2: 85.8%
Player #3: 82.1%
Player #4: 74.4%
Player #5: 69.7%
ArrayDbE exception: 5 is a bad index value
Bye!
```

Поскольку цикл находится внутри блока `try`, при генерировании исключения цикл заканчивается в тот момент, когда управление передается второму блоку `catch`, следующему за блоком `try`.

Кстати, запомните, что переменные, определенные в блоке, включая блок `try`, характерны только для этого блока. Например, переменная `player` становится неопределенной в случае, если управление программой передается за пределы блока `try` в листинге 14.14.

## Исключения и наследование

Наследование взаимодействует с исключениями несколькими способами. Во-первых, если класс включает открыто вложенные классы исключений, производный класс их унаследует. Во-вторых, можно вывести новые классы исключений из уже существующих. В следующем примере будут рассмотрены обе эти возможности.

Прежде всего выведите класс `LimitArE` из класса `ArrayDbE`. Класс `LimitArE` допускает индексирование массива с начала с помощью значений, отличных от нуля. Это можно выполнить, сохранив величину, представляющую начальный индекс с дальнейшим переопределением функций. Внутреннее индексирование массива начнется с нуля. Но, скажем, если вы определите 1900 в качестве начального индекса, метод `operator[]()` транслирует внешний индекс 1908 во внутренний индекс 1908–1900 или 8. В листинге 14.15 приведены подробности этого процесса.

Исключение `BadIndex`, объявленное в классе `ArrayDbE`, содержало некорректную величину индекса. При варьировании пределов индексов было бы неплохо, если бы исключение также сохраняло допустимый диапазон индексов. Этого можно достичь, выведя новый класс исключений из `BadIndex`:

```
class SonOfBad : public ArrayDbE::BadIndex
{ public:
 int l_lim; // нижний индексный предел
 int u_lim; // верхний индексный предел
 SonOfBad(int i, int l, int u) :
 BadIndex(i), l_lim(l), u_lim(u) {} };
```

Можно вложить объявление `SonOfBad` в объявление `LimitArE`. В листинге 14.15 демонстрируется результат выполнения этой операции.

В этом проекте происходит передача проверки индекса от перегруженных методов [] к методам `ok()`, которые вызываются перегруженными методами [].

**Листинг 14.12 Класс arraydbe.h.**

```

// arraydbe.h - определение класса массива с исключениями
#ifndef _ARRAYDBE_H_
#define _ARRAYDBE_H_
#include <iostream>
using namespace std;

class ArrayDbE
{
private:
 unsigned int size; // количество элементов массива
protected:
 double * arr; // адрес первого элемента
public:
 class BadIndex // класс исключений для индексации проблем
 {
public:
 int badindex; // значение проблемного индекса
 BadIndex(int i) : badindex(i) { }
 };
 ArrayDbE(); // конструктор, заданный по умолчанию
 // создание массива ArrayDbE из n элементов, присвоение каждому элементу значения val
 ArrayDbE(unsigned int n, double val = 0.0);
 // создание массива ArrayDbE из n элементов, инициализация массивом pn
 ArrayDbE(const double * pn, unsigned int n);

 // копирование конструктора
 ArrayDbE(const ArrayDbE & a);
 virtual ~ArrayDbE(); // деструктор
 unsigned int ArSize() const; // возврат размера массива
 double Average() const; // возврат среднего значения массива

 // перегруженные операторы
 // индексирование массива, допуск присваивания
 virtual double & operator[](int i) throw(BadIndex &);

 // индексирование массива (не =)
 virtual const double & operator[](int i) const throw(BadIndex &);
 ArrayDbE & operator=(const ArrayDbE & a);
 friend ostream & operator<<(ostream & os, const ArrayDbE & a);
};

#endif

```

**Листинг 14.13 Программа arraydbe.cpp.**

```

// arraydbe.cpp - методы класса ArrayDbE
#include <iostream>
using namespace std;
#include "arraydbe.h"

// конструктор, заданный по умолчанию - отсутствуют аргументы
ArrayDbE::ArrayDbE()
{
 arr = NULL;
 size = 0;
}

// создается массив из n элементов, каждому элементу присваивается значение val
ArrayDbE::ArrayDbE(unsigned int n, double val)
{
 arr = new double[n];
 size = n;
 for (int i = 0; i < size; i++)
 arr[i] = val;
}

```

```

// инициализация объекта ArrayDbE массивом, который не является классом
ArrayDbE::ArrayDbE(const double *pn, unsigned int n)
{
 arr = new double[n];
 size = n;
 for (int i = 0; i < size; i++)
 arr[i] = pn[i];
}

// инициализация объекта ArrayDbE другим объектом ArrayDbE
ArrayDbE::ArrayDbE(const ArrayDbE & a)
{
 size = a.size;
 arr = new double[size];
 for (int i = 0; i < size; i++)
 arr[i] = a.arr[i];
}

ArrayDbE::~ArrayDbE()
{
 delete [] arr;
}

double ArrayDbE::Average() const
{
 double sum = 0;
 int i;
 int lim = ArSize();
 for (i = 0; i < lim; i++)
 sum += arr[i];
 if (i > 0)
 return sum / i;
 else
 {
 cerr << "No entries in score array\n";
 return 0;
 }
}

// возврат размера массива
unsigned int ArrayDbE::ArSize() const
{
 return size;
}

// разрешение доступа пользователя к элементам по индексу (присваивание допускается)
double & ArrayDbE::operator[](int i) throw(BadIndex &)
{
 // проверка индекса перед продолжением
 if (i < 0 || i >= size)
 throw BadIndex(i);
 return arr[i];
}

// разрешение доступа пользователя к элементам по индексу (присваивание не допускается)
const double & ArrayDbE::operator[](int i) const throw(BadIndex &)
{
 // проверка индекса перед продолжением
 if (i < 0 || i >= size)
 throw BadIndex(i);
 return arr[i];
}

// определение назначения класса
ArrayDbE & ArrayDbE::operator=(const ArrayDbE & a)
{
 if (this == &a) // если объект присваивается сам,
 return *this; // ничего не изменять
}

```

```

delete arr;
size = a.size;
arr = new double[size];
for (int i = 0; i < size; i++)
 arr[i] = a.arr[i];
return *this;
}

// быстрый вывод, пять значений в строке
ostream & operator<<(ostream & os, const ArrayDbE & a)
{
 int i;
 for (i = 0; i < a.size; i++)
 {
 os << a.arr[i] << " ";
 if (i % 5 == 4)
 os << "\n";
 }
 if (i % 5 != 0)
 os << "\n";
 return os;
}

```

#### Листинг 14.14 Программа exceptar.cpp.

```

//exceptar.cpp - использование класса ArrayDbE
//Компиляция совместно с программой arraydbe.cpp

#include <iostream>
using namespace std;
#include "arraydbe.h"

const int Players = 5;
int main()
{
 try {
 ArrayDbE Team(Players);
 cout << "Enter free-throw percentages for your 5 "
 "top players as a decimal fraction:\n";
 int player;
 for (player = 0; player < Players; player++)
 {
 cout << "Player " << (player + 1) << ": % = ";
 cin >> Team[player];
 }

 cout.precision(1);
 cout.setf(ios_base::showpoint);
 cout.setf(ios_base::fixed,ios_base::floatfield);
 cout << "Recapitulating, here are the percentages:\n";

 for (player = 0; player <= Players; player++)
 cout << "Player #" << (player + 1) << ": "
 << 100.0 * Team[player] << "%\n";
 } // конец блока try

 catch (ArrayDbE::BadIndex & bi) // начало обработчика исключений
 {
 cout << "ArrayDbE exception: "
 << bi.badindex << " is a bad index value\n";
 } // конец обработчика исключений

 cout << "Bye!\n";
 return 0;
}

```

**Листинг 14.15 Класс limarre.h.**

```

// limarre.h - LimitArE класс с исключениями
#ifndef _LIMARRE_H_
#define _LIMARRE_H_
#include "arraydbe.h"
class LimitArE : public ArrayDbE
{
public:
 class SonOfBad : public ArrayDbE::BadIndex
 {
public:
 int l_lim;
 int u_lim;
 SonOfBad(int i, int l, int u) : BadIndex(i), l_lim(l), u_lim(u) { }
 };
private:
 unsigned int low_bnd; // новый элемент данных
protected:
 // проверка дескрипторов границ
 virtual void ok(int i) const throw(SonOfBad &);
public:
// конструкторы
 LimitArE() : ArrayDbE(), low_bnd(0) { }
 LimitArE(unsigned int n, double val = 0.0) : ArrayDbE(n, val), low_bnd(0) { }
 LimitArE(unsigned int n, int lb, double val = 0.0) : ArrayDbE(n, val), low_bnd(lb) { }
 LimitArE(const double * pn, unsigned int n) : ArrayDbE(pn, n), low_bnd(0) { }
 LimitArE(const ArrayDbE & a) : ArrayDbE(a), low_bnd(0) { }
// новые методы
 void new_lb(int lb) { low_bnd = lb; } // переустановка нижней границы
 int lbound() { return low_bnd; } // возвращение нижней границы
 int ubound() { return ArSize() + low_bnd - 1; } // верхняя граница
// переопределенные операторы
 double & operator[](int i);
 const double & operator[](int i) const;
};
#endif

```

**Листинг 14.16 Программа limarre.cpp.**

```

// limarre.cpp
#include "limarre.h"
#include <iostream>
using namespace std;

// приватный метод
// нижняя граница для массива индекса теперь будет равна low_bnd,
// а верхняя граница - low_bnd + size - 1
void LimitArE::ok(int i) const throw(SonOfBad &)
{
 unsigned long size = ArSize();
 if (i < low_bnd || i >= size + low_bnd)
 throw SonOfBad(i, low_bnd, low_bnd + size - 1);
}

// переопределенные операторы
double & LimitArE::operator[](int i)
{
 ok(i);
 return arr[i - low_bnd];
}

const double & LimitArE::operator[](int i) const
{
 ok(i);
 return arr[i - low_bnd];
}

```

Следовательно, именно метод `LimitArE::ok()` генерирует исключение `SonOfBad`. В листинге 14.16 демонстрируются методы класса.

Предположим, что у вас есть программа, включающая как объекты `ArrayDbE`, так и объекты `LimitArE`. Затем может понадобиться блок `try`, перехватывающий два возможные исключения, `BadIndex` и `SonOfBad`. Это можно выполнить, разместив вслед за блоком `try` два последовательных блока `catch`:

```

try {
 LimitArE income(Years, FirstYear);
 ArrayDbE busywork(Years);
 ...
} // конец блока try
// первый обработчик исключений
catch (LimitArE::SonOfBad & bi)
{
 ...
}
// второй обработчик исключений
catch (LimitArE::BadIndex & bi)
{
 ...
}

```

Если имеется последовательность блоков `catch`, программа пытается установить соответствие генерируемого исключения с первым блоком `catch`, затем со вторым и т.д. Как только соответствие установлено, программа выполняет этот блок. Включение кода в блок `catch` не завершает программу и не генерирует другой вызов исключения; программа переходит к оператору, следующему за последним блоком `catch`, после завершения любого блока `catch` в этой последовательности. Именно эта последовательность блоков `catch` обладает интересным свойством — блок `catch` со ссылкой на `BadIndex` может перехватить либо исключение `BadIndex`, либо исключение `SonOfBad`. Это происходит из-за того, что ссылка на основной класс может относиться к производному объекту. Однако блок `catch` со ссылкой `SonOfBad` не может перехватить объект `BadIndex`, поскольку ссылка на производный объект не может относиться к объекту базового класса без явного приведения типа. Такое положение вещей предполагает размещение блока перехвата `SonOfBad` до блока перехвата `BadIndex`. Таким образом, блок `catch SonOfBad` перехватит исключение `SonOfBad` при передаче исключения `BadIndex` следующему блоку `catch`. Программа из листинга 14.17 иллюстрирует данный подход.

Результаты выполнения программы из листинга 14.17:

```

Enter your income for the last 4 years:
Year 1998: $35000
Year 1999: $34000
Year 2000: $33000
Year 2001: $38000
Recapitulating, here are the figures:
1998: $35000.00
1999: $34000.00
2000: $33000.00
2001: $38000.00
LimitArE exception: 2002 is a bad index value.
Index should be in the range 1998 to 2001.
Total income for 4 years is $140000.00.

```

Исключение `SonOfBad` завершило выполнение блока `try` и передало выполнение второму блоку `catch`. Поскольку программа завершила обработку блока `catch`, она перешла к первому утверждению, следующему за блоками `catch`.

Следующий совет можно расценивать как главный урок этого примера.



### COBET

Если у вас имеется иерархия наследований классов исключений, организуйте блоки `catch` в таком порядке, чтобы первым перехватывался последний производный класс исключения, а последним — базовый класс исключения.

## Класс `exception`

Основная цель исключений C++ состоит в обеспечении поддержки на уровне языка для проектируемых отказоустойчивых программ. Другими словами, благодаря исключениям легче включить обработку ошибок в проект программы, чем добавлять некоторые более устойчивые формы обработки ошибок. Переносимость и относительное удобство исключений должно вдохновить программистов на интеграцию обработки ошибок в процесс проектирования программы, если это возможно. Итак, исключения представляют собой такое свойство, которое позволяет, как и в случае с классами, изменять подход к программированию.

Более новые компиляторы C++ делают исключения частью языка. Например, заголовочный файл `exception` (ранее назывался `exception.h` или `except.h`) определяет класс `exception`, используемый C++ в качестве базового для других классов исключений, поддерживаемых языком. Ваш код также может генерировать объект `exception` или использовать класс `exception` в качестве базового. Одна виртуальная функция-элемент именуется `what()`, она возвращает строку, которая по своей природе зависит от реализации. Однако если вы создадите производный класс, то можете выбрать, какую строку следует возвращать.

## Листинг 14.17 Программа excptinh.cpp.

```

// excptinh.cpp - использование классов ArrayDbE и LimitArE
// Компиляция вместе с программами arraydbe.cpp, limarre.h

#include <iostream>
using namespace std;
#include "arraydbe.h"
#include "limarre.h"

const int Years = 4;
const int FirstYear = 1998;
int main()
{
 int year;
 double total = 0;
 try {
 LimitArE income(Years, FirstYear);
 ArrayDbE busywork(Years);
 cout << "Enter your income for the last " << Years
 << " years:\n";
 for (year = FirstYear; year < FirstYear + Years; year++)
 {
 cout << "Year " << year << ": $";
 cin >> income[year];
 busywork[year - FirstYear] = 0.2 * income[year];
 }
 cout.precision(2);
 cout.setf(ios_base::showpoint);
 cout.setf(ios_base::fixed,ios_base::floatfield);
 cout << "Recapitulating, here are the figures:\n";
 for (year = FirstYear; year <= FirstYear + Years; year++)
 {
 cout << year << ": $" << income[year] << "\n";
 total += income[year];
 }
 cout << "busywork values: " << busywork;
 } // конец блока try

 catch (LimitArE::SonOfBad & bi) // первый обработчик исключений
 {
 cout << "LimitArE exception: "
 << bi.badindex << " is a bad index value\n";
 cout << "Index should be in the range " << bi.l_lim
 << " to " << bi.u_lim << ".\n";
 }

 catch (LimitArE::BadIndex & bi) // второй обработчик исключений
 {
 cout << "ArrayDbE exception: "
 << bi.badindex << " is a bad index value.\n";
 }

 cout << "Total income for " << (year - FirstYear)
 << " years is $" << total << ".\n";
 cout << "Bye!\n";

 return 0;
}

```

Например, можно заменить объявление `BadIndex` на какое-нибудь другое, имеющее в качестве основы класс `exception`. Поскольку содержание заголовочного файла `exception` является частью пространства имен `std`, они могут стать доступными при использовании следующей директивы:

```
#include <exception>
using namespace std;
class ArrayDbE
{
private:
 unsigned int size; //количество элементов
 //массива
protected:
 double * arr; //адрес первого элемента
public:
 class OutOfBounds : public exception
 {
 public:
 OutOfBounds() : exception() {};
 const char * what()
 {return "Array limit out of bounds\n";}
 };
 ...
};
```

Затем перегруженные методы `[]` могли бы генерировать исключение `OutOfBoundsException`. Программа, использующая класс, перехватывает этот тип исключения и использует метод `what()` для распознавания типа проблемы:

```
// обработчик исключений
catch (ArrayDbE::OutOfBoundsException & ob)
{
 cout << "ArrayDbE exception: "
 << ob.what() << endl;
}
```

Кстати, вместо того чтобы, используя директиву, сделать доступными все объявления класса исключений, можно использовать оператор определения диапазона доступа:

```
class OutOfBounds : public std::exception
```

## Исключение `bad_alloc` и оператор `new`

Язык C++ обеспечивает реализацию двух методов обработки проблем, возникающих при распределении памяти с помощью оператора `new`. Первый способ, который длительное время был единственным, заключается в том, что оператор `new` возвращает нулевой указатель, если он не может удовлетворить запрос на выделение памяти. Второй метод состоит в том, что `new` генерирует исключение `bad_alloc`. Заголовок `new` (бывший `new.h`) включает в себя объявление класса `bad_alloc`, который открыто произведен из класса `exception`. Реализация предлагает вам только один выбор, хотя, возможно, переключатель компилятора или любого другого метода предоставит другой варианта выбора.

В листинге 14.18 рассматриваются эти два подхода. При перехвате исключения программа демонстрирует зависимое от реализации сообщение, возвращенное производным методом `what()`, и досрочно завершает свое выполнение. В противном случае программа продолжает выполняться до тех пор, пока возвращаемое значение не будет нулевым указателем. (Цель этого действия — продемонстрировать два способа проверки ошибок при распределении памяти, а не утверждение о том, что типичная программа на самом деле должна использовать оба этих метода).

### Листинг 14.18 Программа newexcp.cpp.

---

```
// newexcp.cpp - исключение bad_alloc
#include <iostream>
using namespace std;
#include <new>
#include <cstdlib>

struct Big
{
 double stuff[2000];
};

int main()
{
 Big * pb;
 try
 {
 cout << "Trying to get a big block
 of memory:\n";
 pb = new Big[10000];
 cout << "Got past the new request:\n";
 }
 catch (bad_alloc & ba)
 {
 cout << "Caught the exception!\n";
 cout << ba.what() << endl;
 exit(1);
 }
 if (pb != 0)
 {
 pb[0].stuff[0] = 4;
 cout << pb[0].stuff[0] << endl;
 }
 else
 cout << "pb is null pointer\n";
 delete [] pb;
 return 0;
}
```

---

## Проблемы, связанные с исключениями

После генерирования исключения могут возникнуть две проблемы. Во-первых, если исключение генерируется функцией, имеющей его описание, то оно должно соответствовать одному из типов в описательном списке. Если этого не происходит, то несоответствующее исключение маркируется как *непредвиденное исключение* и по умолчанию приводит к прерыванию программы. Во-вторых, если исключение преодолевает это первое препят-

ствие (или же избегает его, если функция располагает недостаточным описанием исключения), то его в таком случае нужно перехватить. Если этого не произойдет (что может случиться из-за отсутствия включающего блока `try` или подходящего блока `catch`), исключение маркируется как *неперехваченное* исключение и по умолчанию приводит к прерыванию программы. Тем не менее, вы можете изменить ответ программы на непредвиденные и неперехваченные исключения. Посмотрим, как это делается, начиная с непредвиденных исключений.

**Непредвиденное исключение** не приводит к немедленному прерыванию выполнения программы. Вместо этого программа сначала вызывает функцию `terminate()`. По умолчанию эта функция, в свою очередь, вызывает функцию `abort()`. Можно изменить поведение функции `terminate()`, зарегистрировав функцию, которую вызовет функция `terminate()` вместо функции `abort()`. Чтобы добиться этого, воспользуйтесь функцией `set_terminate()`. Обе функции, `set_terminate()` и `terminate()`, объявлены в заголовочном файле `exception`:

```
typedef void (*terminate_handler)();
terminate_handler
 set_terminate(terminate_handler f) throw();
void terminate();
```

Функция `set_terminate()` в качестве аргумента использует имя функции (т.е. ее адрес), у нее отсутствуют аргументы, а возвращаемый результат будет иметь тип `void`. Она возвращает адрес предыдущей зарегистрированной функции. При вызове функции `set_terminate()` более одного раза функция `terminate()` вызывает функцию, установленную в результате самого последнего вызова функции `set_terminate()`.

Давайте рассмотрим пример. Предположим, вы бы хотели, чтобы с помощью неперехваченного исключения программа вывела на печать сообщение и затем вызвала функцию `exit()`, обеспечив значение состояния выхода 5. Прежде всего введите заголовочный файл `exception` и сделайте доступными его объявления, используя следующую директиву:

```
#include <exception>
using namespace std;
```

Далее создайте функцию, выполняющую требуемые действия и имеющую соответствующий прототип:

```
void myQuit()
{
 cout << "Terminating due to uncaught
 exception\n";
 exit(5);
}
```

И наконец, в начале программы обозначьте эту функцию как выбранное вами заключительное действие:

```
set_terminate(myQuit);
```

Теперь давайте рассмотрим непредвиденные исключения. Используя в функции описания исключений, вы даете возможность пользователям функций узнать, какие исключения необходимо перехватывать. Предположим, у вас есть следующий прототип:

```
double Argh(double, double) throw(exception &);
```

В таком случае эту функцию можно было бы использовать следующим образом:

```
try {
 x = Argh(a, b);
}
catch(exception & ex)
{
 ...
}
```

Неплохо было бы знать, какие именно исключения следует перехватывать; вспомните, что неперехваченное исключение по умолчанию прерывает выполнение программы.

Тем не менее, можно еще кое-что добавить к выше-сказанному. В принципе, описание исключения должно охватывать исключения, которые генерируются функциями, вызываемыми рассматриваемой функцией. Например, если функция `Argh()` вызывает функцию `Duh()`, которая может генерировать исключение объекта `retort`, то этот объект должен появиться в описании исключения как функции `Argh()`, так и функции `Duh()`. Нельзя гарантировать, что все будет сделано правильно, если только вы не создадите все функции самостоятельно и не будете предельно внимательны. Можно, например, воспользоваться устаревшей коммерческой библиотекой, функции которой не содержат описания исключений. Этот факт предполагает более детальное рассмотрение случаев, когда функция генерирует исключение, не характерное для его описания.

Это во многом напоминает неперехваченные исключения. Если появляется непредвиденное исключение, программа вызывает функцию `unexpected()`. (Вы не ожидали появления функции `unexpected()`? Никто не ожидает появления функции `unexpected()!`) Эта функция, в свою очередь, вызывает функцию `terminate()`, которая по умолчанию вызывает функцию `abort()`. Подобно тому как функция `set_terminate()` изменяет поведение функции `terminate()`, функция `set_unexpected()` модифицирует поведение функции `unexpected()`; эти новые функции также объявлены в заголовочном файле исключений:

```
typedef void (*unexpected_handler)();
unexpected_handler
 set_unexpected(unexpected_handler f) throw();
void unexpected();
```

Тем не менее, поведение функции `set_unexpected()` можно предсказать с большей точностью, чем поведение

функции `set_terminate()`. В частности, функция `unexpected_handler` имеет следующие особенности:

- Может завершить выполнение программы, вызвав функции `terminate()` (по умолчанию), `abort()` или `exit()`.
  - Может генерировать исключение.
- Результат генерирования исключений (вторая особенность, описанная выше) зависит от исключения, сгенерированного в результате замены функции `unexpected_handler` и описания исключения:
- Если вновь сгенерированное исключение соответствует описанию исключения, выполнение программы начинается именно с этого места; т.е. она будет пытаться найти блок `catch`, соответствующий вновь генерируемому исключению. По существу, при таком подходе исключение непредвиденного типа превращается в исключение ожидаемого типа.
  - Если вновь сгенерированное исключение не соответствует описанию исключения и если описание исключения не охватывает тип `std::bad_exception`, программа вызывает функцию `terminate()`. Тип `bad_exception` является производным от типа `exception` и объявляется в заголовочном файле `exception`.
  - Если вновь сгенерированное исключение не соответствует описанию исключения и если описание исключения не охватывает тип `std::bad_exception`, несответствующее исключение заменяется исключением типа `std::bad_exception`.

Говоря кратко, если вы хотите перехватить все исключения — ожидаемые и другого типа, то можете осуществить выбор среди следующих строк:

Прежде всего проверьте, доступны ли объявления из заголовочного файла исключений:

```
#include <exception>
using namespace std;
```

Теперь создадим функцию замены, превращающую непредвиденные исключения в тип `bad_exception` и имеющую соответствующий прототип:

```
void myUnexpected()
{
 throw std::bad_exception(); // или просто
 // throw;
}
```

При использовании `throw` без исключения генерируется заново первоначальное исключение. Тем не менее, если описание исключения охватывает этот тип, оно будет заменено объектом `bad_exception`.

Далее в начале программы определите эту функцию как выбранное вами действие непредвиденного исключения:

```
set_unexpected(myUnexpected);
```

И наконец, включите тип `bad_exception` в описания исключений и последовательности блоков `catch`:

```
double Argh(double, double)
 throw(exception &, bad_exception &);

try {
 x = Argh(a, b);
}
catch(exception & ex)
{
 ...
}
catch(bad_exception & ex)
{
 ...
}
```

## Замечание об исключениях

Из предшествующей дискуссии об использовании исключений вы могли бы понять, что обработка исключений должна быть описана в программе раньше, чем фактическое выполнение этой операции. При этом проявляются некоторые недостатки. Например, при использовании исключений увеличивается размер программы и уменьшается скорость ее выполнения. Спецификации исключений не слишком хорошо работают с шаблонами, потому что функции шаблона могут вызывать различные виды исключений в зависимости от вида используемой специализации. Исключения и динамическое распределение памяти не всегда совместимы.

Давайте немного подробнее рассмотрим динамическое распределение памяти и исключения. Сначала обратим внимание на следующую функцию:

```
void test1(int n)
{
 String mesg("I'm trapped in an endless loop");
 ...
 if (oh_no)
 throw exception();
 ...
 return;
}
```

Как вы помните, класс `String` выполняет динамическое распределение памяти. Обычно деструктор `String` для `mesg` вызывается, когда завершается выполнение функции. Благодаря операции по разворачиванию стека, оператор генерирования исключений, несмотря на то, что функция завершается преждевременно, все еще обеспечивает вызов деструктора. Поэтому здесь управление распределением памяти осуществляется должным образом.

Теперь рассмотрим следующую функцию:

```
void test2(int n)
{
```

```

double * ar = new double[n];
...
if (oh_no)
 throw exception();
...
delete [] ar;
return;
}

```

Вот здесь и проявляется проблема. При разворачивании стека переменная `ar` удаляется из стека.

Но преждевременное завершение функции означает, что оператор `delete []`, находящийся в конце функции, пропускается. Указатель перемещается, но блок памяти, помеченный этим указателем, остается неповрежденным и недоступным. Другими словами, происходит "утечка" памяти.

Этой ситуации можно избежать. Например, можно перехватывать исключение в той же самой функции, которая это исключение вызывает, вставив для этого некоторый код выполнения очистки в блок возврата, а затем повторно вызвав исключение:

```

void test3(int n)
{
 double * ar = new double[n];
 ...
 try {
 if (oh_no)
 throw exception();
 }
 catch(exception & ex)
 {
 delete [] ar;
 throw;
 }
 ...
 delete [] ar;
 return;
}

```

Однако, это повышает вероятность появления ошибок. Другое решение заключается в том, чтобы использовать шаблон `auto_ptr`, рассмотренный в главе 15.

Итак, в то время как обработка особых ситуаций чрезвычайно важна для некоторых проектов, это сказывается на сложности программирования, размере программы и скорости ее выполнения. Кроме того, поддержка исключений компилятором и опыт пользователя в настоящее время еще не достигли должного уровня. Так что вы можете использовать эту особенность, несмотря на снижение скорости выполнения программы.

## Библиотека RTTI

Аббревиатура RTTI образована от слов `runtime type information` (тип информации времени выполнения). Подобное название получило одно из самых последних

дополнений к C++, которое не поддерживается многими прежними реализациями языка. Другие реализации могут располагать настройками компилятора для подключения и отключения RTTI. Библиотека RTTI поддерживает стандартные возможности программы по определению типа объекта во время выполнения. Многие библиотеки класса выполняют эти функции для собственных объектов класса. Но если в C++ отсутствует встроенная поддержка и аналогичный механизм создаст кто-либо из поставщиков программного обеспечения, нет уверенности, что этот механизм окажется совместимым с подобными разработками других поставщиков. Создание стандарта языка для RTTI позволит добиться совместимости между библиотеками возможностей.

## Назначение RTTI

Предположим, имеется иерархия классов, которая передается по наследству на общем основании. Для отметки объекта каждого класса этой иерархии можно установить указатель базового класса. Затем вызывается функция, которая после обработки определенной информации выделяет один из этих классов, создает объект этого типа и возвращает его адрес. Этот адрес присваивается указателю базового класса. Таким образом можно определить вид отмечаемых объектов?

Перед тем как дать ответ на этот вопрос, подумаем, зачем нужна информация о типе. Возможно, вам понадобится вызвать корректную версию метода класса. Если это так, вам не обязательно знать тип объекта, поскольку эта функция является виртуальной функцией, которой владеют все элементы иерархии класса. Но если производный объект располагает ненаследуемым методом, применять этот метод могут только определенные объекты. Возможно также, что при решении проблем, возникающих при отладке, приходится отслеживать вид генерируемых объектов. Именно в том случае, если складываются описанные ситуации, удобно обратиться к RTTI.

## Принципы функционирования RTTI

Язык C++ располагает тремя компонентами по поддержке RTTI:

- Оператор `dynamic_cast` генерирует, если это возможно, указатель для производного типа, опираясь на указатель для базового типа. В противном случае оператор возвращает 0 (нулевой указатель).
- Оператор `typeid` возвращает величину, которая идентифицирует точный тип объекта.
- Структура `type_info` содержит информацию об определенном типе.

RTTI можно применять только с иерархией класса, имеющей виртуальные функции. Причина этого явления состоит в том, что только таким иерархиям класса следует присваивать адреса производных объектов в качестве основания для указателей класса.

### ПРЕДОСТЕРЕЖЕНИЕ

Библиотека RTTI функционирует только с классами, которые располагают виртуальными функциями.

Рассмотрим три компонента RTTI.

### Оператор `dynamic_cast`

Оператор `dynamic_cast` является наиболее важным компонентом RTTI. С его помощью нельзя получить ответ на вопрос, какой тип объекта отмечен указателем. Вместо этого можно получить ответ на вопрос, можно ли указателю определенного типа с уверенностью присвоить адрес объекта. Рассмотрим, что это значит, на практике. Предположим, что имеется следующая иерархия:

```
class Grand { //включает виртуальные методы};
class Superb : public Grand { ... };
class Magnificent : public Superb { ... };
```

Предположим также, что имеются следующие указатели:

```
Grand * pg = new Grand;
Grand * ps = new Superb;
Grand * pm = new Magnificent;
```

Наконец, рассмотрим приведения следующего типа:

```
Magnificent * p1 = (Magnificent *) pm; //#1
Magnificent * p2 = (Magnificent *) pg; //#2
Superb * p3 = (Magnificent *) pm; //#3
```

Какие из указанных выше типов приведений являются достаточно надежными? В зависимости от объявлений класса, все они могут быть надежными. Но в то время, как одни из типов гарантируют надежность, другие характеризуются тем, что указатели принадлежат тому же самому типу, что и объект. Также эти типы приведений могут гарантировать наличие прямого или непрямого основного типа для этого объекта. Например, тип приведения #1 является надежным, поскольку он устанавливает тип указателя `Magnificent` для отметки типа объекта `Magnificent`. Тип приведения #2 не является надежным, так как присваивает адрес основного объекта (`Grand`) указателю производного класса (`Magnificent`). Поэтому программа должна ожидать, что основной объект класса имеет свойства производного класса, что, в общем, является ложным утверждением. Объект `Magnificent`, например, может располагать элементами данных, которые будут отсутствовать в объекте `Grand`. Тип приведения #3 является безопасным, поскольку он присваивает адрес производного объекта

указателю основного класса. Иначе говоря, выполнение операции общедоступного произведения способствует тому, что объект `Magnificent` также является объектом `Superb` (прямая база) и объектом `Grand` (непрямая база). Таким образом, вполне безопасно присваивать адрес этих объектов указателям всех трех типов. Виртуальные функции гарантируют, что применение к объекту `Magnificent` указателей любого из этих трех типов приведет к вызову методов `Magnificent`.

Обратите внимание на тот факт, что вопрос о безопасном типе преобразования имеет общий характер и более целесообразен, чем выяснение того, какой тип объекта отмечен. Обычно желание знать тип объясняется тем, что следует уточнить, надежен ли данный тип при вызове метода. При вызове метода нет необходимости в обращении к точно определенному соответствующему типу. Типом может служить и основной тип, для которого определена виртуальная версия метода. Эту точку зрения иллюстрирует пример, приведенный ниже.

Однако сначала рассмотрим синтаксис оператора `dynamic_cast`. Обычно используется оператор следующего формата, где `pg` отмечает объект:

```
Superb pm = dynamic_cast<Superb *>(pg);
```

Возникает вопрос, может ли указатель `pg` являться надежным типом приведения (как описывалось выше) для типа `Superb`? Если это так, оператор возвращает адрес объекта. В противном случае оператор возвращает 0 (нулевой указатель).

### ПОМНИТЕ

В общем случае выражение

```
dynamic_cast<Type *>(pt)
```

преобразует указатель `pt` в указатель типа `Type *`, если отмеченный объект `{*pt}` принадлежит типу `Type` или же является производным прямым или непрямым способом от типа `Type`. В противном случае выражение принимает значение 0 (нулевой указатель).

Листинг 14.19 иллюстрирует этот процесс. Сначала определяются три класса, которые довольно случайно получают наименования `Grand`, `Superb` и `Magnificent`. Класс `Grand` определяет виртуальную функцию `Speak()`, которая переопределяет каждый из других классов. Класс `Superb` определяет виртуальную функцию `Say()`, которую переопределяет `Magnificent` (рис. 14.4). Программа определяет функцию `GetOne()`, которая случайным образом создает и инициализирует объект одного из этих трех типов, затем возвращает адрес в качестве типа указателя `Grand *`. (Функция `GetOne()` при принятии решений моделирует интерактивного пользователя.) При выполнении цикла происходит присваивание этого указателя типу переменной `Grand *` под названием `pg`,

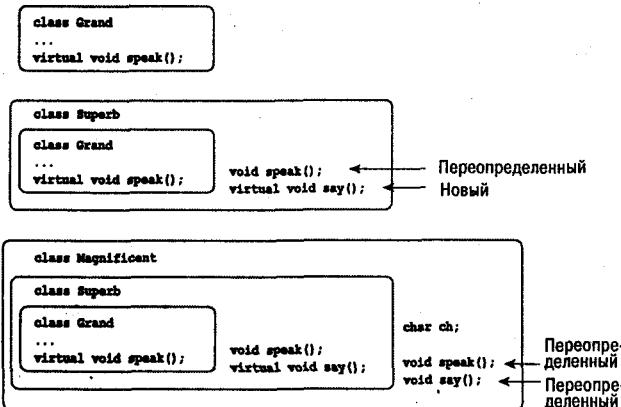


РИСУНОК 14.4 Семейство классов Grand.

затем применяется `pg` для вызова функции `Speak()`. Поскольку эта функция является виртуальной, код корректно активизирует версию `Speak()`, которая соответствует отмеченному объекту:

```

for (int i = 0; i < 5; i++)
{
 pg = GetOne();
 pg->Speak();
 ...
}

```

Правда, в точности этот подход нельзя применять для вызова функции `Say()`; он не определен для класса `Grand`. Однако можно воспользоваться оператором `dynamic_cast` для уточнения, может ли `pg` служить типом приведения для указателя к `Superb`. Ответ будет положительным, если объект является либо типом `Superb`, либо типом `Magnificent`. В любом случае можно вызвать функцию `Say()`, что обеспечит большую степень безопасности:

```

if (ps = dynamic_cast<Superb *>(pg))
 ps->Say();

```

Напомним, что значение левой части выражения присваивания сохраняется. Поэтому значением условного выражения `if` является `ps`. Если тип приведения указан успешно, `ps` будет отличным от нуля, или истинно. Если тип приведения указан ошибочно (когда `pg` указывает на объект `Grand`), `ps` является нулевым, или ложным. Листинг 14.19 содержит полный код программы.

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Даже если ваш компилятор поддерживает библиотеку RTTI, эта возможность может быть отключена по умолчанию. Следует внимательно изучить документацию или меню опций.

Эта программа иллюстрирует важное понятие. Виртуальные функции применяют там, где это возможно, а RTTI — только в том случае, если это необходимо. Ниже приводятся результаты выполнения программы из листинга 14.19:

```

I am a superb class!!
I hold the superb value of 68!
I am a magnificent class!!!
I hold the character R and the integer 68!
I am a magnificent class!!!
I hold the character D and the integer 12!
I am a magnificent class!!!
I hold the character V and the integer 59!
I am a grand class!

```

Как видите, методы `Say()` вызываются только для классов `Superb` и `Magnificent`.

Оператор `dynamic_cast` можно применять также совместно со ссылками. При этом можно получить дополнительную выгоду: если отсутствует значение ссылки, которое соответствует типу нулевого указателя, то отсутствует и специальное значение ссылки, которое может использоваться для индикации сбоя. Вместо этого при получении требования, которое не соответствует стандартам, `dynamic_cast` выдает исключение типа `bad_cast`. Это исключение является производным от класса `exception` и определяется в заголовочном файле `typeinfo`. Поэтому оператор может применяться следующим образом (здесь `pg` представляет собой ссылку на объект `Grand`):

```

#include <typeinfo> // для bad_cast
...
try {
 Superb & rs = dynamic_cast<Superb &>(rg);
 ...
}
catch(bad_cast &){
 ...
};

```

#### Оператор `typeid` и класс `type_info`

Оператор `typeid` позволяет уточнить, будут ли два объекта иметь один и тот же тип. В некотором смысле он функционирует подобно `sizeof`, применяя аргументы двух видов:

- Наименование класса
- Выражение, оценивающее объект

Оператор `typeid` возвращает ссылку на объект `type_info`, где `type_info` представляет собой класс, определенный в заголовочном файле `typeinfo` (бывший `typeinfo.h`). Класс `type_info` перегружает операторы `==` и `!=`. Поэтому данные операторы могут применяться для сравнения типов. Например, выражение

```

typeid(Magnificent) == typeid(*pg)

```

позволяет оценивать величину `bool` следующим образом. Эта величина имеет значение `true`, если `pg` отмечает объект `Magnificent`, и `false` — в противном случае. Если `pg` окажется нулевым указателем, программа обратится к исключению `bad_typeid`.

## Листинг 14.19 Программа rtti1.cpp.

```

// rtti1.cpp - применяется оператор RTTI dynamic_cast
#include <iostream>
using namespace std;
#include <cstdlib>
#include <ctime>

class Grand
{
private:
 int hold;
public:
 Grand(int h = 0) : hold(h) { }
 virtual void Speak() const { cout << "I am a grand class!\n"; }
 virtual int Value() const { return hold; }
};

class Superb : public Grand
{
public:
 Superb(int h = 0) : Grand(h) { }
 void Speak() const { cout << "I am a superb class!!\n"; }
 virtual void Say() const
 { cout << "I hold the superb value of " << Value() << "!\n"; }
};

class Magnificent : public Superb
{
private:
 char ch;
public:
 Magnificent(int h = 0, char c = 'A') : Superb(h), ch(c) { }
 void Speak() const { cout << "I am a magnificent class!!!\n"; }
 void Say() const { cout << "I hold the character " << ch
 << " and the integer " << Value() << "!\n"; }
};

Grand * GetOne();
int main()
{
 srand(time(0));
 Grand * pg;
 Superb * ps;
 for (int i = 0; i < 5; i++)
 {
 pg = GetOne();
 pg->Speak();
 if(ps = dynamic_cast<Superb *>(pg)
 ps->Say();
 }
 return 0;
}

Grand * GetOne() // генерирует случайным образом один из трех видов объектов
{
 Grand * p;
 switch(rand() % 3)
 {
 case 0: p = new Grand(rand() % 100);
 break;
 case 1: p = new Superb(rand() % 100);
 break;
 case 2: p = new Magnificent(rand() % 100, 'A' + rand() % 26);
 break;
 }
 return p;
}

```

## Листинг 14.20 Программа rtti2.cpp.

```

// rtti2.cpp - применяется dynamic_cast, typeid и type_info
#include <iostream>
using namespace std;
#include <cstdlib>
#include <ctime>
#include <typeinfo>

class Grand
{
private:
 int hold;
public:
 Grand(int h = 0) : hold(h) { }
 virtual void Speak() const { cout << "I am a grand class!\n"; }
 virtual int Value() const { return hold; }
};

class Superb : public Grand
{
public:
 Superb(int h = 0) : Grand(h) { }
 void Speak() const { cout << "I am a superb class!!\n"; }
 virtual void Say() const
 { cout << "I hold the superb value of " << Value() << "!\n"; }
};

class Magnificent : public Superb
{
private:
 char ch;
public:
 Magnificent(int h = 0, char c = 'A') : Superb(h), ch(c) { }
 void Speak() const { cout << "I am a magnificent class!!!\n"; }
 void Say() const { cout << "I hold the character " << ch
 << " and the integer " << Value() << "!\n"; }
};

Grand * GetOne();
int main()
{
 srand(time(0));
 Grand * pg;
 Superb * ps;
 for (int i = 0; i < 5; i++)
 {
 pg = GetOne();
 cout << "Now processing type " << typeid(*pg).name() << ".\n";
 pg->Speak();
 if(ps = dynamic_cast<Superb *>(pg))
 ps->Say();
 if (typeid(Magnificent) == typeid(*pg))
 cout << "Yes, you're really magnificent.\n";
 }
 return 0;
}
Grand * GetOne()
{
 Grand * p;
 switch(rand() % 3)
 {
 case 0: p = new Grand(rand() % 100); break;
 case 1: p = new Superb(rand() % 100); break;
 case 2: p = new Magnificent(rand() % 100, 'A' + rand() % 26); break;
 }
 return p;
}

```

Этот тип исключения является производным от класса `exception` и объявляется в заголовочном файле `typeinfo`.

Реализации класса `type_info` вариируются среди поставщиков, но все они содержат элемент `name()`, который возвращает зависящую от реализации строку. Обычно эта строка и является наименованием класса. Например, оператор

```
cout << "Now processing type "
 << typeid(*pg).name() << ".\n";
```

отображает строку, которая определена для класса объекта, который отмечен указателем `pg`.

Листинг 14.20 модифицирует листинг 14.19 таким образом, что в нем применяются оператор `typeid` и функция-элемент `name()`. Обратите внимание, что они используются в тех ситуациях, когда функции `dynamic_cast` и `virtual` не обрабатываются. Тестирование с помощью `typeid` выполняется для выбора действия, которое даже не является методом класса, поэтому оно может быть активизировано с помощью указателя класса. Метод оператора `name()` показывает, каким образом можно применить этот метод в процессе отладки. Обратите внимание, что программа содержит заголовочный файл `typeinfo`.

Результаты выполнения программы из листинга 14.20:

```
Now processing type Magnificent.
I am a magnificent class!!!
I hold the character P and the integer 52!
Yes, you're really magnificent.

Now processing type Superb.
I am a superb class!!!
I hold the superb value of 37!

Now processing type Grand.
I am a grand class!

Now processing type Superb.
I am a superb class!!!
I hold the superb value of 18!

Now processing type Grand.
I am a grand class!
```

### Проблемы, возникающие при использовании RTTI

Разработчики и пользователи C++ располагают большим количеством критических замечаний по поводу библиотеки RTTI. Они относятся к RTTI как к необязательному дополнению, которое является потенциальной причиной снижения эффективности работы программы. RTTI также вызывает нарекания программистов, поскольку, по их мнению, оказывает негативное влияние на качество их работы. Не желая углубляться в дискуссии по поводу RTTI, рассмотрим такие программные примеры, которых следует избегать. Обратим взоры на ядро листинга 14.19:

```
Grand * pg;
Superb * ps;
for (int i = 0; i < 5; i++)
{
 pg = GetOne();
 pg->Speak();
 if (ps = dynamic_cast<Superb *>(pg))
 ps->Say();
}
```

С помощью `typeid`, исключения `dynamic_cast` и виртуальных функций, этот код можно переписать следующим образом:

```
Grand * pg;
Superb * ps;
Magnificent * pm;
for (int i = 0; i < 5; i++)
{
 pg = GetOne();
 if (typeid(Magnificent) == typeid(*pg))
 {
 pm = (Magnificent *) pg;
 pm->Speak();
 pm->Say();
 }
 else if (typeid(Superb) == typeid(*pg))
 {
 ps = (Superb *) pg;
 ps->Speak();
 ps->Say();
 }
 else
 pg->Speak();
}
```

Этот код не только имеет непривлекательный вид, но и более длинен по сравнению с исходным кодом, кроме того, имеется серьезный недостаток в точности наименования каждого класса. Предположим, например, что вам необходимо произвести класс `Insufferable` от класса `Magnificent`. Новый класс переопределяет функции `Speak()` и `Say()`. С помощью версии, которая для точного тестирования каждого типа использует `typeid`, необходимо модифицировать код цикла `for`, добавляя новый раздел `else if`. Однако исходная версия не нуждается в каких-либо изменениях. Оператор

```
pg->Speak();
```

функционирует для всех классов, которые происходят от `Grand`, а оператор

```
if (ps = dynamic_cast<Superb *>(pg))
 ps->Say();
```

функционирует для всех классов, которые происходят от класса `Superb`.



#### СОВЕТ

Если вы обнаружите, что `typeid` используется в расширенном наборе операторов `if else`, уточните, следует ли применять виртуальные функции и `dynamic_cast`.

## Операторы приведения типов

Оператор приведения C, по мнению Бjarne Страуструпа (Bjarne Stroustrup), является слишком неопределенным. Например, рассмотрим следующие выражения:

```
struct Data
{
 double data[200];
};

struct Junk
{
 int junk[100];
};

Data d = { 2.5e33, 3.5e-19, 20.2e32 };

// оператор приведения #1 - преобразование
// числовых данных в строковые
char * pch = (char *) (&d);

// оператор приведения #2 - преобразование
// адреса структуры в тип данных char
char ch = char (&d);

// оператор приведения #3 - преобразование
// числовых данных в указатель для массива Junk
Junk * pj = (Junk *) (&d);
```

Сначала рассмотрим, какой из этих трех типов приведения имеет смысл? Если не выполнить повторную сортировку, ни один из этих типов приведения не будет иметь особого смысла. Затем уточним, какой из этих трех типов приведения разрешен? Все они являются разрешенными.

Страуструп отвечает на эту неопределенность, добавляя четыре типа операторов приведения. Такой подход внесет в процесс приведения больше порядка:

```
dynamic_cast
const_cast
static_cast
reinterpret_cast
```

Вместо применения общего типа приведения можно выделить оператор, предназначенный для определенной цели. Такой подход будет способствовать выделению той причины, которая положена в основу приведения типов, даст компилятору шанс убедиться в том, что вы делали именно то, что нужно.

Вам уже встречался оператор `dynamic_cast`. Чтобы сделать выводы, предположим, что `High` и `Low` представляют собой два класса, причем `ph` является типом `High *`, а `pl` — типом `Low *`. Тогда оператор

```
pl = dynamic_cast<Low *> ph;
```

присваивает указатель `Low *` для `pl`, только если `Low` представляет собой доступный базовый класс (прямой или косвенный) для `High`. В противном случае оператор присваивает `pl` нулевой указатель. Оператор имеет следующий синтаксис:

```
dynamic_cast <имяТипа> (выражение)
```

Этот оператор применяется для того, чтобы обеспечить нахождение верхнего приведения в пределах иерархии классов (подобные типы приведения будут безопасными благодаря взаимоотношению *is-a*). Оператор также применяется в целях отмены разрешения для других приведений.

Оператор `const_cast` предназначен для образования типа приведения, которое выполняет единственную задачу — позволяет выяснить, имеет ли величина значение `const` или `volatile` либо нет. У этого оператора такой же синтаксис, как у оператора `dynamic_cast`:

```
const_cast <имяТипа> (выражение)
```

В результате выполнения подобного приведения ошибка появляется тогда, когда изменяется какой-либо иной аспект типа. Иначе говоря, *имяТипа* и *выражение* должны иметь тот же самый тип, за исключением того случая, когда они имеют значение `const` или `volatile` либо не имеют ни того, ни другого значения. Снова предположим, что `High` и `Low` являются двумя классами:

```
const High bar;
...
High * pb = (const High *) (&bar);
// достоверно
Low * pl = (const Low *) (&bar);
// недостоверно
```

Первый тип приведения образует `*pb` — указатель, применяемый для изменения величины, которая определяет объект; при этом удаляется метка `const`. Второй тип приведения недостоверен, поскольку также предпринимается попытка изменения типа `High *` на `Low *`.

Причина существования этого оператора состоит в том, что иногда возникает необходимость в наличии особой величины. Эта величина является постоянной большую часть времени, но может изменяться от случая к случаю. Тогда эту величину можно объявить как `const` и применять `const_cast`, если возникает необходимость изменения значения. Этого можно добиться путем общего приведения, но общее приведение может одновременно изменять и тип:

```
const High bar;
...
High * pb = (const High *) (&bar);
// достоверно
Low * pl = (const Low *) (&bar);
// также достоверно
```

Поскольку одновременное изменение типа и природы постоянства может привести при программировании к нежелательным последствиям, безопаснее применять оператор `const_cast`.

Оператор `static_cast` имеет такой же синтаксис, как и другие операторы:

`static_cast <имяТипа> (выражение)`

Этот синтаксис является достоверным только в том случае, если `type_name` можно неявно конвертировать к такому же типу, к которому принадлежит `expression`, или наоборот. В противном случае приведение является ошибочным. Предположим, что `High` является базовым классом для `Low` и `Pond` не состоит с ним ни в каких отношениях. Тогда преобразования из `High` в `Low`, из `Low` в `High` корректны, а преобразование из `Low` в `Pond` не разрешается:

```
const High bar;
const Low blow;
...
High * pb = static_cast<High *> (&blow);
 // корректное верхнее приведение
Low * pl = static_cast<Low *> (&bar);
 // корректное нижнее приведение
Pond * pmer = static_cast<Pond *> (&blow);
 // недостоверное
```

Первое преобразование справедливо, поскольку приведение вверх может быть выполнено в явном виде. Второе преобразование, от указателя базового класса к указателю производного класса, не может быть реализовано без выполнения преобразования в явном виде. Но поскольку приведение типа в другом направлении может выполняться без приведения типа, правильным решением было бы применение `static_cast` для приведения вниз.

Аналогично, поскольку перечислимый тип может конвертироваться в интегральный тип без приведения типа, интегральный тип с помощью `static_cast` может конвертироваться в перечислимый тип.

Оператор `reinterpret_cast` предназначен для выполнения приведений типов, которые выполняются с определенной долей риска. Тогда нельзя будет выполнить приведение от `const`, но возможно выполнение других малопривлекательных операций. Иногда программист вынужден заниматься проблемами, тесно связанными с внедрением, осуществлением малопривлекательных операций. Благодаря применению оператора `reinterpret_cast` значительно упрощается процесс отслеживания этих действий. Синтаксис аналогичен трем другим вариантам:

`reinterpret_cast <имяТипа> (выражение)`

Ниже приводится пример использования оператора:

```
struct dat { short a; short b } ;
long value = 0xA224B118;
dat * pd = reinterpret_cast< dat *>
 (&value);
cout << pd->a; // отображает первые
 // два байта значения
```

Обычно подобные приведения выполняются при решении задач программирования на низком уровне, непосредственно связанных с внедрением, и не должны

обладать переносимостью. Например, этот образец кода приводит к генерации другого вывода при его выполнении на IBM-совместимом компьютере и на платформе Macintosh. Дело в том, что в этих двух системах байты сохраняются в мультибайтовых целых типах данных в противоположном порядке.

## Резюме

Дружественные отношения позволяют разработать более гибкий интерфейс для классов. Класс может иметь другие функции, другие классы и функции-элементы других классов в качестве дружественных конструкций. В некоторых случаях может возникнуть необходимость в применении опережающих объявлений. Необходимо также заботиться об упорядочении объявлений классов и методов. Такой подход должен способствовать более удобному получению дружественных конструкций.

Вложенные классы представляют собой классы, объявленные в пределах других классов. Вложенные классы позволяют конструировать классы помощников, которые внедряют другие классы, но при этом не обязательно должны быть частью общедоступного интерфейса.

Механизм исключений C++ поддерживает довольно удобный способ решения различных проблем программирования типа несоответствия значений, неудачных попыток ввода/вывода и др. В случае применения исключения прерывается текущее выполнение функции и управление передается соответствующему блоку `catch`. Блоки `catch` непосредственно следуют за блоком `try`. Для перехвата исключения вызов функции, который прямым или непрямым образом приводит к исключению, должен находиться в блоке `try`. Затем программа выполняет код в блоке `catch`. Этот код может попытаться устранить проблему или же прервать программу. Класс можно сконструировать с помощью вложенных классов исключений, которые используются при выявлении проблем, свойственных классу. Функция может содержать спецификацию исключения, идентифицирующую исключения, которые могут применяться в этой функции. Неперехваченные исключения (которые не соответствуют блоку `catch`) по умолчанию прерывают программу. Такой же эффект проявляется в случае непредвиденных исключений (не входящих в спецификацию исключений).

Библиотека RTTI (runtime type information) позволяет программе уточнять тип объекта. Оператор `dynamic_cast` применяется для приведения указателя производного класса к указателю базового класса; основная цель этой процедуры — гарантировать успешность вызова виртуальной функции. Оператор `typeid` возвращает

щает объект `type_info`. Два оператора `typeid` возвращают значения, которые можно сравнивать для уточнения, принадлежит ли объект к определенному типу. Возвращенный объект `type_info` может применяться для получения информации об объекте.

Оператор `dynamic_cast` наряду с операторами `static_cast`, `const_cast` и `reinterpret_cast` поддерживает более надежные, более удачные приведения типа по сравнению с общепринятым механизмом приведения.

## Вопросы для повторения

1. Какие ошибки проявились при следующих попытках установления дружественных отношений?

```
a. class snap {
 friend clasp;
 ...
};

class clasp { ... };

b. class cuff {
public:
 void snip(muff &) { ... }
 ...
};

class muff {
 friend void cuff::snip(muff &);
 ...
};

c. class muff {
 friend void cuff::snip(muff &);
 ...
};

class cuff {
public:
 void snip(muff &) { ... }
 ...
};
```

2. Вы знаете, как создаются взаимно дружественные классы. Можно ли создать более ограниченную форму дружественных отношений, где только некоторые элементы класса B являются дружественными для класса A, а некоторые элементы A являются дружественными для класса B? Объясните создавшуюся ситуацию.

3. Какие проблемы могут последовать за следующим объявлением вложенного класса?

```
class Ribs
{
private:
 class Sauce
 {
 int soy;
 int sugar;
 public:
 Sauce(int s1, int s2) :
 soy(s1), sugar(s2) { }
 };
 ...
};
```

4. Чем `throw` отличается от `return`?
5. Предположим, имеется иерархия классов исключения, производная от базового класса исключения. В каком порядке следует разместить блоки `catch`?
6. Рассмотрите в этой главе классы `Grand`, `Superb` и `Magnificent`. Предположим, что `pg` является указателем типа `Grand *`, который присваивает адрес объекта одному из этих трех классов, и `ps` является указателем типа `Superb *`. Чем отличается поведение следующих двух образцов программного кода?

```
if (ps = dynamic_cast<Superb *>(pg))
 ps->say(); // образец #1

if (typeid(*pg) == typeid(Superb))
 (Superb *) pg->say(); // образец #2
```

7. В чем отличие оператора `static_cast` от оператора `dynamic_cast`?

## Упражнения по программированию

1. Модифицируйте классы `Tv` и `Remote` следующим образом:

- а. Сформируйте взаимно дружественные отношения.
- б. Добавьте к классу `Remote` элемент переменной состояния, который описывает, будет ли удаленный контроль проходить в обычном или в интерактивном режиме.
- в. Добавьте метод `Remote`, который отображает режим.
- г. Поддержите класс `Tv` с помощью метода по переключению нового элемента `Remote`. Этот метод должен функционировать только в том случае, если `Tv` находится в подключенном состоянии.

Напишите короткую программу по тестированию этих новых возможностей.

2. Модифицируйте листинг 14.10 таким образом, чтобы функция `hmean( )` приводила к исключению типа `hmeanexp`, а функция `gmean( )` приводила к исключению типа `gmeanexp`. Оба эти типа исключения являются производными от класса исключения. С помощью перехвата исключения `hmeanexp` программа также должна отобразить подсказку о вводе новой пары данных и продолжать выполнение. При перехвате исключения `gmeanexp` программа должна продолжать выполнение программного кода, который следует за циклом.

# Класс `string` и стандартная библиотека шаблонов

**В этой главе рассматривается следующее:**

- Стандартный класс языка C++ `string`
- Шаблон `auto_ptr`
- Стандартная библиотека шаблонов (STL)
- Классы-контейнеры
- Итераторы
- Объекты-функции (функторы)
- Алгоритмы STL

Вы уже знакомы с одной из возможностей языка C++, заключающейся в многократном использовании программного кода. Одно из главных достоинств этого языка — возможность использовать код, написанный другими программистами. Именно так и появляются библиотеки классов. Существует много доступных коммерческих библиотек классов C++, а некоторые библиотеки являются частью пакета C++. Например, в C++ используются классы ввода/вывода, поддерживающие в заголовочном файле `ostream`. В этой главе будут приведены примеры многократно используемого кода, позволяющего упростить процесс программирования. Вначале рассмотрен класс `string`, упрощающий обработку строк. Затем описан класс шаблонов `auto_ptr`, являющийся "интеллектуальным указателем", реализующим простое управление динамической памятью. И наконец, рассмотрена стандартная библиотека шаблонов (STL — Standard Template Library), коллекция полезных шаблонов для работы с различными типами объектов-контейнеров. Библиотека STL является воплощением современной парадигмы программирования, называемой обобщенным программированием.

## Класс `string`

В главе 11 был представлен небольшой класс `String`, иллюстрирующий некоторые аспекты создания классов. Язык C++ содержит более мощную версию класса под названием `string`. Этот класс определен в заголовочном файле `string`. (Нужно отметить, что заголовочные файлы `string.h` и `cstring` поддерживают библиотеку функций языка C для работы со строками в стиле языка C, С-строки, а не класс `string`.) Ключом к использованию является изучение его общедоступных интерфейсов. Класс `string` содержит обширный набор методов, включая несколько конструкторов, перегруженные операторы, выполняющие присваивание, конкатенацию, сравнение строк и осуществляющие доступ к отдельным элементам, а также утилиты для поиска символов и подстрок в строке и многое другое. Одним словом, класс `string` содержит много полезных элементов.

### Создание строки

Мы начнем с рассмотрения конструкторов класса `string`. В конце концов, что наиболее важно знать о классе, так это то, какой имеется выбор при создании объектов этого класса. В листинге 15.1 используются все шесть конструкторов `string` (обозначенных `ctor`, традиционной в C++ аббревиатурой для обозначения конструкторов).

**Листинг 15.1 Программа str1.cpp.**

```
// str1.cpp - введение в класс string
#include <iostream>
#include <string>
using namespace std;
// использование строковых конструкторов
int main()
{
 string one("Lottery Winner!"); //конструктор #1
 cout << one << endl; // перегруженный
 // оператор <<
 string two(20, '$'); //конструктор #2
 cout << two << endl;
 string three(one); //конструктор #3
 cout << three << endl;
 one += " Oops!"; //перегруженный оператор +=
 cout << one << endl;
 two = "Sorry! That was ";
 three[0] = 'P';
 string four; // конструктор #4
 four = two + three; // перегруженные
 // операторы +, =
 cout << four << endl;
 char alls[] = "All's well that ends well";
 string five(alls,20); // конструктор #5
 cout << five << "\n";
 string six(alls+6, alls + 10);
 //конструктор #6
 cout << six << ", ";
 string seven(&five[6], &five[10]);
 //и снова конструктор #6
 cout << seven << "...\\n";
 return 0;
}
```

В табл. 15.1 кратко описаны конструкторы в порядке их использования в программе. Представления конструкторов упрощены, и поэтому они скрывают тот факт, что **string** реально является определением **typedef**, используемым для специализации шаблона **basic\_string<char>**. Здесь также не показан дополнительный аргумент, связанный с управлением памятью. (Этот аспект обсуждается далее в этой главе и в приложении F.) Тип **size\_type** — это зависящий от реализации интегральный тип, определенный в строковом заголовочном файле. Класс определяет **string::npos** как максимальную возможную длину строки. Обычно она равна максимальному значению, допускаемому типом **unsigned int**. Кроме того, в таблице используется рас пространенная аббревиатура **NBTS** для строки, завершающейся нулем, т.е. традиционной строки языка С, в конце которой всегда находится нулевой символ.

Программа также использует перегруженный оператор **+=** для добавления одной строки к другой, перегруженный оператор **=** для присвоения одной строки другой, перегруженный оператор **<<** для вывода строкового объекта и перегруженный оператор **[ ]** для выполнения доступа к отдельному элементу строки.

 **ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**  
Более ранние реализации класса **string** не поддерживают конструктор #6.

Результаты выполнения программы:

```
Lottery Winner!
$$$$$$$$$$$$$$$$$$$$$
Lottery Winner!
Lottery Winner! Oops!
Sorry! That was Pottery Winner!
All's well that ends!
well, well...
```

**Примечания к программе**

Начало программы иллюстрирует, что объект класса **string** можно инициализировать с помощью стандартной С-строки и вывести его на печать с помощью перегруженного оператора **<<**:

**Таблица 15.1 Конструкторы класса string.**

| Конструктор                                                              | Описание                                                                                                                                                                                                                                |
|--------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>string(const char * s)</b>                                            | Инициализирует объект класса <b>string</b> строкой NBTS с указателем <b>s</b> .                                                                                                                                                         |
| <b>string(size_type n, char c)</b>                                       | Создает объект класса <b>string</b> размером <b>n</b> элементов, каждый из которых инициализируется символом <b>c</b> .                                                                                                                 |
| <b>string(const string * str, size_type pos = 0, size_type n = npos)</b> | Инициализирует объект класса <b>string</b> объектом <b>str</b> , начиная с позиции <b>pos</b> в объекте <b>str</b> и завершая <b>str</b> или копирует <b>n</b> символов, в зависимости от того, какое событие произойдет раньше.        |
| <b>string()</b>                                                          | Создает объект класса <b>string</b> нулевого размера.                                                                                                                                                                                   |
| <b>string(const string * str, size_type n)</b>                           | Инициализирует объект класса <b>string</b> NBTS с указателем <b>s</b> и длиной <b>n</b> символов, даже если это превышает размер NBTS.                                                                                                  |
| <b>template &lt;class Iter&gt; string (Iter begin, Iter end)</b>         | Инициализирует объект класса <b>string</b> значениями в диапазоне <b>[begin, end]</b> , где <b>begin</b> и <b>end</b> действуют как указатели; диапазон включает в себя <b>begin</b> и простирается до <b>end</b> , но не включает его. |

```
string one("Lottery Winner!"); //конст-р #1
cout << one << endl; // перегруженный
// оператор <<
```

Следующий конструктор инициализирует объект two класса `string`, состоящий из 20 символов \$:

```
string two(20,$); // конструктор #2
```

Конструктор копирования инициализирует объект three класса `string` объектом one:

```
string three(one); // конструктор #3
```

Перегруженный оператор += добавляет строку "Oops!" к строке one:

```
one += "Oops!"; //перегруженный оператор +=
```

В данном примере к объекту класса `string` добавляется С-строка. Однако оператор += многократно перегружен, поэтому к объектам класса `string` можно добавлять и строки, и отдельные символы:

```
//добавление строкового объекта(не в программе)
one += two;
//добавление значения типа char(не в программе)
one += '!';
```

Оператор = тоже перегружен таким образом, что объекту класса `string` можно присвоить объект класса `string`, С-строку или же просто значение типа `char`:

```
//присвоение С-строки
two = "Sorry! That was";
//присвоение объекта string(не в программе)
two = one;
//присвоение значения типа char(не в программе)
two = '?';
```

Переопределение оператора [ ], подобное сделанному в примере `ArrayDb` в главе 13, позволяет получить доступ к отдельным символам объекта `string`, используя форму записи массива:

```
three[0] = 'P';
```

Конструктор, заданный по умолчанию, создает пустую строку, которой в последующем можно присвоить значение:

```
string four; // конструктор #4
four = two + three; // перегруженные
// операторы +, =
```

Во второй строке используется перегруженный оператор + для создания временного объекта `string`, который затем присваивается с помощью перегруженного оператора = объекту four. Как и можно ожидать, оператор + выполняет конкатенацию двух своих операндов в один строковый объект. Этот оператор является многократно перегруженным, поэтому второй операнд может быть объектом класса `string`, С-строкой или же значением типа `char`.

Пятый конструктор получает в качестве аргументов С-строку и целое значение, определяющее, сколько символов необходимо скопировать:

```
char alls[] = "All's well that ends well";
string five(alls,20); // конструктор #5
```

Здесь, как показывает вывод, только первые 20 символов ("All's well that ends") используются для инициализации объекта five. Как отмечено в табл. 15.1, если количество символов превышает длину С-строки, запрашиваемое число символов все равно копируется. Поэтому замена 20 на 40 в приведенном примере дает 15 лишних символов, скопированных в конец строки five (т.е. конструктор будет интерпретировать содержимое памяти строкой "All's well that ends well" как коды символов).

Шестой конструктор включает аргумент шаблона:

```
template<class Iter> string(Iter begin,
 Iter end);
```

Смысл в том, что `begin` и `end` действуют как указатели на две ячейки памяти. (Вообще, `begin` и `end` могут быть итераторами, обобщением указателей, широко используемых в STL.) Затем конструктор использует значения между ячейками памяти, на которые указывают `begin` и `end`, для инициализации создаваемого им строкового объекта. Форма записи `[begin, end)`, заимствованная из математики, обозначает, что диапазон включает `begin`, но не включает `end`, т.е. `end` указывает на точку за последним используемым значением. Рассмотрим следующее выражение:

```
string six(alls + 6, alls + 10); // #6
```

Поскольку имя массива является указателем, а `alls + 6` и `alls + 10` имеют тип `char *`, шаблон Iter заменяется типом `char *`. Первый аргумент указывает на первый символ w в массиве alls, а второй — на пробел, следующий за well. Таким образом, six инициализируется строкой "well". На рис. 15.1 показано, как работает конструктор.

Теперь предположим, что нужно использовать такой конструктор для инициализации объекта частью другого объекта класса `string`, скажем, объекта five. Следующее выражение не имеет смысла:

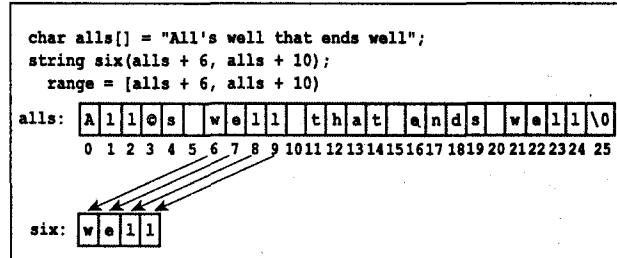


РИС. 15.1 Конструктор класса `string`, использующий диапазон.

```
string seven(five + 6, five + 10);
```

Причина состоит в том, что имя объекта, в отличие от имени массива, не трактуется как адрес объекта, поэтому `five` не является указателем, а `five + 6` не имеет смысла. Однако `five[6]` — значение типа `char`, поэтому `&five[6]` является его адресом и может быть использовано как аргумент конструктора:

```
string seven(&five[6], &five[10]);
// снова конструктор #6
```

## Реализация ввода в классе string

Следующее, что полезно знать о классе, — это то, какие возможности доступны для выполнения ввода данных. При использовании С-строк существует три возможности выполнения ввода данных:

```
char info[100];

// прочесть слово
cin >> info;

// прочесть строку, опустив \n
cin.getline(info, 100);

// прочесть строку, оставить \n в конце
cin.get(info, 100);
```

Каковы же возможности при выполнении ввода для объекта класса `string`? Прежде всего, строковый класс перегружает оператор `>>` почти так же, как это было сделано для класса `String` в главе 11. Поскольку первый operand не является объектом класса `string`, перегруженный оператор-функция `>>` не является методом класса. Он представляет собой общую функцию, воспринимающую объект класса `istream` как первый аргумент и объект класса `string` — как второй. Чтобы обеспечить совместимость с методами обработки оператором `>>` С-строк, версия класса `string` также читает отдельное слово, прерывая операцию ввода при достижении символа пробела, обнаружении признака конца файла или достижении максимального количества символов, которое можно сохранить в строковом объекте. Функция работает следующим образом: сначала удаляет содержимое строки, а затем читает и добавляет по одному символу. До тех пор пока входная последовательность короче, чем возможное количество символов для объекта класса `string`, функция `operator>>(istream &, string &)` автоматически подгоняет размеры объекта `string` под входную строку. Результат этого проявляется в том, что оператор `>>` можно использовать со строковым объектом так же, как это делается при работе с С-строками, но при этом не возникает угроза переполнения массива:

```
char fname[10];
cin >> fname; // может быть проблема, если
// входная строка содержит больше 9 символов
```

```
string lname;
cin >> lname; // может прочесть очень
// длинное слово
```

Использование эквивалента функции `getline()` не столь очевидно, поскольку `getline()`, в отличие от `operator>>()`, нельзя использовать в операторной записи. Вместо этого применяется запись метода класса:

```
cin.getline(fname, 10);
```

Чтобы получить один и тот же синтаксис для работы с объектом класса `string`, потребуется новая функция — элемент класса `istream`, что не является мудрым решением. Вместо этого строковая библиотека определяет функцию, не являющуюся элементом класса `getline()`, имеющую объект класса `istream` первым аргументом и объект класса `string` — вторым. Таким образом, она используется, как показано ниже, для чтения строки ввода объектом класса `string`:

```
string fullName;
getline(cin, fullName); // вместо
// cin.getline(fname, 10);
```

Функция `getline()` вначале удаляет содержимое выходной строки и затем читает по одному символу из входного потока, добавляя символы к строке. Этот процесс продолжается до тех пор, пока функция не достигнет конца строки либо пока она не обнаружит признак конца файла, либо пока не будет заполнен строковый объект. Если обнаружен символ новой строки, то он считывается, но не сохраняется в строке. Заметьте, что, в отличие от версии `istream`, версия класса `string` не имеет параметра размера, указывающего максимальное количество символов для чтения. Это объясняется тем, что данная функция автоматически подгоняет размер объекта `string` для потока ввода. Листинг 15.2 иллюстрирует применение двух возможностей для ввода.

### Листинг 15.2 Программа str2.cpp.

```
// str2.cpp — строковый ввод
#include <iostream>
#include <string>
using namespace std;

int main()
{
 string word;
 cout << "Enter a line: ";
 cin >> word;
 while (cin.get() != '\n')
 continue;
 cout << word << " is all I wanted.\n";

 string line;
 cout << "Enter a line (really!): ";
 getline(cin, line);
 cout << "Line: " << line << endl;
 return 0;
}
```

## ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Microsoft Visual C++ 5.0 имеет ошибку в реализации функции `getline()`, приводящую к тому, что вывод информации, следующий за выполнением функции `getline()`, не проявится вновь до тех пор, пока не будет введен какой-либо символ. Функция `getline()` в Borland C++ Builder 1.0 запрашивает ввод дополнительного символа-разделителя в качестве аргумента: `getline(cin, line, '\n')`.

Результаты выполнения программы:

```
Enter a line: Time and tide wait for no one.
Time is all I wanted.
Enter a line (really!): All things come to he
who waits.
Line: All things come to he who waits.
```

## Работа со строками

К этому моменту вы узнали, что можно создавать объекты класса `string` разными способами, выводить содержимое этих объектов, читать в них данные, добавлять строковые объекты к ним, присваивать другим строковым объектам и конкатенировать два строковых объекта. Что же еще можно делать в этом случае?

Можно сравнивать строки. Все шесть операторов сравнения перегружены для объектов класса `string`, причем один объект предполагается меньшим, чем другой, если он встречается раньше в последовательности сравнения. Если последовательность сравнения включает ASCII-коды, то считается, что коды цифр меньше кодов заглавных букв, а коды заглавных букв меньше кодов строчных букв. Каждый оператор сравнения перегружен тремя способами, поэтому можно сравнивать объект класса `string` с объектом класса `string`, объект класса `string` — с C-строкой и C-строку — с объектом класса `string`:

```
string snake1("cobra");
string snake2("coral");
char snake3[20] = "anaconda";
```

```
if (snake1 < snake2) // operator<(const
 // string &, const string &)
...
if (snake1 == snake3) // operator==(const
 // string &, const char *)
...
if (snake3 != snake2) // operator!=(const
 // char *, const string &)
...
```

Можно определить размер строки. Обе функции-элемента класса, `size()` и `length()`, возвращают количество символов в строке:

```
if (snake1.length() == snake2.size())
 cout << "Both strings have the
 same length.\n"
```

Почему две функции делают одно и то же? Функция-элемент `length()` берет свое начало в предыдущих версиях класса `string`, в то время как функция `size()` добавлена для совместимости с библиотекой STL.

Можно осуществлять поиск заданной подстроки или символа в строке разными способами. В табл. 15.2 представлено короткое описание четырех вариаций метода `find()`. Вспомните, что функция-элемент `string::npos` определяет максимально возможное количество символов в строке, обычно наибольшее значение типа `unsigned int` или `unsigned long`.

В библиотеке также содержатся родственные методы `rfind()`, `find_first_of()`, `find_last_of()`, `find_first_not_of()` и `find_last_not_of()`, каждый из которых перегружен с таким же набором сигнатур функций, как и метод `find()`. Метод `rfind()` находит последнее вхождение подстроки или символа. Метод `find_first_of()` находит первое вхождение в исходной строке любого из символов, содержащихся в аргументе. Например, выражение

```
int where = snake1.find_first_of("hark");
```

Таблица 15.2 Перегруженный метод `find()`.

| Прототип метода                                                                           | Описание                                                                                                                                                                                                                                                                |
|-------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>size_type find(const string &amp; str,<br/>size_type pos = 0) const</code>          | Находит первое вхождение подстроки <code>str</code> , начиная с позиции <code>pos</code> в исходной строке. Возвращает индекс первого символа, если подстрока найдена, или <code>string::npos</code> — в ином случае.                                                   |
| <code>size_type find(const char * s,<br/>size_type pos = 0) const</code>                  | Находит первое вхождение подстроки <code>s</code> , начиная с позиции <code>pos</code> в исходной строке. Возвращает индекс первого символа, если подстрока найдена, или <code>string::npos</code> — в ином случае.                                                     |
| <code>size_type find(const char * s,<br/>size_type pos = 0,<br/>size_type n) const</code> | Находит первое вхождение подстроки, состоящей из первых <code>n</code> символов строки <code>s</code> , начиная с позиции <code>pos</code> в исходной строке. Возвращает индекс первого символа, если подстрока найдена, или <code>string::npos</code> — в ином случае. |
| <code>size_type find(char ch,<br/>size_type pos = 0) const</code>                         | Находит первое вхождение символа <code>ch</code> , начиная с позиции <code>pos</code> в исходной строке. Возвращает индекс символа, если он найден, или <code>string::npos</code> — в ином случае.                                                                      |

возвратит позицию символа `r` в `cobra` (т.е. индекс 3), поскольку это первое вхождение одной из букв строки `hark` в строке `cobra`. Метод `find_last_of()` работает также, находя последнее вхождение. Поэтому выражение

```
int where = snake1.find_last_of("hark");
```

возвратит позицию символа `a` в строке `cobra`. Метод `find_first_not_of()` находит первый символ в исходной строке, не являющийся символом для аргумента. Таким образом,

```
int where = snake1.find_first_not_of("hark");
```

возвратит позицию символа `c` в строке `cobra`, так как `c` не входит в строку `hark`. Описание работы метода `find_last_not_of()` мы оставляем в качестве упражнения читателю.

Существует много и других методов, но приведенных выше достаточно, чтобы создать программу — графически упрощенную версию словесной игры Hangman (Палач). Программа содержит список слов в массиве объектов класса `string`, из которого она случайным образом выбирает слово и предлагает угадывать буквы в нем. После предпринятых шести неверных попыток игрок проигрывает. В программе используется функция `find()` для проверки угаданных букв и оператор `+=` для построения объекта класса `string`, где сохраняются сведения о неверных попытках. Для отслеживания правильных вариантов программа создает слово той же длины, что и загаданное, но состоящее из дефисов. Затем дефисы заменяются на правильно угаданные буквы. Текст программы приведен в листинге 15.3.

### Листинг 15.3 Программа str3.cpp.

```
// str3.cpp - некоторые методы работы со строками
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include <cctype>
using namespace std;
const int NUM = 26;
const string wordlist[NUM] = { "apiary", "beetle", "cereal",
 "danger", "ensign", "florid", "garage", "health", "insult",
 "jackal", "keeper", "loaner", "manage", "nonce", "onset",
 "plaid", "quilt", "remote", "stolid", "train", "useful",
 "valid", "whence", "xenon", "yearn", "zippy" };
int main()
{
 srand(time(0));
 char play;
 cout << "Will you play a word game? <y/n> ";
 cin >> play;
 play = tolower(play);
 while (play == 'y')
 {
 string target = wordlist[rand() % NUM];
 int length = target.length();
 string attempt(length, '-');
 string badchars;
 int guesses = 6;
 cout << "Guess my secret word. It has " << length
 << " letters, and you guess\n"
 << "one letter at a time. You get " << guesses
 << " wrong guesses.\n";
 cout << "Your word: " << attempt << endl;
 while (guesses > 0 && attempt != target)
 {
 char letter;
 cout << "Guess a letter: ";
 cin >> letter;
 if (badchars.find(letter) != string::npos
 || attempt.find(letter) != string::npos)
 {
 cout << "You already guessed that. Try again.\n";
 continue;
 }
 int loc = target.find(letter);
```

```

if (loc == string::npos)
{
 cout << "Oh, bad guess!\n";
 --guesses;
 badchars += letter; // добавить в строку
}
else
{
 cout << "Good guess!\n";
 attempt[loc]=letter;
 // проверка повторного отображения символов
 loc = target.find(letter, loc + 1);
 while (loc != string::npos)
 {
 attempt[loc]=letter;
 loc = target.find(letter, loc + 1);
 }
}
cout << "Your word: " << attempt << endl;
if (attempt != target)
{
 if (badchars.length() > 0)
 cout << "Bad choices: " << badchars << endl;
 cout << guesses << " bad guesses left\n";
}
if (guesses > 0)
 cout << "That's right!\n";
else
 cout << "Sorry, the word is " << target << ".\n";
cout << "Will you play another? <y/n> ";
cin >> play;
play = tolower(play);
}
cout << "Bye\n";
return 0;
}

```

Результаты выполнения программы из листинга 15.3:

```

Will you play a word game? <y/n> y
Guess my secret word. It has 6 letters,
and you guess
one letter at a time. You get 6 wrong
guesses.
Your word: -----
Guess a letter: e
Oh, bad guess!
Your word: -----
Bad choices: e
5 bad guesses left
Guess a letter: a
Good guess!
Your word: a---a-
Bad choices: e
5 bad guesses left
Guess a letter: t
Oh, bad guess!
Your word: a--a--
Bad choices: et
4 bad guesses left

```

```

Guess a letter: i
Good guess!
Your word: a--ar-
Bad choices: et
4 bad guesses left
Guess a letter: y
Good guess!
Your word: a--ary
Bad choices: et
4 bad guesses left
Guess a letter: i
Good guess!
Your word: a-iary
Bad choices: et
4 bad guesses left
Guess a letter: p
Good guess!
Your word: apairy
That's right!
Will you play another? <y/n> n
Bye

```

## Примечания к программе

Тот факт, что операторы сравнения являются перегруженными, позволяет трактовать строки так же, как и числовые переменные:

```
while (guesses > 0 && attempt != target)
```

С ними гораздо проще обращаться, чем, например, с функцией `strcmp()` для С-строк.

Программа использует функцию `find()` для проверки того, был ли символ выбран ранее; если он был выбран, то будет находиться или в строке `badchars` (неудачные попытки), или в строке `attempt` (успешные попытки):

```
if (badchars.find(letter) != string::npos
 || attempt.find(letter) != string::npos)
```

Переменная `npos` является статическим элементом класса `string`. Ее значение равно максимально возможному количеству символов в строковом объекте. Поэтому, так как индексирование начинается с 0, ее значение на 1 больше наибольшего индекса и ее можно использовать для обозначения признака неудачи при поиске символа или строки.

Программа использует тот факт, что одна из перегруженных версий оператора `+=` позволяет добавлять к строке отдельные символы:

```
// добавить символ к строковому объекту
badchars += letter;
```

Главная часть программы начинается с проверки того, есть ли определенная буква в загаданном слове:

```
int loc = target.find(letter);
```

Если `loc` имеет допустимое значение, буква будет помещена в соответствующее место в строке ответа:

```
attempt[loc]=letter;
```

Однако данная буква может встречаться в слове неоднократно, поэтому программа должна продолжить проверку. Здесь программа использует дополнительный аргумент функции `find()`, позволяющий указать стартовую позицию в строке, с которой необходимо начинать поиск. Поскольку буква была найдена в позиции `loc`, следующий поиск должен начинаться с `loc + 1`. Цикл `while` продолжает поиск до тех пор, пока не будет найдено ни одного вхождения данного символа. Заметьте, что `find()` указывает на неудачу, если `loc` находится после конца строки:

```
// проверка: присутствует ли еще раз в
// слове данная буква
loc = target.find(letter, loc + 1);
while (loc != string::npos)
{
 attempt[loc]=letter;
 loc = target.find(letter, loc + 1);
}
```

## Что еще?

Библиотека `string` поддерживает множество различных возможностей. Существуют функции для удаления части или целой строки, замены части или целой строки другой строкой или ее частью, вставки символов в строку или удаления их из строки, сравнения части или целой строки с другой строкой или ее частью и для извлечения подстроки из строки. Существует функция для копирования части одной строки в другую и функция для обмена содержимого двух строк. Большинство из этих функций перегружено, поэтому можно работать с С-строками наравне с объектами класса `string`. Библиотека `string` кратко описана в приложении F.

В этом разделе класс `string` трактовался так, как будто он базируется на типе `char`. В действительности, как упоминалось ранее, библиотека `string` базируется на классе шаблонов:

```
template<class charT,
 class traits = char_traits<charT>,
 class Allocator = allocator<charT> >
basic_string {...};
```

Этот класс включает два следующих определения `typedef`:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Это позволяет использовать строки, базирующиеся на типе `wchar_t` так же, как и на типе `char`. Более того, можно разработать некоторый класс, подобный символьному, и использовать шаблон класса `basic_string` для него при условии, что созданный класс удовлетворяет определенным требованиям. Класс `traits` — это класс, включающий основные сведения о выбранном символьном классе, например, определяющий, как сравнивать значения. Есть предопределенные специализации шаблона `char_traits` для типов `char` и `wchar_t`, они являются значениями, принятыми по умолчанию для класса `traits`. Класс `Allocator` представляет собой класс для управления распределением памяти. Существуют предопределенные специализации шаблона `allocator` для типов `char` и `wchar_t`, и они являются значениями, принятыми по умолчанию. В них функции `new` и `delete` используются стандартным образом, однако можно зарезервировать блок памяти и создать свои собственные методы для выполнения распределения памяти.

## Класс `auto_ptr`

Класс `auto_ptr` является классом шаблонов, предназначенный для управления распределением динамической памяти. Давайте посмотрим, что для этого, может быть, нужно и как это можно реализовать. Представим следующую функцию:

```
void remodel(string & str)
{
 string * ps = new string(str);
 ...
 str = ps;
 return;
}
```

Возможно, вы уже заметили ее недостаток. Каждый раз, когда вызывается эта функция, она выделяет память из "кучи", но не возвращает ее, создавая "утечку памяти". Вы также знаете и решение — просто помнить о том, что нужно освободить зарезервированную память, добавив следующее выражение непосредственно перед возвратом из функции:

```
delete ps;
```

Однако решение, включающее фразу "просто помнить", редко является лучшим решением. Иногда об этом можно забыть или помнить, но случайно удалить или закомментировать необходимый блок кода. И даже если помнить об этом всегда, проблемы остаются. Представим себе такую вариацию:

```
void remodel(string & str)
{
 string * ps = new string(str);
 ...
 if (weird_thing())
 throw exception();
 str = *ps;
 delete ps;
 return;
}
```

Если было сгенерировано прерывание, оператор `delete` не выполняется и снова возникает "утечка памяти".

Эту оплошность можно исправить, как было показано в главе 14, однако было бы лучше найти более тонкое решение. Давайте подумаем, что для этого требуется. Когда функция `remodel()` завершает выполнение или обычным образом, или в результате прерывания, локальные переменные удаляются из стековой памяти, т.е. память, занимаемая указателем `ps`, также очищается. Это значит, что программе необходимо выполнить дополнительное действие, когда необходимость в переменной `ps` отпадает. Такая дополнительная возможность не поддерживается для базовых типов, но ее можно обеспечить для классов с помощью механизма деструктора. Таким образом, проблема с `ps` проявляется в том, что это обычный указатель, а не объект класса. Если бы это был объект, можно было бы построить деструктор, удаляющий память по указателю, когда необходимость в объекте отпадает. Это и есть главная идея, реализуемая при использовании `auto_ptr`.

## Использование шаблона `auto_ptr`

Шаблон `auto_ptr` определяет объект, похожий на указатель, которому присваивается адрес, полученный (прямо или косвенно) с помощью оператора `new`. Когда объект класса `auto_ptr` прекращает свое существование, его деструктор использует оператор `delete` для освобождения памяти. Таким образом, если адрес, возвращаемый функцией `new`, присваивается объекту класса `auto_ptr`, нет необходимости помнить об освобождении памяти; она освободится автоматически, когда объект `auto_ptr` закончит свою работу. На рис. 15.2 иллюстрируется разница в поведении между объектом `auto_ptr` и обычным указателем.

Для создания объекта `auto_ptr` нужно включить заголовочный файл `memory`, в котором описан шаблон класса `auto_ptr`. Затем можно использовать обычный синтаксис для шаблона, чтобы создать тот тип указате-

```
void demo1()
{
 double * pd = new double; // #1
 *pd = 25.5; // #2
 return; // #3
}

#1: Создает место для хранения pd и значения типа double,
сохраняет адрес:
pd 10000 [] 4000 10000

#2: Копирует значение в динамическую память:
pd 10000 [] 25.5 4000 10000

#3: Удаляет указатель pd, оставляет значение
в динамической памяти:
[] 25.5 10000

void demo2()
{
 auto_ptr<double> ap(new double); // #1
 *ap = 25.5; // #2
 return; // #3
}

#1: Создает место для хранения ap и значения типа double,
сохраняет адрес:
ap 10080 [] 6000 10080

#2: Копирует значение в динамическую память:
ap 10080 [] 25.5 6000 10000

#3: Удаляет указатель ap, и деструктор освобождает
динамическую память.
```

РИСУНОК 15.2 Обычные указатели и `auto_ptr`.

ля, который необходим. Шаблон включает в себя следующее:

```
template<class X> class auto_ptr {
public:
 explicit auto_ptr(X* p = 0) throw();
 ...
};
```

(Запись `throw()` обозначает, что конструктор не вызывает прерывания.) Таким образом, запрос указателя `auto_ptr` типа `X` дает `auto_ptr`, указывающий на тип `X`:

```
// указатель auto_ptr на переменную double
// (используйте вместо double *)
auto_ptr<double> pd(new double);

// указатель auto_ptr на переменную string
// (используйте вместо string *)
auto_ptr<string> pd(new string);
```

Здесь `new double` — это указатель на новый блок памяти, значение которого возвращено оператором `new`. Это аргумент конструктора `auto_ptr<double>`; т.е. это реальный аргумент, отвечающий формальному параметру `p` в прототипе. Аналогично `new string` также является реальным аргументом для конструктора.

Таким образом, чтобы преобразовать функцию `remodel()`, нужно выполнить три следующих действия:

1. Включить заголовочный файл `memory`.
2. Заменить указатель на `string` шаблоном `auto_ptr` на переменную `string`.
3. Убрать оператор `delete`.

Вот функция со всеми выполненными изменениями:

```
#include <memory>
void remodel(string & str)
{
 auto_ptr<string> ps (new string(str));
 ...
 if (weird_thing())
 throw exception();
 str = *ps;
 // delete ps; БОЛЬШЕ НЕ ТРЕБУЕТСЯ
 return;
}
```

Заметьте, что конструктор класса `auto_ptr` имеет тип `explicit` (явный), поэтому неявное преобразование типа при присваивании указателя объекту `auto_ptr` невозможно:

```
auto_ptr<double> pd;
double *p_reg = new double;
// запрещено (неявное преобразование)
pd = p_reg;

// разрешено (явное преобразование)
pd = auto_ptr<double>(p_reg);
// запрещено (неявное преобразование)
auto_ptr<double> pauto = pd;
// разрешено (явное преобразование)
auto_ptr<double> pauto(pd);
```

Объект класса `auto_ptr` является примером *интеллектуального указателя*, т.е. объекта, действующего как указатель с дополнительными свойствами. Класс `auto_ptr` определен таким образом, что в большинстве случаев его объекты выступают как обычные указатели. Например, если `ps` является объектом класса `auto_ptr`, то можно разыменовать объект с помощью команды `(*ps)`, увеличить значение объекта на единицу `(++ps)`, воспользоваться этим объектом для доступа к элементам структуры, на которую он указывает, с помощью команды `(ps->pushIndex)` и присвоить его значение обычному указателю, указывающему на тот же тип, что и данный объект. Можно также присвоить значение одного объекта класса `auto_ptr` другому объекту такого же типа, однако при этом уже возникают вопросы, обсуждаемые в следующем разделе.

Шаблон позволяет инициализировать объект класса `auto_ptr` обычным указателем с помощью конструктора:

## Некоторые замечания

Класс `auto_ptr` не является панацеей. Например, представьте следующий программный код:

```
auto_ptr<int> pi(new int [200]); // недопустимо!
```

Помните, что необходимо использовать пары операторов: `delete` вместе с `new`, а `delete []` — с `new[]`. В шаблоне применяется `delete`, а не `delete []`, поэтому его можно использовать только с оператором `new`, а не с `new []`. Эквивалента класса `auto_ptr` для работы с динамическими массивами не существует. Можно поступить так: скопировать шаблон класса `auto_ptr` из заголовочного файла `memory`, переименовать его в `auto_arr_ptr` и изменить копию так, чтобы вместо оператора `delete` использовался оператор `delete []`. В этом случае в библиотеку можно добавить поддержку оператора `[]`.

Еще пример:

```
string vacation("I wandered lonely as a cloud.");
auto_ptr<string> pvac(&vacation);
// недопустимо!
```

В данном случае оператор `delete` будет использован для распределения памяти из свободного блока, а не из статического массива, что недопустимо.



### ПРЕДОСТЕРЕЖЕНИЕ

Используйте объект класса `auto_ptr` только при работе с памятью, выделенной оператором `new`, а не оператором `new []` или простым объявлением переменной.

Теперь рассмотрим выражение:

```
auto_ptr<string> ps (new string("I reigned
 lonely as a cloud."));
auto_ptr<string> vocation;
vocation = ps;
```

Какую именно роль должно играть присваивание в данном выражении? Если бы `ps` и `vocation` были обычными указателями, то в результате они указывали бы на один и тот же объект класса `string`. Такое решение не применимо здесь, поскольку в итоге программа попытается удалить один и тот же объект дважды: первый раз, когда `ps` прекращает свое существование, и второй — когда прекращается существование `vocation`. Для избежания возникновения такой ситуации существует несколько способов:

- Определить оператор присваивания так, чтобы он создавал копию. Результатом такого действия будут два указателя, указывающие на два различных объекта, один из которых является копией другого.
- Ввести концепцию *собственности*, когда только одному интеллектуальному указателю разрешено владеть определенным объектом. Только в том случае, когда указатель владеет объектом, его конструктор сможет удалить объект. При присвоении указателя переносится и право владения объектом. Именно такая стратегия применена в классе `auto_ptr`.
- Создать "более интеллектуальный" указатель, который будет следить за тем, сколько "интеллектуальных" указателей ссылаются на определенный объект. Такая реализация называется *подсчетом ссылок*. В результате присваивания, например, увеличивается значение счетчика на единицу, а после прекращения существования указателя счетчик уменьшается на единицу. И лишь когда прекратит свое существование последний указатель, данный объект будет удален.

Подобные стратегии конечно же можно применить и к конструкторам копирования.

Каждый подход находит свое применение. Вот пример ситуации, когда использование объектов класса `auto_ptr` может привести к неправильной работе:

```
auto_ptr<string> films[5] =
{
 auto_ptr<string> (new string("Fowl Balls")),
 auto_ptr<string> (new string("Duck Walks")),
 auto_ptr<string> (new string("Chicken Runs")),
 auto_ptr<string> (new string("Turkey Errors")),
 auto_ptr<string> (new string("Goose Eggs"))
};
auto_ptr<string> pwin(films[2]);
int i;
cout << "The nominees for best avian baseball
 film are\n";
for (i = 0; i < 5; i++)
 cout << *films[i] << endl;
cout << "The winner is " << *pwin << "!\n";
```

Проблема состоит в том, что перенесение права владения объектом с указателя `films[2]` на `pwin` может привести к ситуации, когда `films[2]` уже больше не ссылается на строку. Иначе говоря, после того как объект класса `auto_ptr` передал право владения объектом, такой указатель больше не должен использоваться. Однако решение этого вопроса зависит уже от конкретной реализации.

## Стандартная библиотека шаблонов

Стандартная библиотека шаблонов, или STL, содержит набор шаблонов, представляющих *контейнеры*, *итераторы*, *функциональные объекты* и *алгоритмы*. Контейнер — это элемент данных, похожий на массив, который может содержать несколько значений. Контейнеры в STL являются однородными, т.е. они могут содержать значения только одинакового типа. Алгоритмы — это рецепты для выполнения определенных задач, таких, например, как сортировка массива или поиск определенного значения в списке. Итераторы представляют собой объекты, позволяющие передвигаться по объекту-контейнеру так же, как обычные указатели позволяют передвигаться по массиву; они являются обобщением указателей. Функциональные объекты — это объекты, действующие как функции; они могут быть объектами класса или указателями на функции (включающими в себя имя функции, поскольку именно оно действует как указатель). Библиотека STL позволяет построить множество контейнеров, включая массивы, очереди и списки, и выполнять различные операции с ними, в том числе поиск, сортировку и заполнение случайным образом.

Алекс Степанов (Alex Stepanov) и Мен Ли (Meng Lee) разработали библиотеку STL в Хьюлетт-Лэбораториз (Hewlett-Laboratories), выпустив первую версию в 1994 г. Комитет по стандартам ISO/ANSI C++ предложил включить ее как часть стандарта C++. Библиотека STL не является примером объектно-ориентированного программирования. Она представляет другую парадигму программирования, называемую *обобщенным программированием*. Это делает STL интересной в таком плане: какие функции она может выполнять и как она реализована. Существует слишком много информации об STL, чтобы представить ее в одной главе, поэтому рассмотрим лишь некоторые ключевые примеры и исследуем суть данного подхода. Мы начнем с нескольких специфических примеров. После знакомства с контейнерами, итераторами и алгоритмами рассмотрим философию, лежащую в основе проектирования с помощью библиотеки, а затем выполним полный обзор STL. В приложении G дано краткое описание различных методов и функций STL.

## Класс шаблонов `vector`

В компьютерном лексиконе термин *вектор* отвечает больше массиву, чем математическому вектору, который рассматривался в главе 10. Вектор содержит набор подобных значений, к которым возможен произвольный доступ. Это значит, что можно использовать индекс для непосредственного доступа, например, к десятому элементу массива без необходимости доступа к девяти предыдущим элементам. Класс `vector` обеспечивает набор операций, подобных операциям в классе `ArrayDb`, рассмотренном в главе 13. Таким образом, можно создать объект класса `vector`, присвоить один такой объект другому и использовать оператор `[]` для обеспечения доступа к элементам вектора. Чтобы сделать класс общим, нужно создать его шаблон. Именно это и делает STL, определяя шаблон класса `vector` в заголовочном файле `vector` (в ранних версиях `vector.h`).

Для создания объекта шаблона `vector` применяется обычная запись `<type>`, несущая информацию о том, какой тип будет использован. Кроме того, шаблон класса `vector` динамически распределяет память, и поэтому при инициализации можно использовать аргумент, определяющий, сколько элементов вектора нужно создать:

```
#include vector
using namespace std;
```

### Листинг 15.4 Программа vect1.cpp.

```
// vect1.cpp - использование шаблона vector
#include <iostream>
#include <string>
#include <vector>
using namespace std;

const int NUM = 5;
int main()
{
 vector<int> ratings(NUM);
 vector<string> titles(NUM);
 cout << "You will do exactly as told. You will enter\n"
 << NUM << " book titles and your ratings (0-10).\n";
 int i;
 for (i = 0; i < NUM; i++)
 {
 cout << "Enter title #" << i + 1 << ": ";
 getline(cin, titles[i]);
 cout << "Enter your rating (0-10): ";
 cin >> ratings[i];
 cin.get();
 }
 cout << "Thank you. You entered the following:\n"
 << "Rating\tBook\n";
 for (i = 0; i < NUM; i++)
 {
 cout << ratings[i] << "\t" << titles[i] << endl;
 }
 return 0;
}
```

```
// вектор из пяти значений типа int
vector<int> ratings(5);
```

```
int n;
cin >> n;
```

```
// вектор из n значений типа double
vector<double> scores(n);
```

После создания векторного объекта, перегрузка оператора `[]` позволяет использовать обычную форму записи массива для доступа к отдельным элементам:

```
ratings[0] = 9;
for (int i = 0; i < n; i++)
 cout << scores[i] << endl;
```

### СНОВА ОБ ОБЪЕКТАХ, ИСПОЛЬЗУЕМЫХ ДЛЯ ВЫДЕЛЕНИЯ ПАМЯТИ

Как и класс `string`, различные шаблоны контейнеров в STL имеют дополнительный аргумент, определяющий, какой именно объект для выделения памяти будет использоваться. Например, шаблон класса `vector` начинается так:

```
template <class T,
 class Allocator = allocator<T> >
class vector { ... }
```

Если аргумент в шаблоне пропущен, то контейнер по умолчанию будет использовать класс `allocator<T>`. Этот класс задает применение операторов `new` и `delete` стандартным образом.

В листинге 15.4 демонстрируется, как используется этот класс в несложном приложении. Данная програм-

ма создает два объекта класса **vector**, один со специализацией **int**, другой — со специализацией **string**; каждый из них содержит пять элементов.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Более ранние реализации используют заголовочный файл **vector.h** вместо **vector**. Хотя порядок включаемых файлов не должен иметь значения, g++ 2.7.1 требует, чтобы заголовочный файл **string** указывался до заголовочных файлов STL. Функция **getline()** в Microsoft Visual C++ 5.0 содержит ошибку, приводящую в беспорядок синхронизацию ввода и вывода, а функция **getline()** в Borland C++ Builder 1.0 запрашивает явный аргумент для символа разделятеля.

Результаты выполнения программы из листинга 15.4:

```
You will do exactly as told. You will enter
5 book titles and your ratings (0-10).
Enter title #1: The Cat Who Knew C++
Enter your rating (0-10): 6
Enter title #2: Felonious Felines
Enter your rating (0-10): 4
Enter title #3: Warlords of Wonk
Enter your rating (0-10): 3
Enter title #4: Don't Touch That Metaphor
Enter your rating (0-10): 5
Enter title #5: Panic Oriented Programming
Enter your rating (0-10): 8
Thank you. You entered the following:
Rating Book
6 The Cat Who Knew C++
4 Felonious Felines
3 Warlords of Wonk
5 Don't Touch That Metaphor
8 Panic Oriented Programming
```

Все, что делает программа, — просто использует шаблон **vector** как наиболее подходящий способ создания динамически выделяемого массива. Дальше рассмотрим пример, в котором применяется больше методов этого класса.

## Операции, допустимые при работе с шаблонами **vector**

Все контейнеры в STL обеспечивают определенные базовые методы, включая **size()**, который возвращает количество элементов в контейнере; **swap()**, который обменивает содержимое двух контейнеров; **begin()**, возвращающий итератор, ссылающийся на первый элемент контейнера; и **end()**, который возвращает итератор, ссылающийся на *элемент за пределами контейнера*.

Что же такое итератор? Это обобщенное понятие указателя. Фактически он может быть и просто указателем, или же объектом, похожим на указатель, для которого определены операции разыменования (**operator\*()**) и инкремента (**operator++()**). Как будет видно в дальнейшем, обобщение указателей в итераторы позволяет STL обеспечить унифицированный

интерфейс для различных классов контейнеров, включая и такие, для которых неприменимы обычные указатели. Каждый класс-контейнер определяет соответствующий итератор. Имя типа для такого итератора образовано из оператора **typedef** необходимого класса, объявленного как **iterator**. Например, чтобы объявить итератор для объекта **vector** специализации **double**, необходимо выполнить следующее:

```
vector<double>::iterator pd; // pd — это
 // итератор
```

Предположим, **scores** является объектом **vector<double>**:

```
vector<double> scores;
```

Тогда итератор **pd** можно использовать в следующих целях:

```
// указывать с помощью pd на первый элемент
pd = scores.begin();
// разыменовать итератор pd и присвоить
// значение первому элементу
*pd = 22.3;
// сделать pd указателем на следующий элемент
++pd;
```

Как видно из примера, итератор ведет себя как указатель.

Далее, что означает термин *элемент за пределами контейнера*? Суть этого термина заключается в том, что итератор ссылается на элемент, следующий за последним элементом контейнера. Эта идея похожа на идею о нулевом символе, который находится за последним реальным символом С-строки, с той лишь разницей, что нулевой символ является значением элемента, а итератор, указывающий на элемент за контейнером, является указателем (итератором). Функция-элемент **end()** идентифицирует конец контейнера. Если установить итератор на первый элемент контейнера, а затем увеличивать его значение, то рано или поздно он достигнет конца контейнера, т.е. все содержимое контейнера будет просмотрено. Таким образом, если объекты **scores** и **pd** определены выше, то вывести их содержимое можно с помощью следующего программного кода:

```
for (pd = scores.begin();
 pd != scores.end(); pd++)
 cout << *pd << endl;
```

Все контейнеры включают методы, подобные тем, которые мы только что рассмотрели. Кроме них, шаблон класса **vector** имеет и методы, доступные лишь для нескольких контейнеров в STL. Один из удобных методов, названный **push\_back()**, добавляет элемент в конец вектора. Выполняя это, данный метод обращается к управлению памятью, чтобы увеличить размер вектора для

помещения добавляемых элементов. Это значит, что можно применять следующий код:

```
vector<double> scores; //создать пустой вектор
double temp;
while (cin >> temp && temp >= 0)
 scores.push_back(temp);
cout << "You entered" << scores.size()
 << "scores.\n";
```

При каждой итерации цикла к вектору `scores` добавляется один элемент. При написании или запуске программы не нужно запоминать номер элемента. До тех пор пока в распоряжении программы будет достаточный объем памяти, она сможет расширять вектор `scores` при необходимости.

Метод `erase()` предназначен для устранения заданного диапазона вектора. Он имеет два аргумента-итератора, определяющих нужный диапазон. Важно четко понять, как именно STL определяет диапазоны, используя два итератора. Первый итератор ссылается на начало диапазона, а второй — на элемент, следующий за последним элементом диапазона. Например:

```
scores.erase(scores.begin(), scores.begin() + 2);
```

удаляет первый и второй элементы, т.е. те, на которые указывают `begin()` и `begin() + 1`. (Поскольку вектор обеспечивает произвольный доступ, такие операции, как `begin() + 2`, определены для итераторов класса `vector`.) Если `it1` и `it2` — два итератора, используется запись `[p1,p2]`, отражающая то, что диапазон начинается с `p1` и простирается дальше, но не включает `p2`. Таким образом, диапазон `[begin(), end()]` заключает в себе полное содержимое коллекции (рис. 15.3). Кроме того, диапазон `[p1,p1]` считается пустым. Запись `[]` не относится к синтаксису C++, поэтому ее нет в программном коде; она появляется только в документации.

### ПОМНИТЕ

Диапазон `[it1, it2]` задается двумя итераторами, `it1` и `it2`, и простирается от `it1` до `it2`, но не включая последний.

РИСУНОК 15.3

Концепция диапазона STL.

Метод `insert()` дополняет метод `erase()`. Он имеет три аргумента-итератора. Первый задает позицию, в которую будет вставлен новый элемент. Второй и третий определяют диапазон, который будет вставлен. Этот диапазон, как правило, является частью другого объекта. Например, в следующем коде

```
vector<int> old;
vector<int> new;
...
old.insert(old.begin(), new.begin() + 1,
 new.end());
```

все элементы (кроме первого) вектора `new` вставляются перед первым элементом вектора `old`. В связи с этим именно в такой ситуации удобно работать с итератором, ссылающимся на элемент за вектором, так как он позволяет легко добавить элементы в конец вектора:

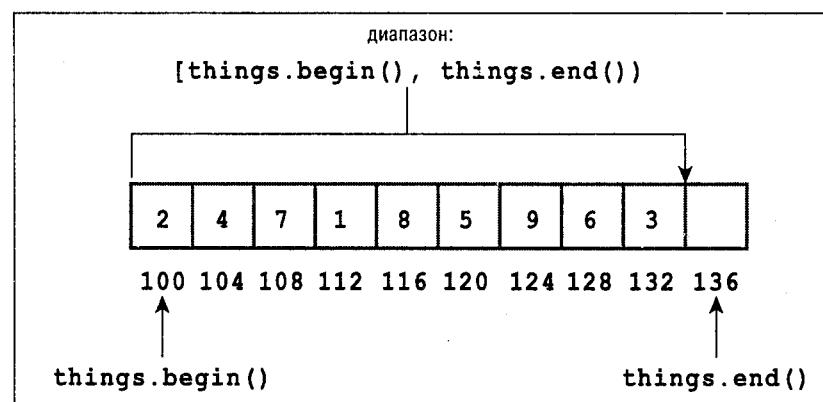
```
old.insert(old.end(), new.begin() + 1, new.end());
```

Здесь новые данные вставлены перед элементом `old.end()`, т.е. они помещены после последнего элемента вектора.

Листинг 15.5 иллюстрирует применение методов `size()`, `begin()`, `end()`, `push_back()`, `erase()` и `insert()`. Чтобы упростить работу с данными, компоненты `rating` и `title` в листинге 15.4 объединены в единую структуру `Review`, а функции `FillReview()` и `ShowReview()` обеспечивают ввод и вывод свойств для объектов `Review`.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Более ранние реализации компиляторов C++ используют заголовочный файл `vector.h` вместо `vector`. Хотя порядок включаемых файлов не должен иметь значения, g++ 2.7.1 требует, чтобы заголовочный файл `string` включался до заголовочных файлов STL. Функция `getline()` в Microsoft Visual C++ 5.0 содержит ошибку, приводящую к задержке строки при выводе до тех пор, пока не будут введены какие-то данные вновь. Кроме того, Microsoft Visual C++ 5.0 требует, чтобы операции `<` и `==` были определены для типа, сохраненного в объекте `vector`. Это значит, что необходимо добавить определения для операторов `operator<()` и `operator==()` для типа `Review`. Функция `getline()` в Borland C++ Builder 1.0 запрашивает явный аргумент для символа-разделителя.



**Листинг 15.5 Программа vect2.cpp.**

```
// vect2.cpp - методы и итераторы
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Review {
 string title;
 int rating;
};

bool FillReview(Review & rr);
void ShowReview(const Review & rr);

int main()
{
 vector<Review> books;
 Review temp;
 while (FillReview(temp))
 books.push_back(temp);
 cout << "Thank you. You entered the following:\n" << "Rating\tBook\n";
 int num = books.size();
 for (int i = 0; i < num; i++)
 ShowReview(books[i]);
 cout << "Reprising:\n" << "Rating\tBook\n";
 vector<Review>::iterator pr;
 for (pr = books.begin(); pr != books.end(); pr++)
 ShowReview(*pr);
 vector <Review> oldlist(books); // использование конструктора копирования
 if (num > 3)
 {
 // удалить два элемента
 books.erase(books.begin() + 1, books.begin() + 3);
 cout << "After erasure:\n";
 for (pr = books.begin(); pr != books.end(); pr++)
 ShowReview(*pr);

 // вставить один элемент
 books.insert(books.begin(), oldlist.begin() + 1, oldlist.begin() + 2);
 cout << "After insertion:\n";
 for (pr = books.begin(); pr != books.end(); pr++)
 ShowReview(*pr);
 }
 books.swap(oldlist);
 cout << "Swapping oldlist with books:\n";
 for (pr = books.begin(); pr != books.end(); pr++)
 ShowReview(*pr);
 return 0;
}

bool FillReview(Review & rr)
{
 cout << "Enter book title (quit to quit): ";
 getline(cin, rr.title);
 if (rr.title == "quit")
 return false;
 cout << "Enter book rating: ";
 cin >> rr.rating;
 if (!cin)
 return false;
 cin.get();
 return true;
}

void ShowReview(const Review & rr)
{
 cout << rr.rating << "\t" << rr.title << endl;
}
```

Результаты выполнения программы из листинга 15.5:

```

Enter book title (quit to quit): The Cat Who Knew Vectors
Enter book rating: 5
Enter book title (quit to quit): Candid Canines
Enter book rating: 7
Enter book title (quit to quit): Warriors of Wonk
Enter book rating: 4
Enter book title (quit to quit): Quantum Manners
Enter book rating: 8
Enter book title (quit to quit): quit
Thank you. You entered the following:
Rating Book
5 The Cat Who Knew Vectors
7 Candid Canines
4 Warriors of Wonk
8 Quantum Manners
Reprising:
Rating Book
5 The Cat Who Knew Vectors
7 Candid Canines
4 Warriors of Wonk
8 Quantum Manners
After erasure:
5 The Cat Who Knew Vectors
8 Quantum Manners
After insertion:
7 Candid Canines
5 The Cat Who Knew Vectors
8 Quantum Manners
Swapping oldlist with books:
5 The Cat Who Knew Vectors
7 Candid Canines
4 Warriors of Wonk
8 Quantum Manners

```

## Дополнительные операции с векторами

Обычно программисты выполняют множество действий с массивами — поиск в них, сортировку, заполнение случайным образом и т.д. Имеет ли класс шаблонов векторов методы для таких распространенных операций? Нет! В STL представлен более общий подход, определяющий для таких операций функции, не являющиеся элементами каких-либо классов. Поэтому вместо отдельной функции-элемента **find()** для каждого класса-контейнера определена одна для всех классов-контейнеров функция **find()**, не являющаяся элементом класса. Подобная философия проектирования позволяет избежать необходимости выполнять одну и ту же работу. Например, пусть имеется 8 классов-контейнеров и 10 операций для их поддержки. Если бы каждый класс имел свои собственные функции-элементы, то потребовалось бы  $8 \times 10$ , т.е. 80 отдельных определений функций-элементов. Но при подходе с использованием STL требуется лишь 10 отдельных определений функций, не являющихся элементами класса. Если определить новый класс-контейнер, следуя определенным требованиям, то для него также можно будет использовать существующие 10 функций для поиска, сортировки и т.д.

Давайте рассмотрим три типичные функции STL: **for\_each()**, **random\_shuffle()** и **sort()**. Функцию **for\_each()** можно использовать с любым классом-контейнером. Она имеет три аргумента. Первые два являются итераторами, задающими диапазон в контейнере, а последний представляет собой указатель на функцию. (В более общем смысле последний аргумент — это объект-функция; об этих объектах будет рассказываться далее.) Функция **for\_each()** выполняет действие, на которое указывает последний аргумент для каждого элемента контейнера в заданном диапазоне. Функция, на которую указывает третий аргумент, не должна изменять значения элементов контейнера. Функцию **for\_each()** можно использовать вместо цикла **for**. Например, следующий код можно заменить:

```

vector<Review>::iterator pr;
for (pr=books.begin(); pr != books.end(); pr++)
 ShowReview(*pr);

```

таким:

```
for_each(books.begin(), books.end(), ShowReview);
```

Такое применение функции позволит не "захламлять" код явно используемыми переменными-итераторами.

Функция **random\_shuffle()** включает два итератора, задающих диапазон; в этом диапазоне она переставляет элементы в случайном порядке. Например, в выражении

```
random_shuffle(books.begin(), books.end());
```

случайным образом переставляются все элементы в векторе **books**. В отличие от функции **for\_each**, работающей со всеми классами-контейнерами, данная функция требует, чтобы класс-контейнер имел возможность произвольного доступа к элементам, что как раз и позволяет делать класс **vector**.

Функция **sort()** также требует того, чтобы контейнер обеспечивал произвольный доступ. Она имеет две версии. Первая версия использует два итератора, задающих диапазон, и сортирует его, применяя оператор **<**, определенный для элементов того типа, который сохранен в контейнере. Например,

```

vector<int> coolstuff;
...
sort(coolstuff.begin(), coolstuff.end());

```

сортирует содержимое объекта **coolstuff** в возрастающем порядке, используя оператор **<** для сравнения значений.

Если элементы контейнера являются объектами, заданными пользователем, тогда требуется функция **operator<()**, определенная для работы с такими объектами (в целях использования функции **sort()**). Например, можно сортировать вектор, содержащий объекты **Review**, если определить функцию **operator<()**, являющуюся или нет элементом класса. Поскольку **Review** — это структура, ее элементы общедоступны,

поэтому для работы с ними вполне подойдет функция, не являющаяся элементом класса:

```
bool operator<(const Review & r1,
 const Review & r2)
{
 if (r1.title < r2.title)
 return true;
 else if (r1.title == r2.title &&
 r1.rating < r2.rating)
 return true;
 else
 return false;
}
```

Располагая подобной функцией, можно сортировать вектор, состоящий из объектов **Review** (таких, как книги):

```
sort(books.begin(), books.end());
```

Эта версия функции **operator<()** сортирует названия элементов в лексикографическом порядке. Если два объекта имеют одинаковые элементы названий, тогда они упорядочиваются по рейтингам. А теперь представьте, что необходима сортировка в убывающем порядке или по порядку рейтингов, а не названий. Тогда можно воспользоваться второй формой функции **sort()**. Она имеет три аргумента. Первые два аргумента являются итераторами, задающими диапазон. Последний, третий аргумент — это указатель на функцию (в более общем смысле — на функциональный объект), которую необходимо использовать вместо **operator<()** для сравнения. Возвращаемое значение должно быть конвертируемым в тип **bool**, причем значение **false** означает, что два аргумента расположены в неверном порядке. Вот пример подобной функции:

```
bool WorseThan(const Review & r1,
 const Review & r2)
{
 if (r1.rating < r2.rating)
 return true;
 else
 return false;
}
```

### Листинг 15.6 Программа vect3.cpp.

```
// vect3.cpp – использование функций STL

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct Review {
 string title;
 int rating;
};

bool operator<(const Review & r1, const Review & r2);
bool worseThan(const Review & r1, const Review & r2);
bool FillReview(Review & rr);
void ShowReview(const Review & rr);
```

Имея такую функцию, можно сортировать вектор **books**, состоящий из объектов **Review**, в порядке возрастания рейтингов с помощью следующего кода:

```
sort(books.begin(), books.end(), WorseThan);
```

Заметьте, что функция **WorseThan()** выполняет меньше работы, чем **operator<()**, при сравнении объектов **Review**. Если два объекта имеют одинаковые элементы-названия, функция **operator<()** сортирует объекты, используя элемент, задающий рейтинг. Но если два объекта имеют одинаковый элемент-рейтинг, функция **WorseThan()** считает их одинаковыми. Первый способ сортировки называется *полной сортировкой*, а второй — *сортировкой со строгим ограничением*. При полной сортировке, если результаты сравнения **a < b**, и **b < a** неверны, то **a** и **b** должны быть идентичны. При сортировке со строгим ограничением это не так. Значения могут быть идентичными, но у них может совпадать лишь один элемент, такой как элемент **rating** в примере с функцией **WorseThan()**. Поэтому, вместо того чтобы говорить, что два объекта идентичны, лучше при сортировке со строгим ограничением сказать, что два объекта эквивалентны. Листинг 15.6 иллюстрирует использование функций STL.

 **ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**  
Более ранние реализации используют заголовочный файл **vector.h** вместо **vector** и **algo.h** — вместо **algorithm**. Хотя порядок включаемых файлов не должен иметь значения, g++ 2.7.1 требует, чтобы заголовочный файл **string** появился до заголовочных файлов STL. Функция **getline()** в Microsoft Visual C++ 5.0 содержит ошибку, задерживающую строку при выводе до тех пор, пока что-либо не будет введено вновь. Кроме того, Microsoft Visual C++ 5.0 требует определить функцию **operator==()** в дополнение к функции **operator<()**. Функция **getline()** в Borland C++ Builder 1.0 запрашивает явный аргумент для символа-разделителя.

```
int main()
{
 vector<Review> books;
 Review temp;
 while (FillReview(temp))
 books.push_back(temp);

 cout << "Thank you. You entered the following "
 << books.size() << " ratings:\n"
 << "Rating\tBook\n";

 for_each(books.begin(), books.end(), ShowReview);

 sort(books.begin(), books.end());
 cout << "Sorted by title:\nRating\tBook\n";
 for_each(books.begin(), books.end(), ShowReview);

 sort(books.begin(), books.end(), worseThan);
 cout << "Sorted by rating:\nRating\tBook\n";
 for_each(books.begin(), books.end(), ShowReview);

 random_shuffle(books.begin(), books.end());
 cout << "After shuffling:\nRating\tBook\n";
 for_each(books.begin(), books.end(), ShowReview);

 return 0;
}

bool operator<(const Review & r1, const Review & r2)
{
 if (r1.title < r2.title)
 return true;
 else if (r1.title == r2.title && r1.rating < r2.rating)
 return true;
 else
 return false;
}

bool worseThan(const Review & r1, const Review & r2)
{
 if (r1.rating < r2.rating)
 return true;
 else
 return false;
}

bool FillReview(Review & rr)
{
 cout << "Enter book title (quit to quit): ";
 getline(cin, rr.title);
 if (rr.title == "quit")
 return false;
 cout << "Enter book rating: ";
 cin >> rr.rating;
 if (!cin)
 return false;
 cin.get();
 return true;
}

void ShowReview(const Review & rr)
{
 cout << rr.rating << "\t" << rr.title << endl;
}
```

Результаты выполнения программы из листинга 15.6:

```

Enter book title (quit to quit): The Cat Who Can Teach You Weight Loss
Enter book rating: 8
Enter book title (quit to quit): The Dogs of Dharma
Enter book rating: 6
Enter book title (quit to quit): The Wimps of Wonk
Enter book rating: 3
Enter book title (quit to quit): Farewell and Delete
Enter book rating: 7
Enter book title (quit to quit): quit
Thank you. You entered the following 4 ratings:
Rating Book
8 The Cat Who Can Teach You Weight Loss
6 The Dogs of Dharma
3 The Wimps of Wonk
7 Farewell and Delete
Sorted by title:
Rating Book
7 Farewell and Delete
8 The Cat Who Can Teach You Weight Loss
6 The Dogs of Dharma
3 The Wimps of Wonk
Sorted by rating:
Rating Book
3 The Wimps of Wonk
6 The Dogs of Dharma
7 Farewell and Delete
8 The Cat Who Can Teach You Weight Loss
After shuffling:
Rating Book
7 Farewell and Delete
3 The Wimps of Wonk
6 The Dogs of Dharma
8 The Cat Who Can Teach You Weight Loss

```

## Обобщенное программирование

Теперь, когда у вас появился некоторый опыт в использовании STL, давайте ознакомимся с лежащей в ее основе философией. STL представляет собой пример *обобщенного программирования*. Объектно-ориентированное программирование концентрирует внимание на таком аспекте, как данные, а обобщенное программирование — на алгоритмах. Два элемента, общие в этих подходах, — это абстракция и создание повторно используемого кода, но у них разная философия.

Целью обобщенного программирования является написание программного кода, независимого от типов данных. Шаблоны в C++ представляют собой инструменты общего программирования. Шаблоны конечно же позволяют определить функцию или класс в терминах общего типа. Библиотека STL продвигается дальше, обеспечивая обобщенное представление алгоритмов. Шаблоны делают возможным и это при условии аккуратного и осознанного процесса проектирования. Чтобы понять, как работает эта смесь шаблонов и выполняется проектирование, давайте выясним, для чего нужны итераторы.

## Почему именно итераторы?

Понимание работы итераторов, возможно, является ключом к пониманию функционирования STL. Подобно тому как шаблоны делают алгоритмы независимыми от сохраняемых типов данных, итераторы делают алгоритмы независимыми от используемых контейнеров. Таким образом, они являются существенным компонентом обобщенного подхода STL.

Чтобы увидеть, зачем нужны итераторы, давайте посмотрим, как можно реализовать функцию поиска для двух различных представлений данных и как затем можно обобщить этот подход. Сначала представим функцию, которая осуществляет поиск определенного значения в обычном массиве типа **double**. Подобную функцию можно написать так:

```

double * find_ar(double * ar, int n,
 const double & val)
{
 for (int i = 0; i < n; i++)
 if (ar[i] == val)
 return &ar[i];
 return 0;
}

```

Если функция находит значение в массиве, она возвращает адрес в массиве, содержащем найденное значение; в ином случае она возвращает нулевой указатель. Функция использует форму записи с нижними индексами для перемещения по массиву. Можно использовать шаблон для обобщения функции на массивы любого типа, включающие операцию `==`. Тем не менее, этот алгоритм все равно связан с одной определенной структурой данных — массивом.

Теперь рассмотрим поиск в другом типе структур данных — связанным списке. (В главе 11 связанный список применялся для реализации класса `Queue`.) Список состоит из связанных структур `Node`:

```
struct Node
{
 double item;
 Node * p_next;
};
```

Представьте, что у вас есть указатель на первый узел в списке. Указатель `p_next` в каждом узле указывает на следующий узел, а указатель `p_next` последнего узла установлен в 0. Функция для поиска в таком списке, `find_ll()`, может быть реализована следующим образом:

```
Node* find_ll(Node * head, const double & val)
{
 Node * start;
 for (start = head; start != 0;
 start = start->next)
 if (start->item == val)
 return start;
 return 0;
}
```

Вновь можно использовать шаблон для обобщения таких списков на любые типы данных, поддерживающие выполнение операции `==`. Тем не менее, данный алгоритм все равно связан с одной определенной структурой данных — связанным списком.

При рассмотрении деталей реализации ясно, что две функции поиска используют различные алгоритмы: в одной выполняется индексирование массива для перемещения по списку элементов, в другой значение переменной `start` заменяется на `start->next`. Но в широком смысле оба алгоритма одинаковы: они последовательно сравнивают значение с каждым значением в контейнере до тех пор, пока не будет найдено совпадение.

Цель обобщенного программирования — создать одну функцию поиска, которая работала бы с массивами, связанными списками или любыми другими типами контейнеров. Иначе говоря, функция должна быть независимой не только от типа данных, сохраняемых в контейнере, но и от самой структуры контейнера. Шаблоны обеспечивают общее представление типов данных, сохраняемых в контейнере. Все, что еще нужно, — это

общее представление процесса перемещения от значения к значению в контейнере. Итератор реализует именно такое обобщенное представление.

Какими же возможностями должен обладать итератор для реализации функции поиска? Необходимо, чтобы он позволял выполнять следующие функции:

- Разыменование итератора, чтобы получить доступ к значению, на которое он ссылается. Другими словами, если `p` — итератор, то необходимо определить объект `*p`.
- Присвоение одного итератора другому. Если `p` и `q` — итераторы, то должно быть определено выражение `p = q`.
- Сравнение итераторов на предмет равенства. Если `p` и `q` — итераторы, должны быть определены выражения `p == q` и `p != q`.
- Перемещение его по всем элементам контейнера. Это условие можно удовлетворить, определив операции `++p` и `p++` для итератора `p`.

Существует еще множество действий, которые позволяет выполнить итератор, но для функции поиска больше ничего не требуется. На самом деле, в библиотеке STL определены несколько уровней итераторов с нарастающими возможностями, и к этому вопросу мы вернемся позднее. Заметьте, кстати, что обычный указатель удовлетворяет требованиям, предъявляемым к итератору. Следовательно, функцию `find_arr()` можно переписать следующим образом:

```
typedef double * iterator;
iterator find_ar(iterator ar, int n,
 const double & val)
{
 for (int i = 0; i < n; i++, ar++)
 if (*ar == val)
 return ar;
 return 0;
}
```

Для функции `find_ll()` можно определить класс итераторов, в котором заданы операторы `*` и `++`:

```
struct Node
{
 double item;
 Node * p_next;
};

class iterator
{
 Node * pt;
public:
 iterator() : pt(0) { }
 iterator (Node * pn) : pt(pn) { }
 double operator*()
 {
 return pt->item;
 }
 iterator& operator++() // для ++it
```

```

{
 pt = pt->next;
 return *this;
}
iterator operator++(int) // для it++
{
 iterator tmp = *this;
 pt = pt->next;
 return tmp;
}
// ... operator==(), operator!=() и т.д.
};

```

(Чтобы различать префиксную и постфиксную версии оператора `++`, язык C++ использует соглашение, позволяющее функции `operator++()` быть префиксной версией, функции `operator++(int)` — постфиксной версией; аргумент не используется, следовательно, его имя задавать не нужно.)

Главная мысль заключается не в том, как в подробностях определить класс итераторов, а в том, что, имея такой класс, второй вариант функции поиска можно задать следующим образом:

```

iterator find_ll(iterator head,
 const double & val)
{
 iterator start;
 for (start = head; start != 0; ++start)
 if (*start == val)
 return start;
 return 0;
}

```

Это очень близко к реализации функции `find_ar()`. Различие состоит в том, как две функции определяют, что достигнута крайняя точка в диапазоне поиска значений. Функция `find_ar()` использует счетчик элементов, а функция `find_ll()` — нулевое значение, сохраненное в последнем узле. Устранив это различие, обе функции можно сделать идентичными. Например, можно потребовать, чтобы массив и связанный список имели один дополнительный элемент после "фактического" последнего элемента. Тогда поиск будет прекращаться при достижении позиции, находящейся за контейнером. В этом случае функции `find_ar()` и `find_ll()` будут одинаковым образом определять окончание ввода данных и алгоритмы станут идентичными. Заметьте, что требование наличия элемента, размещенного за контейнером, перемещает акцент с требований для итераторов на требования для класса-контейнера.

Библиотека STL следует только что описанному подходу. Сначала каждый класс-контейнер (`vector`, `list`, `deque` и т.д.) определяет тип итератора, подходящий для данного класса. Для одного класса итератор может быть указателем, для другого — объектом. Независимо от реализации итератор должен обеспечивать выполнение необходимых операций `*` и `++`. (Одним классам требу-

ется больше операций, чем другим.) Далее каждый класс-контейнер задает маркер элемента, размещенного за контейнером, который представляет собой значение, присваиваемое итератору, когда он применяется к элементу контейнера, находящемуся за последним элементом. Методы `begin()` и `end()` устанавливают итераторы на первый элемент и элемент, находящийся за контейнером. Операция `++` позволяет итератору просмотреть все элементы — от первого до последнего.

Чтобы использовать класс-контейнер, не нужно знать, как реализованы его итераторы или элемент, находящийся за пределами контейнера. Достаточно лишь знать, что класс включает итераторы, что метод `begin()` возвращает итератор на первый элемент и что метод `end()` возвращает итератор на элемент, находящийся за границами контейнера. Например, представьте, что нужно отобразить значения объекта `vector<double>`. Тогда можно сделать следующее:

```

vector<double>::iterator pr;
for (pr = scores.begin();
 pr != scores.end(); pr++)
 cout << *pr << endl;

```

Здесь строка

```
vector<double>::iterator pr;
```

идентифицирует `pr` как итератор типа, определенного для класса `vector<double>`. Если вместо этого класса использовался шаблон класса `list<double>`, то можно применить следующий программный код:

```

list<double>::iterator pr;
for (pr = scores.begin();
 pr != scores.end(); pr++)
 cout << *pr << endl;

```

Единственное изменение — тип, объявленный для `pr`. Таким образом, при наличии итераторов, определенных соответствующим образом для каждого класса, и классов, разработанных с учетом универсального подхода, STL позволяет написать одинаковый код для контейнеров, имеющих абсолютно разное внутреннее представление.

Фактически это касается стиля, но все же лучше избегать прямого использования итераторов; вместо этого по возможности лучше применять функции STL, такие как `for_each()`, которые сами позаботятся о деталях.

Давайте подытожим, что дает подход STL. Он позволяет начать с алгоритма для работы с контейнером. Затем алгоритм выражается в наиболее общих терминах, независимо от типа данных и типа контейнера. Чтобы заставить общий алгоритм выполнятьсь в специфических случаях, нужно определить соответствующие итераторы и наложить

требования на структуру контейнера. Таким образом, базовые свойства итератора и контейнера проистекают из требований, наложенных на них алгоритмом.

## Типы итераторов

Различные алгоритмы накладывают разные требования на итераторы. Например, алгоритм поиска требует, чтобы был определен оператор `++` и итератор мог пройти весь контейнер. Кроме того, итератору требуется доступ для чтения, но не требуется доступ для записи. (Он просто просматривает данные, не изменяя их.) Однако алгоритм сортировки требует произвольного доступа, чтобы иметь возможность выполнения операции обмена между двумя несоседними элементами. Если `iter` — это итератор, произвольный доступ можно получить, определив операцию `+`, чтобы можно было использовать выражения типа `iter + 10`. Алгоритм сортировки также должен иметь возможность считывать и записывать данные.

В библиотеке STL определены пять типов итераторов и описаны алгоритмы, для реализации которых они требуются. К этим пяти типам относятся: *итератор ввода и вывода, прямой итератор, двусторонний итератор и итератор произвольного доступа*. Например, прототип функции `find()` выглядит так:

```
template<class InputIterator, class T>
InputIterator find(InputIterator first,
 InputIterator last,
 const T& value);
```

Этот прототип говорит о том, что данному алгоритму требуется итератор ввода. Аналогично прототип

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first,
 RandomAccessIterator last);
```

говорит о том, что алгоритму сортировки требуется итератор произвольного доступа.

Все пять типов итераторов можно разыменовать (т.е. для них определена операция `*`) и все пять типов можно сравнивать на предмет установки соответствия (используя оператор `==`, возможно, перегруженный) и неравенство (используя оператор `!=`, возможно, перегруженный). Если два итератора считаются равными, то разыменование одного из них должно давать такой же эффект, как и разыменование другого. Иначе говоря, если равенство

```
iter1 == iter2
```

справедливо, то и следующее равенство также справедливо:

```
*iter1 == *iter2
```

Конечно, эти свойства справедливы для встроенных итераторов и указателей, поэтому данные требования являются руководящими при выполнении перегрузки операторов для класса итераторов. Теперь рассмотрим другие свойства итераторов.

## Итератор ввода

Термин "ввод" используется с точки зрения программы, т.е. информация, поступающая из контейнера в программу, считается вводом, так же как и информация, поступающая, например, с клавиатуры. Значит, итератор ввода используется программой для чтения значений из контейнера. В частности, запрашивание объекта по итератору ввода должно позволить программе прочесть значение из контейнера, но не изменить его. Таким образом, алгоритмы, требующие использования итераторов ввода, являются алгоритмами, не изменяющими значений, сохраняемых в контейнерах.

Итератор ввода позволяет получить доступ ко всем значениям в контейнере. Он выполняет это, поддерживая операцию `++` и в префиксной, и в постфиксной формах. Если итератор ввода установлен на первый элемент контейнера, а затем его значение увеличивается до тех пор, пока не будет достигнут элемент, находящийся за пределами контейнера, то итератор будет указывать на каждый элемент контейнера только один раз. В связи с этим нет никаких гарантий, что при обходе контейнера второй раз итератор ввода пройдет по значениям в таком же порядке, как раньше. К тому же, если значение итератора ввода увеличено на единицу, нет никаких гарантий, что его можно разыменовать по предыдущему значению. Следовательно, всякий алгоритм, базирующийся на итераторе ввода, должен быть однопроходным, т.е. не требующим значений итератора, определенных при прошлом прохождении контейнера, или предыдущих его значений при текущем прохождении.

Заметьте, что итератор ввода односторонний; его значение можно увеличить, но нельзя вернуться назад.

## Итератор вывода

Здесь термин "вывод" показывает, что итератор используется для передачи информации из программы контейнеру. Итератор вывода похож на итератор ввода с той разницей, что разыменование итератора с гарантией позволит программе изменить значение в контейнере, но не позволит прочесть его. Если способность осуществлять запись без возможности чтения кажется странной, вспомните, что таким же свойством обладает поток вывода, посыпаемый на дисплей; оператор `cout` может изменить поток символов, посыпаемых на дисплей, но он не может прочесть того, что находится на экране.

Библиотека STL является достаточно общей, чтобы ее контейнеры могли представлять собой устройства вывода, поэтому такая же ситуация справедлива и для контейнеров. Кроме того, если алгоритм изменяет значения в контейнере (например, генерируя новые значения для сохранения) без чтения его содержимого, нет причин требовать, чтобы он использовал итератор, который может читать содержимое контейнера.

Итак, итератор ввода можно применять для однопроходных алгоритмов, используемых только для чтения, а итератор вывода — для однопроходных алгоритмов, используемых только для записи.

### Прямой итератор

Как итераторы ввода и вывода, прямой итератор использует только операции `++` для навигации по контейнеру. Поэтому он может осуществлять только перемещение вперед по контейнеру на один элемент при каждом приращении. Однако, в отличие от итераторов ввода и вывода, он всегда проходит последовательность значений в одном и том же порядке. К тому же после приращения прямого итератора, его все равно можно разыменовать по его предыдущему значению и получить тот же элемент, что и раньше. Эти свойства делают возможным применение многопроходных алгоритмов.

Прямой итератор может позволить реализовать два типа доступа: либо читать и изменять данные, либо только читать их:

```
// итератор, осуществляющий операции
// чтения-записи
int * pirw;

// итератор, осуществляющий операции чтения
const int * pir;
```

### Двусторонний итератор

Представьте себе, что вам нужен алгоритм, способный обходить контейнер в обоих направлениях. Например, функция реверсирования контейнера может выполнить обмен между первым и последним элементами, увеличить указатель на первый элемент, уменьшить указатель на второй элемент и повторять этот процесс. Двусторонний итератор имеет все свойства прямого итератора, к которым добавлена возможность поддержки двух операторов декремента (префиксного и постфиксного).

### Итератор произвольного доступа

Некоторые алгоритмы, такие, например, как сортировка и бинарный поиск, требуют возможности перехода непосредственно к произвольному элементу контейнера. Это действие реализуется с помощью итератора произвольного доступа. Он имеет все свойства двустороннего итератора плюс операции (наподобие сложения указателей), поддерживающие произвольный доступ и операторы сравнения, используемые для упорядочения элементов. В табл. 15.3 приведены операции, которые может выполнять итератор произвольного доступа и не может выполнять двусторонний итератор. В этой таблице символ `X` представляет итератор произвольного доступа, `T` — тип, на который он указывает, `a` и `b` — значения итератора, `n` — значение целого типа и `r` — переменную итератора произвольного доступа или ссылку.

Выражения типа `a + n` справедливы только в том случае, если и `a`, и `a + n` попадают в диапазон контейнера (включая элемент, находящийся за пределами контейнера).

Таблица 15.3 Операции итератора произвольного доступа.

| Выражение              | Комментарии                                                                                     |
|------------------------|-------------------------------------------------------------------------------------------------|
| <code>a + n</code>     | Указывает на <code>n</code> -й элемент после элемента, на который указывает <code>a</code>      |
| <code>n + a</code>     | То же самое, что и <code>a + n</code>                                                           |
| <code>a - n</code>     | Указывает на <code>n</code> -й элемент перед тем элементом, на который указывает <code>a</code> |
| <code>r += n</code>    | Эквивалентно <code>r = r + n</code>                                                             |
| <code>r -= n</code>    | Эквивалентно <code>r = r - n</code>                                                             |
| <code>a[n]</code>      | Эквивалентно <code>*(a + n)</code>                                                              |
| <code>b - a</code>     | Значение <code>n</code> таково, что <code>b = a + n</code>                                      |
| <code>a &lt; b</code>  | Справедливо, если <code>b - a &gt; 0</code>                                                     |
| <code>a &gt; b</code>  | Справедливо, если <code>b &lt; a</code>                                                         |
| <code>a &gt;= b</code> | Справедливо, если <code>!(a &lt; b)</code>                                                      |
| <code>a &lt;= b</code> | Справедливо, если <code>!(a &gt; b)</code>                                                      |

## Иерархия итераторов

Наверное, вы уже заметили, что типы итераторов формируют иерархию. Прямой итератор включает все возможности итераторов ввода и вывода, плюс дополнительные возможности. Двусторонний итератор располагает всеми возможностями прямого итератора, а также включает свои собственные возможности. И, наконец, итератор произвольного доступа имеет все возможности прямого итератора плюс свои собственные возможности. В табл. 15.4 показаны основные возможности итераторов. Здесь символ *i* обозначает итератор, а *n* — значение целочисленного типа данных.

Алгоритм, написанный с применением определенного типа итератора, может использовать или этот тип итератора, или любой другой, имеющий требуемые возможности. Поэтому, например, контейнер с итератором произвольного доступа может использовать алгоритм, написанный для итератора ввода.

Зачем же нужны различные типы итераторов? Идея состоит в том, чтобы создать алгоритм, использующий итератор с наименьшим возможным числом требований, позволяя, таким образом, применять его с большим числом различных контейнеров. Поэтому функция `find()`, в которой применен низкоуровневый итератор ввода, может быть использована с любым контейнером, содержащим доступные для чтения значения. Однако функция `sort()`, требующая наличия итератора произвольного доступа, может применяться только для контейнеров, которые поддерживают такой тип итератора.

Заметьте, что различные типы итераторов не являются определенными типами; скорее, они представляют собой набор концептуальных характеристик. Как упоминалось ранее, каждый класс-контейнер определяет свой итератор как имя класса `typedef` с последующей декларацией `iterator`. Таким образом, класс `vector<int>` включает итераторы типа `vector<int>::iterator`. В документации об этом классе сказано, что его итераторы являются итераторами произвольного доступа. Это, в свою очередь, позволяет использовать алгоритмы, базирующиеся на любом типе итератора, поскольку итератор произвольного доступа располагает всеми возможностями итератора. Подобным образом класс `list<int>` имеет итераторы типа `list<int>::iterator`. В STL реализован двусвязный список, поэтому в нем используется двусторонний итератор. В связи с этим класс `list<int>` не может использовать алгоритмы, построенные на основе итераторов произвольного доступа, но вполне может применять алгоритмы с менее требовательными итераторами.

## Концепции, уточнения и модели

Библиотека STL включает несколько свойств, такие, например, как типы итераторов, которые нельзя выразить на языке C++. Иначе говоря, хотя можно создать класс, имеющий свойства прямого итератора, нельзя заставить компилятор ограничить алгоритм использованием только этого класса. Причина состоит в том, что прямой итератор — это набор требований, а не тип.

Таблица 15.4 Возможности итераторов.

| Возможность итератора               | Итератор ввода | Итератор вывода | Прямой итератор | Двусторонний итератор | Итератор произвольного доступа |
|-------------------------------------|----------------|-----------------|-----------------|-----------------------|--------------------------------|
| Разыменование для чтения            | да             | нет             | да              | да                    | да                             |
| Разыменование для записи            | нет            | да              | да              | да                    | да                             |
| Фиксированный и повторяемый порядок | нет            | нет             | да              | да                    | да                             |
| <code>++i</code>                    |                |                 |                 |                       |                                |
| <code>i++</code>                    | да             | да              | да              | да                    | да                             |
| <code>-i</code>                     |                |                 |                 |                       |                                |
| <code>i-</code>                     | нет            | нет             | нет             | да                    | да                             |
| <code>i[n]</code>                   | нет            | нет             | нет             | нет                   | да                             |
| <code>i + n</code>                  | нет            | нет             | нет             | нет                   | да                             |
| <code>i - n</code>                  | нет            | нет             | нет             | нет                   | да                             |
| <code>i += n</code>                 | нет            | нет             | нет             | нет                   | да                             |
| <code>i -= n</code>                 | нет            | нет             | нет             | нет                   | да                             |

Данным требованиям может удовлетворять не только созданный класс-итератор, но и обычный указатель. Алгоритм STL работает с любой реализацией итератора, которая удовлетворяет необходимым требованиям. В литературе, посвященной STL, используется слово **концепция** для описания набора требований. Таким образом, существует концепция итератора ввода, концепция прямого итератора и т.д. Кстати, если итераторы необходимы для создаваемого класса-контейнера, то библиотека STL включает в себя шаблоны итераторов стандартных разновидностей.

Концепции могут быть связаны отношением, похожим на наследование. Например, двусторонний итератор наследует все возможности прямого итератора. Однако механизм наследования C++ невозможно применить к итераторам. Например, затруднительно реализовать прямой итератор как класс, а двусторонний итератор как обычный указатель. Таким образом, согласно терминологии языка C++, данный двусторонний итератор будет встроенным типом, поэтому его нельзя получить путем наследования из класса. Однако в концептуальном смысле итератор все же проявляет отношение наследования. В некоторой литературе по STL используется термин **уточнение** для определения такого концептуального наследования. Другими словами, двусторонний итератор является уточнением концепции прямого итератора.

Определенная реализация концепции называется **моделью**. Таким образом, обычный указатель на тип **int** представляет собой модель концепции прямого итератора. Это также модель и прямого итератора, поскольку он удовлетворяет всем требованиям этой концепции.

### Указатель как итератор

Итераторы являются обобщением указателей, кроме того, указатели удовлетворяют всем требованиям итераторов. Итераторы образуют интерфейс алгоритмов STL, а указатели являются итераторами, поэтому алгоритмы STL могут использовать указатели для операции с внешними, не входящими в STL контейнерами. Например, можно применять алгоритмы STL для массивов. Пусть **Receipts** — это массив значений типа **double**, и нужно выполнить его сортировку в возрастающем порядке:

```
const int SIZE = 100;
double Receipts[SIZE];
```

Вспомните, что функция **sort()** в STL получает в качестве аргументов итератор, указывающий на первый элемент контейнера, и итератор указывающий на элемент, находящийся за пределами контейнера. Здесь **&Receipts[0]** (или просто **Receipts**) — это адрес первого элемента, а **&Receipts[SIZE]** (или просто **Receipts + SIZE**) — это адрес элемента, следующего за

последним элементом массива. Поэтому при вызове функции

```
sort(Receipts, Receipts + SIZE);
```

выполняется сортировка массива. Язык C++ гарантирует, что выражение **Receipts + n** определено, если результат находится в диапазоне элементов массива, включая и элемент, находящийся за пределами массива.

Таким образом, те факты, что указатели являются итераторами и что алгоритмы базируются на итераторах, делают возможным применение алгоритмов STL для обычных массивов. Подобным образом можно воспользоваться алгоритмами STL для форм данных, проектированных пользователем, при условии поддержки подходящих итераторов (которые могут быть указателями или объектами) и индикаторов элемента, находящегося за пределами массива.

### Итераторы **copy()**, **ostream\_iterator** и **istream\_iterator**

Библиотека STL имеет несколько предопределенных итераторов. Чтобы понять, зачем нужны подобные итераторы, необходимо ознакомиться с некоторыми основами. В частности, имеется алгоритм (**copy()**), предназначенный для копирования данных из одного контейнера в другой. Этот алгоритм написан с учетом терминологии итераторов, поэтому он может копировать данные из контейнера одного типа в контейнер другого типа, а также в массив или из него, поскольку можно использовать указатели на массивы в качестве итераторов. Например, сутью следующего выражения является копирование массива в вектор:

```
int casts[10] = { 6, 7, 2, 9, 4, 11, 8,
 7, 10, 5 };
vector<int> dice[10];
// скопировать массив в вектор
copy(casts, casts + 10, dice.begin());
```

Первые два аргумента-итератора в функции **copy()** задают диапазон значений, которые нужно скопировать, а последний аргумент-итератор указывает, куда нужно скопировать первый элемент из диапазона. Первые два аргумента должны быть итераторами ввода (как минимум), а последний аргумент — итератором вывода (как минимум). Функция **copy()** перезаписывает существующие данные в контейнере назначения, причем этот контейнер должен быть достаточно большим, чтобы вместить копируемые элементы. Поэтому функцию **copy()** нельзя использовать для размещения данных в пустом векторе, по крайней мере, не применив хитрость, о которой будет рассказано далее.

Теперь представим, что необходимо вывести информацию на дисплей. Можно использовать функцию

`copy()` при условии, что имеется итератор, представляющий выходной поток. STL определяет такой итератор с помощью шаблона **ostream\_iterator**. Используя терминологию STL, этот шаблон можно представить как модель концепции итератора вывода. Это также и пример *адаптера*, класса или функции, которая преобразует некоторый внешний интерфейс в интерфейс, используемый в STL. Итератор такого типа можно создать, включив заголовочный файл **iterator** (в более ранних версиях **iterator.h**) и объявив декларацию:

```
#include <iterator>
...
ostream_iterator<int, char> out_iter(cout, " ");
```

Итератор **out\_iter** становится интерфейсом, позволяющим использовать конструкцию **cout** для вывода информации. Первый аргумент шаблона (в данном случае типа **int**) показывает, какой тип данных посыпается в поток вывода. Второй аргумент шаблона (в данном случае типа **char**) показывает тип символов, используемый потоком вывода. (Другое возможное значение — тип **wchar\_t**.) Первый аргумент конструктора (в данном случае **cout**) определяет, какой именно поток вывода будет использован. Это также может быть и поток, используемый для файлового вывода, который рассматривается в главе 16. Последний строковый аргумент — это разделитель, который будет выводиться после каждого элемента, посыпанного в поток вывода.

#### ПРЕДОСТЕРЕЖЕНИЕ

Более ранние реализации используют только первый аргумент шаблона для **ostream\_iterator**:

```
ostream_iterator<int> out_iter(cout, " ");
// более ранняя реализация
```

Итератор можно использовать, например, так:

```
*out_iter++ = 15; // работает так же, как
// и cout << 15 << " ";
```

Для обычного указателя это означает присвоение значения 15 объекту, на который указывает указатель, и затем приращение указателя на единицу. Однако для **ostream\_iterator** данное выражение обозначает: направить 15 в строку, состоящую из пробелов, в поток вывода, который управляемся оператором **cout**. Затем необходимо подготовиться к следующей операции вывода. С функцией **copy()** итератор можно использовать следующим образом:

```
// копировать вектор в поток вывода
copy(dice.begin(), dice.end(), out_iter);
```

Это означает: скопировать весь диапазон контейнера **dice** в поток вывода, т.е. просто вывести содержимое контейнера на экран.

Или же можно пропустить создание итератора с именем и построить вместо этого анонимный итератор, т.е. можно применить адаптер так:

```
copy(dice.begin(), dice.end(),
ostream_iterator<int, char>(cout, " "));
```

Подобным образом заголовочный файл **iterator** определяет шаблон **istream\_iterator** для адаптации потока ввода к интерфейсу итераторов. Это модель итератора ввода. Можно использовать два объекта **istream\_iterator**, чтобы определить диапазон ввода для функции **copy()**:

```
copy(istream_iterator<int, char>(cin),
istream_iterator<int, char>(),
dice.begin());
```

Как и **ostream\_iterator**, итератор **istream\_iterator** использует два аргумента шаблона. Первый задает тип данных, предназначенных для чтения, а второй — тип символа, используемый в потоке ввода. Аргумент **cin** конструктора обозначает, что будет использоваться входной поток, управляемый конструкцией **cin**. Пропуск аргумента конструктора указывает на ошибку ввода. Таким образом, код, приведенный выше обозначает, что поток ввода необходимо читать до тех пор, пока не встретится символ конца файла, несоответствие типов или другая ошибка ввода.

#### Другие полезные итераторы

Заголовочный файл **iterator** имеет несколько специально предопределенных типов итераторов в дополнение к **ostream\_iterator** и **istream\_iterator**. Это итераторы **reverse\_iterator**, **back\_insert\_iterator**, **front\_insert\_iterator** и **insert\_iterator**.

Начнем с рассмотрения того, какие функции выполняет обратный итератор (**reverse\_iterator**). По существу, приращение обратного итератора заставляет его уменьшаться. Почему же вместо этого просто не уменьшить обычный итератор? Основная причина заключается в том, чтобы упростить существующие функции. Представьте, что вам необходимо отобразить содержимое контейнера **dice**. Как вы только что видели, можно воспользоваться функцией **copy()** и итератором **ostream\_iterator** для копирования содержимого контейнера в поток вывода

```
ostream_iterator<int, char> out_iter(cout, " ");
// вывести в прямом порядке
copy(dice.begin(), dice.end(), out_iter);
```

Теперь представьте, что вам необходимо вывести содержимое контейнера в обратном порядке. (Возможно, вы изучаете принципы обращения времени.) Существует несколько подходов, которые не сработывают, но, вместо того чтобы разбирать их, мы сразу перейдем к тому, который дает результаты. Класс **vector** имеет

функцию-элемент `rbegin()`, которая возвращает обратный итератор, указывающий на элемент за контейнером, и функцию-элемент `rend()`, возвращающую обратный итератор, указывающий на первый элемент. Поскольку приращение обратного итератора заставляет его уменьшаться, можно воспользоваться выражением:

```
// вывести в обратном порядке
copy(dice.rbegin(), dice.rend(), out_iter);
```

чтобы вывести содержимое контейнера в обратном порядке. Фактически обратный итератор даже не нужно объявлять.

### ПОМНИТЕ

Обе функции, `rbegin()` и `end()`, возвращают одно и то же значение (итератор, указывающий на элемент за пределами контейнера), но имеющие разный тип (`reverse_iterator` и `iterator`). Аналогично обе функции, `rend()` и `begin()`, возвращают одинаковые значения (итератор на первый элемент), но разного типа.

Обратный указатель должен выполнять некоторый вид специальной компенсации. Пусть `gr` — это обратный указатель, инициализированный как `dice.rbegin()`. Чем же должен быть `*gr`? Поскольку `rbegin()` возвращает указатель на элемент, находящийся за пределами контейнера, его нельзя разыменовать по этому адресу. Подобным образом, если `rend()` — это местоположение

первого элемента, функция `copy()` остановится на один элемент раньше, поскольку конечный элемент диапазона не входит в диапазон. Обратные указатели решают эту проблему таким образом, что сначала происходит их уменьшение на единицу, а затем разыменование, т.е. `*gr` разыменовывает итератор предшествующим значением. Если `gr` указывает на шестую позицию, то `*gr` дает значение пятой позиции и т.д. В листинге 15.7 иллюстрируется использование функции `copy()`, итератора `istream` и обратного итератора.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В более ранних реализациях используются заголовочные файлы `iterator.h` и `vector.h`. Кроме того, может использоваться определение итератора `ostream_iterator<int>` вместо `ostream_iterator<int, char>`.

Результаты выполнения программы из листинга 15.7:

```
Let the dice be cast!
6 7 2 9 4 11 8 7 10 5
Implicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
Explicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
```

Если есть выбор между явным объявлением итераторов или внутренним использованием функций STL, например при передаче значения, возвращаемого функцией `rbegin()` другой функции, лучше использовать

### Листинг 15.7 Программа copy.cpp.

```
// copy.cpp — функция copy() и итераторы

#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int main()
{
 int casts[10] = { 6, 7, 2, 9, 4, 11, 8, 7, 10, 5 };
 vector<int> dice(10);
 // скопировать из массива в вектор
 copy(casts, casts + 10, dice.begin());
 cout << "Let the dice be cast!\n";
 // создать итератор ostream
 ostream_iterator<int, char> out_iter(cout, " ");
 // скопировать из вектора в поток вывода
 copy(dice.begin(), dice.end(), out_iter);
 cout << endl;
 cout << "Implicit use of reverse iterator.\n";
 copy(dice.rbegin(), dice.rend(), out_iter);
 cout << endl;
 cout << "Explicit use of reverse iterator.\n";
 vector<int>::reverse_iterator ri;
 for (ri = dice.rbegin(); ri != dice.rend(); ++ri)
 cout << *ri << ' ';
 cout << endl;
}

return 0;
}
```

вторую возможность. Это проще и не столь чревато появлением ошибок.

Три других итератора (`back_insert_iterator`, `front_insert_iterator` и `insert_iterator`) также повышают степень обобщения алгоритмов STL. Многие функции STL действуют наподобие `copy()` в том, что они посылают свои результаты туда, куда указывает итератор вывода.

Вспомните, что

```
copy(casts, casts + 10, dice.begin());
```

копирует значения в область, которая начинается с `dice.begin()`. Эти значения переписывают предыдущее содержимое контейнера `dice`, а функция предполагает, что `dice` имеет достаточно места, чтобы поместить значения. Иначе говоря, функция `copy()` не изменяет автоматически размер контейнера назначения в соответствии с объемом данных, посылаемых в него. В листинге 15.7 предусмотрена подобная ситуация, поскольку объявляется, что `dice` имеет 10 элементов, но представьте, что заранее неизвестно, какой должна быть емкость `dice`? Или представьте, что вам нужно добавить элементы в `dice`, а не заменить существующие?

Три итератора вставки решают эти проблемы, превращая процесс копирования в процесс вставки. В результате выполнения вставки добавляются новые элементы, не заменяя существующие данные, и выполняется автоматическое распределение памяти, чтобы удостовериться, что новая информация попадет в контейнер. Итератор `back_insert_iterator` вставляет элементы в конец контейнера, тогда как `front_insert_iterator` делает это в начале контейнера. И наконец, итератор `insert_iterator` вставляет элементы туда, куда указывает аргумент его конструктора. Все три итератора являются моделями концепции итератора вывода.

Существуют и определенные ограничения. Итератор конечной вставки (`back_insert_iterator`) можно использовать только с теми типами контейнеров, которые позволяют выполнять быструю вставку в конце контейнера. ("Быстрый" в данном случае обозначает алгоритм, не изменяющийся со временем; в разделе, посвященном контейнерам, эта концепция рассматривается глубже.) Класс `vector` удовлетворяет такому требованию. Итератор выполнения начальной вставки (`front_insert_iterator`) можно использовать только вместе с контейнерами, позволяющими выполнять вставку, независимую от времени, в начале контейнера. Этому требованию класс `vector` не удовлетворяет. Итератор вставки не имеет таких ограничений. Поэтому его можно использовать для вставки в начале вектора. Однако итератор начальной вставки делает это быстрее для тех типов контейнеров, которые его поддерживают.



## СОВЕТ

Используя итератор вставки, можно изменить алгоритм, копирующий данные так, что он будет вставлять их.

Данные итераторы используют тип контейнера как шаблон аргумента и реальный идентификатор контейнера — как аргумент конструктора. Чтобы создать итератор конечной вставки для контейнера типа `vector<int>` под названием `dice`, нужно воспользоваться таким оператором:

```
back_insert_iterator<vector<int>> back_iter(dice);
```

Объявление итератора начальной вставки имеет точно такую же форму. Объявление итератора вставки имеет дополнительный аргумент конструктора, определяющий место для вставки:

```
insert_iterator<vector<int> >
 insert_iter(dice, dice.begin());
```

Листинг 15.8 иллюстрирует использование этих двух итераторов.



## ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Более ранние версии компиляторов C++ используют `list.h` и `iterator.h`. Кроме того, в них может использоваться итератор `ostream_iterator<int>` вместо `ostream_iterator<int,char>`.

Результаты выполнения программы из листинга 15.8:

```
fine fish fashion fate
fine fish fashion fate busy bats
silly singers fine fish fashion fate busy bats
```

Пусть вас не смущает подобное обилие итераторов, вы сможете научиться использовать их на практике. Помните, что все предопределенные итераторы расширяют степень общности алгоритмов STL. Например, функция `copy()` может копировать информацию не только из одного контейнера в другой, но и из контейнера в поток вывода или из потока ввода в контейнер. Кроме того, используя `copy()`, можно вставить данные в контейнер. Таким образом, одна функция выполняет работу многих других функций. Поскольку `copy()` является одной из нескольких функций STL, использующих итератор вывода, предопределенные итераторы также приумножают возможности других функций.

## Типы контейнеров

Библиотека STL включает концепции контейнеров, и типы контейнеров. Концепции — это общие категории с такими названиями, как контейнер, последовательный контейнер, ассоциативный контейнер и т.д. Типы контейнеров — это шаблоны, которые можно использовать для создания определенных объектов-контейнеров. Одиннадцать типов контейнеров именуются `deque`, `list`, `queue`, `priority_queue`, `stack`, `vector`, `map`, `multimap`, `set`,

### Листинг 15.8 Программа inserts.cpp.

```
// inserts.cpp - функция copy() и итераторы вставки
#include <iostream>
#include <string>
#include <iiterator>
#include <vector>
using namespace std;

int main()
{
 string s1[4] = { "fine", "fish", "fashion", "fate"};
 string s2[2] = { "busy", "bats"};
 string s3[2] = { "silly", "singers"};
 vector<string> words(4);
 copy(s1, s1 + 4, words.begin());
 copy(s2, s2 + 2, words.begin());
 ostringstream out(cout, ' ');
 copy (words.begin(), words.end(), out);
 cout << endl;

 // создание анонимного объекта типа back_insert_iterator
 copy(s2, s2 + 2, back_insert_iterator<vector<string> >(words));
 copy (words.begin(), words.end(), out);
 cout << endl;

 // создание анонимного объекта типа insert_iterator
 copy(s3, s3 + 2, insert_iterator<vector<string> >(words, words.begin()));
 copy (words.begin(), words.end(), out);
 cout << endl;

 return 0;
}
```

**multiset** и **bitset**. (В этой главе тип **bitset** не рассматривается; он представляет собой контейнер для работы с данными на уровне битов.) Мы начнем с концепций, поскольку именно они определяют категории типов контейнеров.

#### Концепция контейнера

Не существует типа, отвечающего базовой концепции контейнера, но сама концепция описывает элементы, общие для всех классов контейнеров. Концепция контейнера — это концептуальный абстрактный базовый класс (концептуальный, потому что классы-контейнеры реально не используют механизм наследования). Другими словами, концепция контейнера устанавливает набор требований, которым должны удовлетворять все классы-контейнеры в STL.

Контейнер — это объект, содержащий другие объекты, причем одного типа. Сохраняемые объекты могут быть объектами в смысле ООП или же значениями встроенных типов. Данными, сохраненными в контейнере, владеет сам контейнер. Это значит, что, если контейнер прекращает свое существование, то же самое происходит и с данными. (Однако если данными в контейнере являются указатели, то данные, на которые они указывают, не обязательно исчезнут.)

В контейнере нельзя сохранить любой объект. В частности, типы объектов должны быть способны к *конструированию с точки зрения копирования и присваивания*. Базовые типы удовлетворяют этим требованиям, то же самое относится и к классам, если только определение класса не делает конструктор копирования или оператор присваивания приватным или защищенным.

Базовый контейнер не гарантирует, что его элементы сохранены в каком-либо определенном порядке или что этот порядок неизменен, но уточнения к концепции могут обеспечить такие гарантии. Все контейнеры имеют определенные свойства и обеспечивают выполнение определенных операций. В табл. 15.5 представлены несколько из таких общих свойств. В таблице X обозначает тип контейнера, такой как **vector**, T — тип объекта, сохраненного в контейнере, а u — значения типа X, а b — идентификатор типа X.

В столбце "Сложность" описываются затраты времени, необходимые для выполнения оператора. В данной таблице представлены три варианта, размещенные в порядке от самого быстрого до самого медленного:

- Время компиляции
- Постоянная
- Линейная

Таблица 15.5 Некоторые базовые свойства контейнеров.

| Выражение                       | Возвращаемый тип                              | Комментарий                                                                                                                                                                                         | Сложность        |
|---------------------------------|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|
| <code>X::iterator</code>        | Тип итератора, указывающего на <code>T</code> | Любая категория итераторов, кроме итератора вывода                                                                                                                                                  | Время компиляции |
| <code>X::value_type</code>      | <code>T</code>                                | Тип для <code>T</code>                                                                                                                                                                              | Время компиляции |
| <code>X u;</code>               |                                               | Создает контейнер нулевого размера под названием <code>u</code>                                                                                                                                     | Постоянная       |
| <code>X();</code>               |                                               | Создает анонимный контейнер нулевого размера                                                                                                                                                        | Постоянная       |
| <code>X u(a);</code>            |                                               | Конструктор копирования                                                                                                                                                                             | Линейная         |
| <code>X u = a;</code>           |                                               | Тот же эффект, что и <code>X u(a);</code>                                                                                                                                                           | Линейная         |
| <code>(&amp;a)-&gt;-X();</code> | <code>void</code>                             | Выполнить деструктор для каждого элемента контейнера                                                                                                                                                | Линейная         |
| <code>a.begin()</code>          | <code>iterator</code>                         | Возвращает итератор, указывающий на первый элемент контейнера                                                                                                                                       | Постоянная       |
| <code>a.end()</code>            | <code>iterator</code>                         | Возвращает итератор, указывающий на элемент, находящийся за последним элементом контейнера                                                                                                          | Постоянная       |
| <code>a.size()</code>           | Беззнаковый интегрального типа                | Количество элементов, равное <code>a.end()</code> и <code>a.begin()</code>                                                                                                                          | Постоянная       |
| <code>a.swap(b)</code>          | <code>void</code>                             | Обмен значениями <code>a</code> и <code>b</code>                                                                                                                                                    | Постоянная       |
| <code>a == b</code>             | Конвертируемый в <code>bool</code>            | Справедливо, если <code>a</code> и <code>b</code> имеют одинаковый размер и каждый элемент в <code>a</code> эквивалентен ( <code>==</code> справедливо), соответствующему элементу в <code>b</code> | Линейная         |
| <code>a != b</code>             | Конвертируемый в <code>bool</code>            | То же самое, что и <code>!(a == b)</code>                                                                                                                                                           | Линейная         |

Если сложность описывается временем компиляции, то действие происходит на этапе компиляции и во время запуска программы времени на выполнение не требуется. Сложность, описываемая по линейному закону, обозначает, что операция выполняется во время выполнения программы, но не зависит от количества элементов в объекте. Сложность, описываемая по линейному закону, обозначает, что время выполнения операции пропорционально количеству элементов. Таким образом, если `a` и `b` — контейнеры, то `a == b` описывается по линейному закону, так как операция `==` должна быть выполнена для каждого элемента в контейнере. Фактически — это наихудший сценарий. Если два контейнера имеют разный размер, то индивидуального сравнения элементов выполнять не нужно:

#### СЛОЖНОСТЬ ОПРЕДЕЛЯЕМАЯ ПО ЛИНЕЙНОМУ И ПОСТОЯННОМУ ЗАКОНУ

Представьте себе длинную узкую коробку, открытую с конца, и залитую большими пакетами, выстроенные в линию. Представьте, что моя задача — вытащить пакет у открытого края. Это пример задачи с постоянным временем выполнением. Неважно, сколько пакетов, 10 или 1000, находится за первым.

Теперь, пусть моя задача — достать пакет, находящийся у закрытого края коробки. Это уже пример задачи с линейным временем выполнением. Если всего 10 пакетов, то мне нужно вытащить 10 пакетов вместе с находящимся у закрытого края. Если 100 пакетов, то нужно вытащить 100 пакетов. Предположим, что я неутомимый работник, вытаскивающий один пакет за один раз, тогда второй вариант займет в 10 раз больше времени.

Пусть теперь мне нужно вытащить определенный пакет. Может оказаться, что это первый пакет. Однако в среднем количество пакетов, которые необходимо вытащить, все равно пропорционально количеству пакетов в контейнере, поэтому сложность данной задачи тоже определяется по линейному закону.

Замена длинной узкой коробки на такую же, но без сторон, превратит эту задачу в задачу, сложность которой определяется постоянным временем, поскольку можно будет забрать необходимый пакет, не двигая остальные.

Идея временной сложности охватывает влияние размера контейнера на время выполнения, но не затрагивает других факторов. Если супергерой может разгружать пакеты из коробки, открытой с одной стороны, в 1000 раз быстрее, чем я, задача, выполняемая им, все равно имеет сложность, которая линейно зависит от времени. В таком случае его линейно-временное быстродействие с закрытой коробкой (открыта с одной стороны) будет выше, чем мое константно-временное быстродействие с коробкой без сторон, лишь до тех пор, пока в коробках находится не слишком много пакетов.

Требования к уровню сложности — это одна из характеристик STL. Если детали реализации могут быть и скрытыми, то спецификации быстродействия должны быть общедоступными, чтобы знать, сколько времени занимает выполнение определенной операции.

## Последовательности

Уточнить базовую концепцию контейнера можно, добавив требования. *Последовательность* — это важное уточнение, так как шесть типов контейнеров в STL (`deque`, `list`, `queue`, `priority_queue`, `stack` и `vector`) являются последовательностями. (Вспомните, что очередь (`queue`) позволяет добавлять элементы в конце и удалять их в начале. Двусторонняя очередь, представленная контейнером `deque`, позволяет добавлять и удалять элементы с обеих сторон.) Концепция последовательности добавляет требование, чтобы итератор был как минимум прямым. Это, в свою очередь, гарантирует, что элементы упорядочены определенным образом и этот порядок неизменен от итерации к итерации.

Последовательность также требует, чтобы ее элементы были выстроены в строго линейном порядке. Это значит, что существует первый элемент, последний элемент, а каждый элемент, кроме этих двух, имеет один элемент перед собой и один элемент после. Массив и связанный список являются примерами последовательности, в то время как разветвленная структура (в которой каждый узел указывает на два дочерних узла) — нет.

Поскольку элементы в последовательности имеют определенный порядок, возможны такие операции, как вставка значения в определенное место и удаление определенного диапазона. В табл. 15.6 перечислены эти и другие операции, обязательные для последовательности. В таблице применяется такая же запись, как и в табл. 15.5, с некоторыми добавлениями: `t` обозначает значение типа `T`, т.е. значение, сохраняемое в контейнере, `n` — значение целого типа, `p`, `q`, `i` и `j` — итераторы.

Поскольку все шаблоны классов `deque`, `list`, `queue`, `stack` и `vector` являются моделью концепции последовательности, они поддерживают операторы, приведенные в табл. 15.6. В дополнение к этому существуют операторы, доступные лишь для некоторых из описанных пяти моделей. Они описываются в терминах сложности, характеризуемой постоянным временем. В табл. 15.7 перечислены эти добавочные операции.

Эта таблица требует некоторых комментариев. Прежде всего, вы заметили, что `a[n]` и `a.at(n)` возвращает ссылку на `n`-й элемент (отсчитывая с нуля) контейнера. Разница заключается в том, что `a.at(n)` выполняет проверку границ контейнера и генерирует прерывание `out_of_range`, если `n` выходит за пределы границ. Далее, вы можете удивиться, почему, скажем, функция `push_front()` определена для `list` и `deque`, а не для `vector`.

Таблица 15.6 Требования к последовательности.

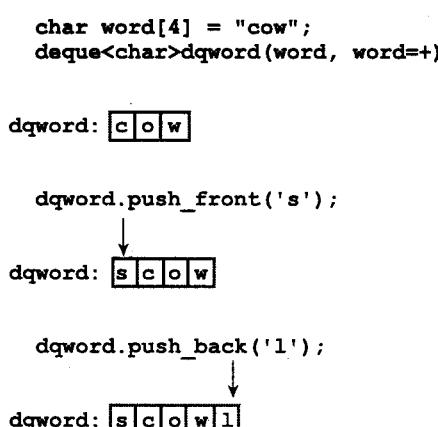
| Выражение                    | Возвращаемый тип | Комментарии                                                                                              |
|------------------------------|------------------|----------------------------------------------------------------------------------------------------------|
| <code>X a(n,t);</code>       |                  | Объявляет последовательность <code>a</code> , состоящую из <code>n</code> копий значения <code>t</code>  |
| <code>X(n, t)</code>         |                  | Создает анонимную последовательность, состоящую из <code>n</code> копий значения <code>t</code>          |
| <code>X a(i, j)</code>       |                  | Объявляет последовательность <code>a</code> , инициализированную содержимым диапазона <code>[i,j)</code> |
| <code>X(i, j)</code>         |                  | Создает анонимную последовательность, инициализированную содержимым диапазона <code>[i,j)</code>         |
| <code>a.insert(p,t)</code>   | iterator         | Вставляет копию <code>t</code> перед <code>p</code>                                                      |
| <code>a.insert(p,n,t)</code> | void             | Вставляет <code>n</code> копий <code>t</code> перед <code>p</code>                                       |
| <code>a.insert(p,i,j)</code> | void             | Вставляет копии элементов из диапазона <code>[i,j)</code> перед <code>p</code>                           |
| <code>a.erase(p)</code>      | iterator         | Удаляет элемент, на который указывает <code>p</code>                                                     |
| <code>a.erase(p,q)</code>    | iterator         | Удаляет элементы в диапазоне <code>[p,q)</code>                                                          |
| <code>a.clear()</code>       | void             | То же самое, что и <code>erase(begin(), end())</code>                                                    |

Таблица 15.7 Дополнительные требования к последовательности.

| Выражение       | Возвращаемый тип | Значение               | Контейнер           |
|-----------------|------------------|------------------------|---------------------|
| a.front()       | T&               | *a.begin()             | vector, list, deque |
| a.back()        | T&               | *--a.end()             | vector, list, deque |
| a.push_front(t) | void             | a.insert(a.begin(), t) | vector, deque       |
| a.push_back(t)  | void             | a.insert(a.end(), t)   | vector, list, deque |
| a.pop_front()   | void             | a.erase(a.begin())     | vector, deque       |
| a.pop_back()    | void             | a.erase(--a.end())     | vector, list, deque |
| a[n]            | T&               | *(a.begin() + n)       | vector, deque       |
| a.at(n)         | T&               | *(a.begin() + n)       | vector, deque       |

Представьте, что нужно вставить новое значение в начало вектора из 100 элементов. Чтобы освободить место, необходимо передвинуть 99-й элемент в позицию 100, затем 98-й — в позицию 99 и т.д. Это операция имеет сложность, описываемую линейным законом, поскольку перемещение 100 элементов в 100 раз дольше, чем перемещение одного. А операции из табл. 15.7 реализуются лишь в том случае, если им соответствует сложность, описываемая постоянным временем. Однако структура списков и двусторонних очередей позволяет добавить элемент в начало без перемещения остальных элементов на другие позиции, поэтому операцию `push_front()` можно реализовать с константно-временной сложностью. Рис. 15.4 иллюстрирует работу функций `push_front()` и `push_back()`.

Давайте подробнее рассмотрим шесть типов контейнеров.



## Класс vector

Вы уже видели несколько примеров, использующих шаблон `vector`, объявленный в заголовочном файле `vector`. Можно сказать, что `vector` — это представление массива в виде класса. Класс обеспечивает автоматическое управление памятью, позволяющее динамически изменять размер объекта класса `vector`, увеличивая или уменьшая его по мере добавления или удаления элементов. Класс обеспечивает произвольный доступ к элементам. Элементы в конце можно добавлять или удалять за постоянное время, но вставка и удаление элементов в начале и середине является операцией, описываемой линейным законом.

Вектор не только является последовательностью, но и представляет собой модель концепции *обратимого контейнера*. При этом добавляются еще два метода класса: `rbegin()`, который возвращает итератор к первому элементу обратной последовательности, и `rend()` — итератор на элемент, находящийся за последним элементом обратной последовательности. Так, если `dice` — это контейнер типа `vector<int>`, а `Show(int)` — функция, выводящая целое значение, то следующий код отобразит содержимое контейнера `dice` сначала в прямом порядке, а затем в обратном:

```

// отобразить в прямом порядке
for_each(dice.begin(), dice.end(), Show);
cout << endl;

// отобразить в обратном порядке
for_each(dice.rbegin(), dice.rend(), Show);
cout << endl;

```

Итератор, возвращаемый двумя методами, имеет тип `reverse_iterator`. Вспомните, что увеличение такого итератора заставляет его просматривать обратимый контейнер в обратном порядке.

Класс шаблонов `vector` — это самый простой из типов последовательностей, и он должен использоваться

РИСУНОК 15.4 Функции `push_front()` и `push_back()`.

по умолчанию, если только требования программы не удовлетворяются лучше средствами других типов.

## Класс `deque`

Класс шаблонов `deque` (объявленный в заголовочном файле `deque`) представляет собой *двустороннюю очередь*. В реализации STL она во многом сходна с вектором и поддерживает произвольный доступ. Основное различие заключается в том, что вставка и удаление элементов в начале двусторонней очереди являются операциями, сложность которых не зависит от времени, а для *вектора* это операции с линейным временем выполнения. Так что если доступ к началу и концу последовательности выполняется довольно часто, используйте структуру данных `deque`.

Необходимость вставки и удаления элементов, не зависящих от времени, с обоих концов двусторонней очереди делает ее реализацию более сложной, чем реализацию вектора. Поэтому, хотя обе структуры и позволяют выполнить произвольный доступ к своим элементам, а также вставку и удаление элементов, определяемые линейным законом, из середины последовательности, с вектором эти операции осуществляются быстрее.

## Класс `list`

Класс шаблонов `list` (объявленный в заголовочном файле `list`) представляет собой *двусвязный список*. Каждый элемент, кроме первого и последнего, связан с элементом до него и после него, это значит, что по списку можно пройти в двух направлениях. Основное различие между `list` и `vector` заключается в том, что `list` обеспечивает вставку и удаление элементов, не зависящие от времени, в любом месте списка. (Вспомните, что шаблон класса `vector` обеспечивает вставку и удаление по линейному закону везде, кроме конца, где вставка и удаление

не зависят от времени.) Таким образом, `vector` делает основным быстрый доступ по сравнению с произвольным доступом, в то время как `list` делает основной операцию быстрой вставки и удаления элементов.

Как и класс `vector`, класс `list` является обратимым контейнером. В отличие от вектора, список не поддерживает запись в форме массива и произвольный доступ. Итератор списка, в отличие от итератора вектора, продолжает указывать на тот же элемент, что и раньше, даже после выполнения вставки или удаления элементов из контейнера. Давайте поясним это утверждение. Представьте себе, что есть итератор, указывающий на пятый элемент контейнера `vector`. Затем в начало контейнера вставляется элемент. Все другие элементы сдвигаются, чтобы освободить место, поэтому после вставки пятый элемент содержит значение, находившееся ранее в четвертом. Тогда итератор указывает на то же местоположение, но на другие данные. При вставке нового элемента в список не сдвигаются существующие элементы, только изменяется информация о связях. Поэтому итератор, указывавший на определенный элемент, продолжает указывать на него, но этот элемент может быть связан уже с другими элементами.

Класс шаблонов `list` в дополнение к функциям, которые присущи последовательности и обратимому контейнеру, имеет несколько функций-элементов, ориентированных специально на работу со списком. В табл. 15.8 перечислено большинство из них. (Полный список методов и функций STL приведен в приложении G.) Параметр шаблона `Alloc` имеет обычно значение, принятое по умолчанию, поэтому о нем не нужно беспокоиться.

Листинг 15.9 иллюстрирует применение этих методов вместе с методом `insert()`.

Таблица 15.8 Некоторые методы класса `list`.

| Функция                                                        | Описание                                                                                                                                                                                                                                                             |
|----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void merge(list&lt;T, Alloc&gt;&amp; x)</code>           | Объединяет список <code>x</code> с исходным списком. Оба списка должны быть отсортированы. Результирующий отсортированный список сохранен в исходном контейнере, а <code>x</code> оставлен пустым. Данная функция имеет сложность, определенную по линейному закону. |
| <code>void remove(const T &amp; val)</code>                    | Удаляет все копии <code>val</code> из списка. Данная функция описывается в терминах сложности, определяемой по линейному закону.                                                                                                                                     |
| <code>void sort()</code>                                       | Сортирует список, используя оператор <code>&lt;</code> ; сложность данной функции определяется по формуле $N \log N$ для $N$ элементов.                                                                                                                              |
| <code>void splice(iterator pos, list&lt;T, Alloc&gt; x)</code> | Вставляет содержимое списка <code>x</code> перед позицией <code>pos</code> , а <code>x</code> оставляет пустым. Данная функция имеет сложность, не зависящую от времени.                                                                                             |
| <code>void unique()</code>                                     | Объединяет каждую последовательную группу одинаковых элементов в один элемент. Данная функция описывается в терминах сложности, определяемой линейным законом.                                                                                                       |

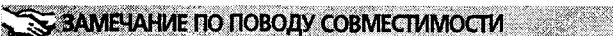
## Листинг 15.9 Программа list.cpp.

```
// list.cpp - использование списка
#include <iostream>
#include <list>
#include <iterator>
using namespace std;

int main()
{
 list<int> one(5, 2); // список из 5 двоек
 int stuff[5] = { 1, 2, 4, 8, 6 } ;
 list<int> two;
 two.insert(two.begin(), stuff, stuff + 5);
 int more[6] = { 6, 4, 2, 4, 6, 5 } ;
 list<int> three(two);
 three.insert(three.end(), more, more + 6);

 cout << "List one: ";
 ostream_iterator<int,char> out(cout, " ");
 copy(one.begin(), one.end(), out);
 cout << endl << "List two: ";
 copy(two.begin(), two.end(), out);
 cout << endl << "List three: ";
 copy(three.begin(), three.end(), out);
 three.remove(2);
 cout << endl << "List three minus 2s: ";
 copy(three.begin(), three.end(), out);
 three.splice(three.begin(), one);
 cout << endl << "List three after splice: ";
 copy(three.begin(), three.end(), out);
 cout << endl << "List one: ";
 copy(one.begin(), one.end(), out);
 three.unique();
 cout << endl << "List three after unique: ";
 copy(three.begin(), three.end(), out);
 three.sort();
 three.unique();
 cout << endl << "List three after sort & unique: ";
 copy(three.begin(), three.end(), out);
 two.sort();
 three.merge(two);
 cout << endl << "Sorted two merged into three: ";
 copy(three.begin(), three.end(), out);
 cout << endl;

 return 0;
}
```

 ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Более ранние версии используют заголовочные файлы **list.h** и **iterator.h**. Кроме того, в старых версиях может использоваться **ostream\_iterator<int>** вместо **ostream\_iterator<int,char>**.

Результат выполнения программы из листинга 15.9:

```
List one: 2 2 2 2 2
List two: 1 2 4 8 6
List three: 1 2 4 8 6 6 4 2 4 6 5
List three minus 2s: 1 4 8 6 6 4 4 6 5
List three after splice: 2 2 2 2 2 1 4 8 6 6
 ↪ 4 4 6 5
List one:
List three after unique: 2 1 4 8 6 4 6 5
List three after sort & unique: 1 2 4 5 6 8
Sorted two merged into three: 1 1 2 2 4 4 5
 ↪ 6 6 8 8
```

## Примечания к программе

В программе для вывода содержимого контейнера применяются методы, рассмотренные ранее, по использованию общей функции STL **copy()** и объекта **ostream\_iterator**.

Основное различие между функциями **insert()** и **splice()** состоит в том, что **insert()** вставляет копию исходного диапазона в назначенное место, в то время как **splice()** перемещает туда исходный диапазон. Поэтому, после того как содержимое **one** перемещено в **three** функцией **splice()**, контейнер **one** становится пустым. (Метод **splice()** имеет несколько прототипов для перемещения отдельных элементов и их диапазонов.) Метод **splice()** сохраняет итераторы действующими. Это значит, что если определенный итератор указывает на не-

который элемент в `one`, то итератор все равно будет указывать на этот же элемент даже после того, как функция `splice()` переместит его в `three`.

Заметьте, что функция `unique()` просто заменяет соседние равные значения одним значением. Когда программа запускает метод `three.unique()`, контейнер `three` содержит две цифры "4" и две цифры "6", не являющиеся соседними. Но применение функции `sort()`, а затем `unique()` ограничит количество вхождений каждой цифры до одного.

Существует функция `sort()`, не являющаяся элементом класса (листинг 15.6), но для выполнения она требует наличия итератора произвольного доступа. Поскольку компромисс при рассмотрении быстрой вставки был решен в пользу произвольного доступа, функцию класса `sort()`, не являющуюся функцией-элементом, нельзя использовать совместно со списком. Поэтому класс включает функцию-элемент, которая работает и при ограничениях, вносимых классом.

### Инструментарий для работы с контейнером `list`

Методы контейнера `list` формируют удобный инструментарий. Представьте, например, что нужно организовать два почтовых списка. Каждый из них можно сортировать, объединить их, а затем использовать `unique()` для удаления повторных вхождений.

Каждый из методов `sort()`, `merge()` и `unique()` имеет также и версию, имеющую дополнительный аргумент, определяющий альтернативную функцию для сравнения элементов. Подобным образом метод `remove()` имеет версию с дополнительным аргументом, определяющим функцию, которая проверяет, был удален данный элемент или нет. Эти аргументы являются примерами функций-предикатов — тема, к которой мы вернемся позднее.

### Класс `queue`

Класс шаблонов `queue` (объявленный в заголовочном файле `queue` (ранее `queue.h`)) — это класс-адаптер. Вспомните, что шаблон `ostream_iterator` — это итера-

тор, позволяющий использовать интерфейс итератора для потока вывода. Аналогично шаблон `queue` позволяет основному классу (по умолчанию `deque`) установить типичный для очереди интерфейс.

Шаблон `queue` имеет больше ограничений, чем класс `deque`. Он не позволяет не только выполнить произвольный доступ к элементам очереди, но даже пошагово перемещаться по очереди. Он ограничивает использование очереди до базовых операций, входящих в определение очереди. Можно добавить элемент в конце очереди, удалить элемент в начале очереди, просматривать значения в начале и конце, проверять количество элементов и устанавливать, не является ли очередь пустой. Эти операции перечислены в табл. 15.9.

Заметьте, что `pop()` — это метод, применяемый для удаления данных, а не для их получения. Если необходимо использовать значение из очереди, то сначала нужно выполнить функцию `front()` для получения значения, а затем `pop()` — для удаления его из очереди.

### Класс `priority_queue`

Класс шаблонов `priority_queue` (также объявленный в заголовочном файле `queue`) — еще один пример класса-адаптера. Он поддерживает такие же операции, как и класс `queue`. Основное различие состоит в том, что наибольший элемент в `priority_queue` сдвигается в начало. (В жизни не всегда все поступают честно, то же самое происходит и с очередями.) Внутреннее различие заключается в том, что данный класс является вектором. Можно изменить способ сравнения, используемый для того, чтобы определить, что именно перемещать в начало очереди, если воспользоваться дополнительным аргументом конструктора:

```
// версия, принятая по умолчанию
priority_queue<int> pq1;
//использование greater<int> для упорядочения
priority_queue<int> pq2(greater<int>);
```

Функция `greater<>()` является предопределенной для данного объекта, она рассматривается далее в этой главе.

Таблица 15.9 Операции, выполняемые с классом `queue`.

| Метод                                  | Описание                                                                              |
|----------------------------------------|---------------------------------------------------------------------------------------|
| <code>bool empty() const</code>        | Возвращает значение "истина", если очередь является пустой, и "ложь" — в ином случае. |
| <code>size_type size() const</code>    | Возвращает количество элементов в очереди.                                            |
| <code>T&amp; front()</code>            | Возвращает ссылку на элемент, находящийся в начале очереди.                           |
| <code>T&amp; back()</code>             | Возвращает ссылку на элемент, находящийся в конце очереди.                            |
| <code>void push(const T&amp; x)</code> | Добавляет <code>x</code> в конец очереди.                                             |
| <code>void pop()</code>                | Удаляет элемент, находящийся в начале очереди.                                        |

## Класс stack

Как и класс `queue`, класс `stack` (объявленный в заголовочном файле `stack` — ранее `stack.h`) тоже является классом-адаптером. Он позволяет лежащему в его основе классу (классу `vector`, заданному по умолчанию) использовать обычный интерфейс стека.

Класс шаблонов `stack` имеет больше ограничений, чем класс `vector`. Он не только не позволяет выполнить произвольный доступ к элементам стека, но даже не дает возможности пошагового перемещаться по стеку. Он ограничивает использование стека базовыми операциями, входящими в определение стека. Можно добавить значение в вершину стека, удалить значение из вершины стека, прочитать значение из вершины стека, проверить количество элементов и установить, не является ли стек пустым. Эти операции перечислены в табл. 15.10.

Как и в случае с очередью, если необходимо использовать значение из стека, сначала нужно воспользоваться функцией `top()` для получения значения, а затем функцией `pop()` для его удаления из стека.

## Ассоциативные контейнеры

**Ассоциативный контейнер** представляет собой следующее уточнение концепции контейнера. Ассоциативный контейнер ставит в соответствие значению ключ, а затем использует ключ для поиска значения. Например, в качестве значений могут использоваться структуры, содержащие информацию о сотрудниках, такую как имя, адрес, номер офиса, домашний и рабочий телефоны, медицинскую карточку и т.д., а в качестве ключа — уникальный номер сотрудника. Для получения информации о сотруднике программа воспользуется ключом, чтобы найти необходимую структуру. Вспомните, что, в общем, для контейнера `X` выражение `X::value_type` показывает тип значений, сохраненных в контейнере. Для ассоциативного контейнера выражение `X::key_type` показывает, какой тип используется для ключа.

Достоинство ассоциативного контейнера заключается в том, что он обеспечивает быстрый доступ к своим элементам. Как и последовательность, ассоциативный контейнер позволяет вставлять новые элементы; однако

для них нельзя указать конкретное местоположение. Причина состоит в том, что для ассоциативного контейнера обычно имеется специальный алгоритм для определения того, куда следует поместить данные, чтобы затем информацию можно было получить быстро.

В библиотеке STL имеется четыре ассоциативных контейнера: `set`, `multiset`, `map` и `multimap`. Первые два типа определены в заголовочном файле `set` (ранее они были разделены на `set.h` и `multiset.h`), а два вторых — в заголовочном файле `map` (ранее они были разделены на `map.h` и `multimap.h`).

Наиболее простым из ассоциативных контейнеров является `set`; тип значений — такой же, как и тип ключей, причем ключи являются уникальными, а это значит, что существует не более одной копии ключа в множестве. Для класса `set` фактически значение и является ключом. Тип `multiset` очень похож на тип `set`, кроме того, что он может содержать более одного значения с одинаковым ключом. Например, если и ключи, и значения имеют тип `int`, то объект класса `multiset` может содержать, скажем, 1, 2, 2, 2, 3, 5, 7, 7.

Для типа `map` тип значений отличается от типа ключей, причем ключи являются уникальными (только одно значение соответствует ключу). Тип `multimap` похож на `map`, с той лишь разницей, что одному ключу может соответствовать несколько значений.

Подробно рассказать об этих типах трудно в рамках одной главы (в приложении G перечислены методы этих классов), поэтому мы рассмотрим простые примеры, использующие `set` и `multimap`.

### Пример класса `set`

Класс `set` в STL моделирует несколько концепций. Он является ассоциативным, обратимым, отсортированным контейнером, а ключи в нем — уникальными, поэтому он может содержать не более одного вхождения каждого значения. Как `vector` и `list`, класс `set` использует параметр шаблона, определяющий, какой тип сохраняется в контейнере:

```
set<string> A; // объект класса set,
// состоящий из объектов класса string
```

Таблица 15.10 Операции, выполняемые с классом `stack`.

| Метод                                  | Описание                                                                           |
|----------------------------------------|------------------------------------------------------------------------------------|
| <code>bool empty() const</code>        | Возвращает значение “истина”, если стек является пустым, и “ложь” — в ином случае. |
| <code>size_type size() const</code>    | Возвращает количество элементов в стеке.                                           |
| <code>T&amp; top()</code>              | Возвращает ссылку на элемент, находящийся в вершине стека.                         |
| <code>void push(const T&amp; x)</code> | Помещает элемент в вершину стека.                                                  |
| <code>void pop()</code>                | Удаляет элемент из вершины стека.                                                  |

Второй, дополнительный аргумент шаблона используется для указания функции сравнения или объекта, необходимого для упорядочения ключей. По умолчанию применяется шаблон `less<>` (рассматриваемый далее). Более ранние реализации могут не обеспечивать значения, заданные по умолчанию, и поэтому не могут требовать наличия явного параметра шаблона:

```
// более ранняя реализация
set<string, less<string> > A;
```

Рассмотрим следующий программный код:

```
const int N = 6;
string s1[N] = { "buffoon", "thinkers",
 "for", "heavy", "can", "for"};
//инициализация объекта A класса set с
//использованием диапазона значений массива
set<string> A(s1, s1 + N);
ostream_iterator<string, char> out(cout, " ");
copy(A.begin(), A.end(), out);
```

Как и другие контейнеры, класс `set` имеет конструктор (см. табл. 15.6), получающий диапазон итераторов в качестве аргументов. Это обеспечивает простой способ для инициализации объекта `set` содержимым массива. Помните, что последний элемент диапазона является элементом, находящимся за последним реальным элементом, поэтому `s1 + N` указывает на одну позицию за концом массива `s1`. Вывод данных для приведенного фрагмента кода показывает, что ключи являются уникальными (строка "for" появляется дважды в массиве, но только один раз в множестве), а множество — отсортированным:

```
buffoon can for heavy thinkers
```

В математике определяется несколько стандартных операций над множествами. Объединением двух множеств является множество, в котором находится содержимое обоих исходных множеств. Если определенное значение является общим для обоих множеств, то в объединении оно появляется только один раз, поскольку ключи должны быть уникальными. Пересечением двух множеств является множество, состоящее из элементов, общих для обоих множеств. Разность двух множеств — это первое множество, исключая элементы, общие для обоих множеств.

Библиотека STL обеспечивает алгоритмы, поддерживающие эти операции. Они больше являются общими функциями, чем методами, поэтому их применение не ограничено объектами `set`. Однако все объекты `set` автоматически удовлетворяют необходимым условиям для применения этих алгоритмов, например, потому, что контейнер является отсортированным.

Функция `set_union()` имеет пять аргументов-итераторов. Первые два определяют диапазон в одном множестве, вторые два — во втором, а последний — местоположение, куда необходимо скопировать результирующее множество. Например, чтобы вывести содержимое объединение множеств `A` и `B`, можно поступить так:

```
set_union(A.begin(), A.end(), B.begin(),
 B.end(), ostream_iterator<string, char>
 out(cout, " "));
```

Представьте, что результат необходимо не отображать, а поместить в множество `C`. Тогда последний аргумент будет итератором множества `C`. Очевидный выбор — это `C.begin()`, но он не срабатывает по двум причинам. Первая причина состоит в том, что ассоциативное множество считает ключи постоянными значениями, поэтому итератор, возвращаемый функцией `C.begin()`, является постоянным. По этой причине он не может быть использован в качестве итератора вывода. Вторая причина, по которой нельзя прямо использовать `C.begin()`, состоит в том, что функция `set_union()`, как и функция `copy()`, перезаписывает существующие в контейнере данные и требует, чтобы контейнер имел достаточно места для сохранения новой информации. Контейнер `C`, будучи пустым, не удовлетворяет этому требованию. Обе проблемы решает рассмотренный ранее шаблон `insert_iterator`. Вы уже знаете, что он преобразует копирование во вставку элементов. Кроме того, он моделирует концепцию итератора вывода, поэтому его можно использовать для записи данных в контейнер. Таким образом, можно сконструировать анонимный `insert_iterator` для копирования информации в контейнер `C`. Вспомните, что конструктор получает имя контейнера и итератор в качестве аргументов:

```
set_union(A.begin(), A.end(), B.begin(),
 B.end(), insert_iterator<set<string> >
 (C, C.begin()));
```

Функции `set_intersection()` и `set_difference()` находят пересечение и разность двух множеств и имеют точно такой же интерфейс, как и функция `set_union()`.

Два полезных метода в классе множеств — это `lower_bound()` и `upper_bound()`. Метод `lower_bound()` получает ключ в качестве своего аргумента и возвращает итератор, указывающий на первый элемент множества, который не меньше, чем аргумент-ключ. Аналогично метод `upper_bound()` получает ключ в качестве аргумента и возвращает итератор, указывающий на первый элемент множества, который больше, чем аргумент-ключ. Например, если имеется множество строк, этими методами можно воспользоваться для определения диапазона строк, заключающего в себе все строки от "b" до "f".

Поскольку сортировка определяет, куда поместить новые элементы множества, методы вставки элементов, имеющиеся у данного класса, не требуют указания позиции вставки. Например, если A и B — это множества строк, то можно сделать следующее:

```
string s("tennis");
A.insert(s); // вставить значение
B.insert(A.begin(), A.end()); // вставить
 // диапазон
```

### Листинг 15.10 Программа set.cpp.

```
// set.cpp - некоторые операции, выполняемые с классом set

#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
 const int N = 6;
 string s1[N] = { "buffoon", "thinkers", "for", "heavy", "can", "for" } ;
 string s2[N] = { "metal", "any", "food", "elegant", "deliver", "for" } ;

 set<string> A(s1, s1 + N);
 set<string> B(s2, s2 + N);

 ostream_iterator<string, char> out(cout, " ");
 cout << "Set A: ";
 copy(A.begin(), A.end(), out);
 cout << endl;
 cout << "Set B: ";
 copy(B.begin(), B.end(), out);
 cout << endl;

 cout << "Union of A and B:\n";
 set_union(A.begin(), A.end(), B.begin(), B.end(), out);
 cout << endl;

 cout << "Intersection of A and B:\n";
 set_intersection(A.begin(), A.end(), B.begin(), B.end(), out);
 cout << endl;

 cout << "Difference of A and B:\n";
 set_difference(A.begin(), A.end(), B.begin(), B.end(), out);
 cout << endl;

 set<string> C;
 cout << "Set C:\n";
 set_union(A.begin(), A.end(), B.begin(), B.end(),
 insert_iterator<set<string> >(C, C.begin()));
 copy(C.begin(), C.end(), out);
 cout << endl;

 string s3("grungy");
 C.insert(s3);
 cout << "Set C after insertion:\n";
 copy(C.begin(), C.end(), out);
 cout << endl;

 cout << "Showing a range:\n";
 copy(C.lower_bound("ghost"), C.upper_bound("spook"), out);
 cout << endl;

 return 0;
}
```

Листинг 15.10 иллюстрирует применение множеств.

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В более ранних реализациях используются файлы `set.h`, `iterator.h` и `algo.h`. Может потребоваться `less<string>` в качестве второго аргумента для класса `set`. Кроме того, ранние версии могут использовать `ofstream_iterator<string,char>` вместо `ostream_iterator<string,char>`.

Результаты выполнения программы из листинга 15.10:

```
Set A: buffoon can for heavy thinkers
Set B: any deliver elegant food for metal
Union of A and B:
any buffoon can deliver elegant food for
 ^heavy metal thinkers
Intersection of A and B:
for
Difference of A and B:
buffoon can heavy thinkers
Set C:
any buffoon can deliver elegant food for
 ^heavy metal thinkers
Set C after insertion:
any buffoon can deliver elegant food for
 ^grungy heavy metal thinkers
Showing a range:
grungy heavy metal
```

### Пример класса multimap

Как и класс `set`, класс `multimap` представляет собой обратимый, отсортированный, ассоциативный контейнер. Однако тип ключа отличается от типа сохраняемых значений, и объект `multimap` может иметь более одного значения, поставленного в соответствие определенному ключу.

В базовом объявлении класса `multimap` используется тип ключа и тип значения в качестве аргументов шаблона. Например, следующее объявление создает объект `multimap`, в котором ключи имеют тип `int`, а сохраняемые значения — тип `string`:

```
multimap<int, string> codes;
```

Дополнительный, третий аргумент шаблона используется для указания функции сравнения или объекта, необходимого для упорядочения ключей. По умолчанию с типом ключа как параметр используется шаблон `less<>` (рассматривается далее). Более ранние реализации компилятора C++ могут требовать, чтобы этот параметр шаблона был указан явно.

Чтобы совместно сохранять информацию, реальный тип значений комбинирует тип ключа и тип данных в одиночную пару. Для этого в STL применяется класс шаблонов `pair<class T, class U>`, который сохраняет два типа значений в одном объекте. Если `keytype` — это тип ключа, а `datatype` — тип данных, тогда тип значения будет `pair<const keytype, datatype>`. Например, тип значения для объекта `codes`, объявленного ранее, — это `pair<const int, string>`.

Представьте себе, например, что нужно сохранить имена городов, используя код города в качестве ключа. Тогда можно удовлетворить объявления объекта `codes`, в котором тип `int` используется для ключей, а тип `string`

— для данных. Один из подходов состоит в том, чтобы создать пару элементов данных, а затем вставить ее:

```
pair<const int, string>
 item(213, "Los Angeles");
codes.insert(item);
```

Или же можно создать анонимный объект `pair` и вставить его в одном и том же выражении:

```
codes.insert(pair<const int, string>
 (213, "Los Angeles"));
```

Поскольку вхождения сортируются по ключам, нет необходимости в указании местоположения вставки.

Имея объект `pair`, можно получить доступ к двум его компонентам, воспользовавшись элементами `first` и `second`:

```
pair<const int, string>
 item(213, "Los Angeles");
cout << item.first << ' '
 << item.second << endl;
```

Следующий вопрос: как получить информацию об объекте `multimap`? Функция-элемент `count()` получает ключ в качестве аргумента и возвращает количество элементов, имеющих такой ключ. Существуют функции-элементы `lower_bound()` и `upper_bound()`, получающие в качестве аргумента ключ и работающие так же, как и для класса `set`. Имеется функция-элемент `equal_range()`, которая получает в качестве аргумента ключ и возвращает итераторы, указывающие на диапазон элементов с данным ключом. Чтобы возвратить два значения, этот метод объединяет их в объекте `pair`, причем в этом случае оба аргумента шаблона имеют тип итератора. Например, следующий код отображает список городов в объекте `codes`, у которых междугородний код равен 718:

```
pair<multimap<Keytype, string>::iterator,
 multimap<Keytype, string>::iterator>
range = codes.equal_range(718);
cout << "Cities with area code 718:\n";
for (it = range.first; it != range.second; ++it)
 cout << (*it).second << endl;
```

Листинг 15.11 демонстрирует большинство из этих методов. В нем также используется оператор `typedef` для упрощения работы с кодами.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Более ранние реализации компилятора C++ используют заголовочные файлы `multimap.h` и `algo.h`. Может потребоваться `less<Pair>` в качестве третьего аргумента шаблона для `multimap`. В более ранних версиях может использоваться `ostream_iterator<string>` вместо `ostream_iterator<string,char>`. В Borland's C++Builder 1.0 `const` в операторе `typedef` для объекта `Pair` не должен указываться.

## Листинг 15.11 Программа multimap.cpp.

```

// multimap.cpp - использование класса multimap
#include <iostream>
using namespace std;
#include <string>
#include <map>
#include <algorithm>

typedef int KeyType;
typedef pair<const KeyType, string> Pair;
typedef multimap<KeyType, string> MapCode;

int main()
{
 MapCode codes;

 codes.insert(Pair(415, "San Francisco"));
 codes.insert(Pair(510, "Oakland"));
 codes.insert(Pair(718, "Brooklyn"));
 codes.insert(Pair(718, "Staten Island"));
 codes.insert(Pair(415, "San Rafael"));
 codes.insert(Pair(510, "Berkeley"));

 cout << "Number of cities with area code 415: " << codes.count(415) << endl;
 cout << "Number of cities with area code 718: " << codes.count(718) << endl;
 cout << "Number of cities with area code 510: " << codes.count(510) << endl;
 cout << "Area Code City\n";

 MapCode::iterator it;
 for (it = codes.begin(); it != codes.end(); ++it)
 cout << " " << (*it).first << " " << (*it).second << endl;

 pair<MapCode::iterator, MapCode::iterator> range = codes.equal_range(718);
 cout << "Cities with area code 718:\n";
 for (it = range.first; it != range.second; ++it)
 cout << " " << endl;

 return 0;
}

```

Результаты выполнения программы из листинга 15.11:

```

Number of cities with area code 415: 2
Number of cities with area code 718: 2
Number of cities with area code 510: 2
Area Code City
 415 San Francisco
 415 San Rafael
 510 Oakland
 510 Berkeley
 718 Brooklyn
 718 Staten Island
Cities with area code 718:
Brooklyn
Staten Island

```

## Функциональные объекты (функторы)

Многие алгоритмы STL используют функциональные объекты (функторы). Функтор — это любой объект, который можно использовать с обозначением () так, как используется обычная функция. Он включает в себя обычные имена функций, указатели на функции и объекты класса, для которых определена имеющая

странный вид функция operator(). Например, класс можно определить так:

```

class Linear
{
private:
 double slope;
 double y0;
public:
 Linear(double _s1 = 1, double _y = 0)
 : slope(_s1), y0(_y) { }
 double operator()(double x)
 { return y0 + slope * x; }
};

```

Тогда объекты Linear можно использовать как функции:

```

Linear f1;
Linear f2(2.5, 10.0);
double y1 = f1(12.5); // rhs - это // f1.operator()(12.5)
double y2 = f2(0.4);

```

Помните про функцию for\_each? Она выполняет заданную функцию для каждого элемента диапазона:

```
for_each(books.begin(), books.end(), ShowReview);
```

Вообще, третий аргумент может быть функтором, а не только обычной функцией. При этом возникает вопрос: как объявлен третий аргумент? Его нельзя объявить как указатель на функцию, поскольку такой указатель определяет тип аргумента. Поскольку контейнер может содержать любой тип, наперед неизвестно, какой именно аргумент нужно применить. В STL эта проблема решается путем использования шаблонов. Прототип функции `for_each` выглядит примерно так:

```
template<class InputIterator, class Function>
Function for_each(InputIterator first,
 InputIterator last,
 Function f);
```

Прототип функции `ShowReview()` был определен так:

```
void ShowReview(const Review &);
```

Тогда идентификатор `ShowReview` имеет тип `void (*)(const Review &)`, поэтому такой же тип присваивается аргументу шаблона `Function`.

## Концепции функторов

Подобно тому как в STL определены концепции для контейнеров и итераторов, определена и концепция функтора:

- *Генератор* — это функтор, который вызывается без аргументов.
- *Унарная функция* — это функтор, имеющий один аргумент.
- *Бинарная функция* — это функтор, имеющий два аргумента.

Например, функтор, необходимый для функции `for_each()`, должен быть унарной функцией, поскольку он выполняется для одного элемента контейнера.

Конечно же данные концепции имеют уточнения:

- Унарная функция, которая возвращает значение типа `bool`, является *предикатом*.
- Бинарная функция, которая возвращает значение типа `bool`, является *бинарным предикатом*.

Несколько функций STL требуют использования предикатов или бинарных предикатов в качестве аргументов. Например, в листинге 15.6 применяется версия функции `sort()`, которая получает бинарный предикат в своем третьем аргументе:

```
bool WorseThan(const Review & r1,
 const Review & r2);
...
sort(books.begin(), books.end(), WorseThan);
```

Класс шаблонов `list` имеет функцию-элемент `remove_if()`, аргументом которой является предикат.

Функция выполняет предикат для каждого элемента заданного диапазона, удаляя те элементы, для которых предикат возвращает значение "истина". Например, следующий код удалит все элементы, которые больше 100, из списка `three`:

```
bool tooBig(int n){ return n > 100; }
list<int> scores;
...
scores.remove_if(tooBig);
```

Кстати, данный пример показывает явно, где может быть полезен класс-функтор. Представьте, что нужно удалить все значения, превышающие 200, из другого списка. Было бы удобно передать это значение функции `tooBig()` во втором аргументе и, таким образом, использовать функцию с различными значениями, но предикат может иметь только один аргумент. Однако, если создать класс `TooBig`, можно воспользоваться элементами класса вместо аргументов функции, чтобы передать дополнительную информацию:

```
template<class T>
class TooBig
{
private:
 T cutoff;
public:
 TooBig(const T & t) : cutoff(t) { }
 bool operator()(const T & v)
 { return v > cutoff; }
};
```

Здесь одно значение (`v`) передается как аргумент функции, тогда как второй аргумент (`cutoff`) устанавливается с помощью конструктора класса. Имея такое определение, можно инициализировать различные объекты `TooBig` различными значениями для удаления элементов:

```
TooBig<int> f100(100);
list<int> froobies;
list<int> scores;
...
froobies.remove_if(f100); //использование
 //именованного функционального объекта
scores.remove_if(TooBig<int>(200));
 //конструктор функционального объекта
```

 **ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**

Метод `remove_if()` является шаблоном метода в шаблоне класса. Шаблоны методов — это достаточно новые дополнения к свойствам шаблонов C++ (появившиеся по той причине, что они требуются в STL), и большинство компиляторов ко времени написания данной книги еще не реализует их. Однако существует также и функция `remove_if()`, не являющаяся элементом класса, которая имеет два аргумента, задающих диапазон (два итератора), и предикат в качестве третьего аргумента.

Предположим, что уже есть шаблон функции с двумя аргументами:

```
template <class T>
bool tooBig(const T & val, const T & lim)
{
 return val > lim;
}
```

Можно воспользоваться классом, чтобы преобразовать функцию в функциональный объект с одним аргументом:

```
template<class T>
class TooBig2
{
private:
 T cutoff;
public:
 TooBig2(const T & t) : cutoff(t) { }
 bool operator()(const T & v)
 { return tooBig<T>(v, cutoff); }
};
```

Иначе говоря, можно сделать следующее:

```
TooBig2<int> tB100(100);
int x;
cin >> x;
if (tB100(x)) // то же самое,
 // что и if (tooBig(x, 100))
...
```

Таким образом, вызов функции `tB100(x)` — это то же самое, что и вызов функции `tooBig(x, 100)`. Но в данном случае функция двух аргументов заменена функциональным объектом с одним аргументом, где второй аргумент используется при создании функционального объекта его конструктором. Другими словами, класс-функтор `TooBig2` — это функциональный адаптер, который адаптирует функцию в соответствии с определенным интерфейсом.

## Предопределенные функторы

В STL определено несколько элементарных функциональных объектов, которые выполняют такие операции, как сложение двух значений, проверка равенства двух значений и т.д. Они необходимы для тех функций STL, которые используют функции в качестве аргументов. Например, рассмотрим функцию `transform()`, имеющую две версии. Первая версия имеет четыре аргумента. Первые два — это итераторы, определяющие диапазон в контейнере. (Теперь вы уже знакомы с таким подходом.) Третий аргумент — итератор, задающий место копирования результата. Последний, четвертый аргумент — это функтор, который выполняется для каждого элемента диапазона, чтобы в результате получить новый элемент. Например, рассмотрим следующее:

```
const int LIM = 5;
double arr1[LIM] = { 36, 39, 42, 45, 48 };
vector<double> gr8(arr1, arr1 + LIM);
ostream_iterator<double, char> out(cout, " ");
transform(gr8.begin(), gr8.end(), out, sqrt);
```

Данный программный код вычисляет квадратный корень для каждого элемента и посыпает результат в поток вывода. Целевой итератор может принимать значения и в исходном диапазоне. Например, замена `out` на `gr8.begin()` в рассматриваемом примере приведет к тому, что новые значения будут просто скопированы поверх старых. Ясно, что используемый функтор должен иметь один аргумент.

Вторая версия использует функцию, имеющую два аргумента, и выполняет ее для каждого элемента из двух диапазонов. Эта версия имеет дополнительный аргумент, третий по счету, задающий начало второго диапазона. Например, если `m8` — это второй объект типа `vector <double>` и функция `mean(double, double)` возвращает среднее из двух значений, то следующая программа выведет среднее значение для каждой пары элементов из контейнеров `gr8` и `m8`:

```
transform(gr8.begin(), gr8.end(),
 m8.begin(), out, mean);
```

Предположим, что нужно сложить два массива. Операцию `+` нельзя использовать в качестве аргумента, поскольку для типа `double` `+` является встроенной операцией, а не функцией. В таком случае можно определить функцию для сложения двух чисел и применить ее так:

```
double add(double x, double y)
{
 return x + y;
}
transform(gr8.begin(), gr8.end(),
 m8.begin(), out, add);
```

Но тогда для каждого типа нужно определить отдельную функцию. Было бы лучше определить шаблон, но этого не нужно делать, поскольку он уже определен в STL. Заголовочный файл `functional` (ранее `function.h`) определяет несколько шаблонов классов функциональных объектов.

Использование класса `plus<>` для обычного сложения также возможно, хотя и неудобно:

```
#include <functional>
...
// создать объект класса plus<double>
plus<double> add;
double y = add(2.2, 3.4); // использование
 // plus<double>::operator()()
```

Проще передавать функциональный объект в качестве аргумента:

```
transform(gr8.begin(), gr8.end(),
 m8.begin(), out, plus<double>());
```

Здесь, вместо того чтобы создавать именованный объект, с помощью конструктора класса `plus<double>` создается функциональный объект, выполняющий сложение. (Скобки задают вызов конструктора по умолчанию.)

нию; то, что передается функции `transform()`, — это созданный конструктором функциональный объект.)

В библиотеке STL определены функциональные объекты, являющиеся эквивалентами всех встроенных знаков арифметических операций, операторов сравнения и логических операторов. В табл. 15.11 приведены имена этих функторов-эквивалентов. Их можно использовать вместе со встроенными типами данных C++ или же с любыми определенными пользователем типами, которые перегружают соответствующий оператор.

### ПРЕДОСТЕРЕЖЕНИЕ

В более ранних реализациях используется имя `times` вместо `multiples`.

Таблица 15.11 Операторы и эквивалентные им функциональные объекты.

| Оператор | Эквивалентный функциональный объект |
|----------|-------------------------------------|
| +        | <code>plus</code>                   |
| -        | <code>minus</code>                  |
| *        | <code>multiples</code>              |
| /        | <code>divides</code>                |
| %        | <code>modulus</code>                |
| -        | <code>negate</code>                 |
| ==       | <code>equal_to</code>               |
| !=       | <code>not_equal_to</code>           |
| >        | <code>greater</code>                |
| <        | <code>less</code>                   |
| >=       | <code>greater_equal</code>          |
| <=       | <code>less_equal</code>             |
| &&       | <code>logical_and</code>            |
|          | <code>logical_or</code>             |
| !        | <code>logical_not</code>            |

## Адаптируемые функторы и функции-адаптеры

Все предопределенные функторы в табл. 15.11 являются *адаптируемыми*. Фактически STL имеет пять связанных между собой концепций, таких как: *адаптируемый генератор*, *адаптируемая унарная функция*, *адаптируемая бинарная функция*, *адаптируемый предикат* и *адаптируемый бинарный предикат*.

То, что делает объект-функтор адаптируемым, — это его свойство задавать элементы оператором `typedef`, которые определяют тип аргументов функтора и возвращаемое значение. Элементы называются `result_type`,

`first_argument_type` и `second_argument_type`; они представляют тип результата, тип первого аргумента и тип второго аргумента соответственно. Например, возвращаемое значение объекта `plus<int>` имеет тип `plus<int>::result_type`.

Важность адаптируемости функционального объекта состоит в том, что его могут использовать объекты функций-адаптеров, которые предполагают существование элементов, заданных оператором `typedef`. Например, функция с аргументом, который является адаптируемым функтором, может использовать элемент `result_type` для объявления переменной, которая имеет тип значения, возвращаемый функтором.

Кроме того, в STL имеются классы функций-адаптеров, которые используют эти свойства. Например, предположим, что каждый элемент вектора `gr8` нужно умножить на 2.5. Для этого нужно воспользоваться версией функции `transform()` с унарной функцией-аргументом:

```
transform(gr8.begin(), gr8.end(), out, sqrt);
```

в примере, показанном ранее. Функтор `multiples()` может выполнять умножение, но это бинарная функция. Поэтому требуется функция-адаптер, которая заменит функтор с двумя аргументами на функтор с одним аргументом. В примере с функцией `TooBig2`, рассмотренным ранее, был показан один из способов достижения этой цели, но в STL этот процесс автоматизирован с помощью классов `binder1st` и `binder2nd`, преобразующих адаптируемые бинарные функции в адаптируемые унарные функции.

Сначала рассмотрим `binder1st`. Предположим, что имеется адаптируемый бинарный функциональный объект `f2()`. Можно создать объект `binder1st`, который заставляет использовать определенное значение, скажем `val`, в качестве первого аргумента функции `f2()`:

```
binder1st(f2, val) f1;
```

Тогда в результате выполнения функции `f1(x)` с одним аргументом возвращается такое же значение, как и в результате выполнения функции `f2(x, val)`, где `val` является первым аргументом, а аргумент функции `f1()` — вторым. Иначе говоря, `f1(x)` эквивалентна `f2(val, x)` с той лишь разницей, что вместо бинарной функции используется унарная. Таким образом, функция `f2()` — адаптируемая. Опять же, все это возможно только в том случае, если `f2()` — адаптируемая функция.

Это может показаться несколько запутанным. Однако в STL имеется функция `binder1st()`, упрощающая применение класса `binder1st`. Ей передается имя и значение, используемое конструктором класса `binder1st`, и она возвращает объект этого класса. Например, можно конвер-

тировать бинарную функцию `multiples()` в унарную функцию, которая умножает свой аргумент на 2.5. Это просто:

```
bind1st(multiples<double>(), 2.5)
```

В итоге умножение каждого элемента контейнера `gr8` на 2.5 и вывод результатов выполняется так:

```
transform(gr8.begin(), gr8.end(), out,
 bind1st(multiples<double>(), 2.5));
```

Класс `binder2nd` организован так же, но он присваивает постоянное значение второму аргументу, а не первому. Он имеет вспомогательную функцию `bind2nd`, аналогичную функции `bind1st`.



Если функция STL вызывает унарную функцию и является адаптируемой бинарной функцией, выполняющей необходимую задачу, можно воспользоваться функцией `bind1st()` или `bind2nd()`, чтобы адаптировать бинарную функцию к унарному интерфейсу.

Листинг 15.12 объединяет некоторые из приведенных примеров в короткую программу.

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В более ранних реализациях используются заголовочные файлы `vector.h`, `iterator.h`, `algo.h` и `function.h`. Кроме того, вместо `multiples` может применяться имя `times`.

Результаты выполнения программы из листинга 15.12:

```
36 39 42 45 48
25 27 29 31 33
61 66 71 76 81
90 97.5 105 112.5 120
```

## Алгоритмы

Библиотека STL содержит большое количество функций, не являющихся элементами каких-либо классов, для работы с контейнерами. Здесь вы уже сталкивались с некоторыми из них: `sort()`, `copy()`, `find()`, `for_each()`, `random_shuffle()`, `set_union()`, `set_intersection()`, `set_difference()` и `transform()`. Вероятно, вы заметили, что они созданы единообразно, используют итераторы для задания диапазонов обрабатываемых данных и определения, куда нужно направлять результаты. Некоторые из них используют функциональный объект в качестве аргумента для обработки данных.

Структура алгоритмов функций имеет два основных компонента. Во-первых, алгоритмы используют шаблоны, чтобы обеспечить общие типы данных. Во-вторых, они используют итераторы, чтобы представление доступа к данным в контейнере носило наиболее общий характер. Таким образом, функция `copy()` может работать с контейнером, содержащим значения типа `double` в массиве, с контейнером, содержащим значения типа `string` в связанным списке, или с контейнером, содержащим

### Листинг 15.12 Программа funadap.cpp.

```
// funadap.cpp - использование функций-адаптеров
#include <iostream>
using namespace std;
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>

const int LIM = 5;
int main()
{
 double arr1[LIM] = { 36, 39, 42, 45, 48 };
 double arr2[LIM] = { 25, 27, 29, 31, 33 };
 vector<double> gr8(arr1, arr1 + LIM);
 vector<double> m8(arr2, arr2 + LIM);
 ostream_iterator<double, char> out(cout, " ");
 copy(gr8.begin(), gr8.end(), out);
 cout << endl;
 copy(m8.begin(), m8.end(), out);
 cout << endl;

 transform(gr8.begin(), gr8.end(), m8.begin(), out, plus<double>());
 cout << endl;

 transform(gr8.begin(), gr8.end(), out, bind1st(multiples<double>(), 2.5));
 cout << endl;

 return 0;
}
```

объекты, определенные пользователем, в древовидной структуре, как это делается в классе `set`. Поскольку указатели являются специальными типами итераторов, такие функции STL, как `copy()`, можно использовать с обычными массивами.

Класс `string`, хотя он и не является частью STL, разработан с учетом требований STL. Например, он включает методы `start()` и `end()`. Таким образом, он может применять интерфейс STL.

Универсальный дизайн контейнеров позволяет создавать удобные связи между контейнерами различных типов. Например, можно воспользоваться функцией `copy()` для копирования значений из обычного массива в объект класса `vector`, из объекта класса `vector` в объект класса `list` и из объекта класса `list` в объект класса `set`. Можно использовать оператор присваивания `==` для сравнения различных типов контейнеров, например, `deque` и `vector`. Это возможно, потому что перегруженный оператор `==` для контейнеров использует итераторы для сравнения их содержимого, поэтому объект класса `deque` и объект класса `vector` считаются равными, если они имеют одинаковое содержимое, расположенные в одинаковом порядке.

## Группы алгоритмов

В STL библиотека алгоритмов разделена на четыре группы:

- Операции, не изменяющие последовательность
- Операции, изменяющие последовательность
- Операции сортировки и подобные им
- Обобщенные числовые операции

Три первые группы объявлены в заголовочном файле `algorithm` (ранее `algo.h`), а четвертая группа, специально ориентированная на работу с числовыми данными, имеет свой собственный заголовочный файл, названный `numeric`. (Ранее они также находились в файле `algol.h`.)

Операции, не изменяющие последовательность, выполняются над каждым элементом в диапазоне. Эти операции оставляют контейнер неизменным. Например, функции `find()` и `for_each()` принадлежат именно к этой категории.

Операции, изменяющие последовательность, также выполняются над каждым элементом в диапазоне. Однако, как подразумевает название, они изменяют содержимое контейнера. Изменениям могут подвергнуться значения или же порядок, в котором они сохранены. Например, функции `transform()`, `random_shuffle()` и `copy()` попадают в данную категорию.

Сортировка и связанные с ней операции включают несколько функций сортировки (в том числе `sort()`) и набор других функций, включая операции работы с множествами.

Числовые операции включают функции суммирования содержимого диапазона, подсчета внутреннего произведения двух контейнеров, частичных сумм и разностей соседних элементов. Обычно эти операции характерны для массивов, поэтому класс `vector` — это контейнер, часто используемый с ними.

В приложении G приведен полный список этих функций.

## Общие свойства

Как вы убедились, функции STL работают с итераторами и диапазонами итераторов. Прототип функции показывает, какие предположения об итераторах применяются в этой функции. Например, функция `copy()` имеет такой прототип:

```
template<class InputIterator,
 class OutputIterator>
OutputIterator copy(InputIterator first,
 InputIterator last,
 OutputIterator result);
```

Поскольку идентификаторы `InputIterator` и `OutputIterator` являются параметрами шаблона, их можно было назвать просто `T` и `U`. Однако в документации по библиотеке STL используются имена параметров шаблонов, чтобы указать, какую именно концепцию моделируют данные параметры. Поэтому такое объявление демонстрирует, что параметры диапазона должны быть итераторами ввода, а итератор, указывающий, куда направить результат, должен быть итератором вывода.

Один из способов классификации алгоритмов базируется на определении того, куда помещается результат выполнения алгоритма. Одни алгоритмы выполняются сразу над контейнерами, другие создают копии. Например, когда функция `sort()` завершается, результат находится там же, где были исходные данные. Поэтому `sort()` является *местным алгоритмом*. Однако функция `copy()` размещает результат своего выполнения в другом месте, поэтому `copy()` является *копирующим алгоритмом*. Функция `transform()` может выполняться по-разному. Как и `copy()`, она использует итератор вывода для указания того, куда поместить результат. В отличие от `copy()`, функция `transform()` позволяет итератору вывода указывать на входной диапазон, поэтому она может копировать преобразованные значения поверх исходных.

Некоторые алгоритмы имеют две версии: местную и копирующую. Соглашение STL: добавлять `_copy` к имени копирующей версии. Эта версия имеет дополнитель-

ный параметр — итератор вывода, указывающий, куда нужно скопировать вывод. Например, функция `replace()` имеет такой прототип:

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first,
 ForwardIterator last,
 const T& old_value,
 const T& new_value);
```

Она заменяет каждое вхождение `old_value` новым значением `new_value`. Это происходит на местном уровне. Поскольку этот алгоритм считывает, и записывает данные в элементы контейнера, тип итератора должен быть не менее чем прямым итератором (`ForwardIterator`). Копирующая версия имеет такой прототип:

```
template<class InputIterator,
 class OutputIterator, class T>
OutputIterator replace_copy(
 InputIterator first,
 InputIterator last,
 OutputIterator result,
 const T& old_value,
 const T& new_value);
```

На этот раз результирующие данные копируются туда, куда указывает `result`, поэтому итератора ввода вполне достаточно для задания диапазона.

Заметьте, что функция `replace_copy()` возвращает значение типа `OutputIterator` (итератор вывода). Соглашение для копирующих алгоритмов состоит в том, что они возвращают итератор, указывающий на элемент, находящийся за последним скопированным элементом.

Другая распространенная вариация — некоторые функции имеют версию, которая выполняется по-разному, в зависимости от результата применения функции к каждому элементу контейнера. Такие версии обычно добавляют `_if` к имени функции. Например, функция `replace_if()` заменяет старое значение новым только в том случае, если при выполнении функции над старым значением возвращается результат "истина". Вот ее прототип:

```
template<class ForwardIterator,
 class Predicate class T>
void replace_if(ForwardIterator first,
 ForwardIterator last,
 Predicate pred,
 const T& new_value);
```

Вспомните, что предикат — это имя унарной функции, возвращающей значение типа `bool`. Существует также версия `replace_copy_if()`. Вы можете догадаться сами, что именно она выполняет и какой имеет прототип.

Как и в случае с `InputIterator`, можно сказать, что `Predicate` — это имя параметра шаблона, которое можно с легкостью заменить на `T` или `U`. Однако в STL при-

меняют имя `Predicate`, чтобы напомнить пользователю, что реальный аргумент должен быть моделью концепции `Predicate`. Аналогично в STL используются такие термины, как `Generator` и `BinaryPredicate`, для идентификации аргументов, которые моделируют другие концепции функциональных объектов.

## Использование STL

STL — это библиотека, части которой предназначены для совместной работы. Компоненты STL являются не просто инструментами, но и строительными блоками для создания других инструментов. Проиллюстрируем это примером. Предположим, что нужно написать программу, которая позволяет пользователю вводить слова. В конце необходимо сохранить список слов в порядке их ввода, список использованных слов в алфавитном порядке (регистр букв игнорируется), а также то, сколько раз вводилось каждое слово. Для простоты будем считать, что ввод не содержит цифр или символов пунктуации.

Ввод и сохранение списка слов достаточно просты. Следуя примеру из листинга 15.5, можно создать объект `vector<string>` и воспользоваться функцией `push_back()` для добавления вводимых слов к вектору:

```
vector<string> words;
string input;
while (cin >> input && input != "quit")
 words.push_back(input);
```

Как теперь получить список слов в алфавитном порядке? Можно использовать функцию `sort()`, а затем `unique()`, но при таком подходе будут перезаписаны исходные данные, поскольку `sort()` является местным алгоритмом. Существует более простой способ, позволяющий избежать этих проблем. Он состоит в том, чтобы создать объект `set<string>` и копировать (используя итератор вставки) слова из вектора в множество. Множество автоматически сортируется с помощью функции `sort()`, а поскольку множество позволяет иметь только одну копию ключа элемента, вызывается функция `unique()`. Стоп! Одна из спецификаций программы требует, чтобы регистр букв игнорировался. Один из способов, позволяющий учесть это, заключается в использовании функции `transform()` вместо `copy()` для копирования данных из вектора в множество. Для преобразования нужно использовать функцию, которая конвертирует все буквы строки в строчные.

```
set<string> wordset;
transform(words.begin(), words.end(),
 insert_iterator<set<string> >
 (wordset, wordset.begin()), ToLower);
```

Функцию `ToLower()` легко написать. Нужно просто использовать функцию `transform()`, чтобы выполнить функцию `tolower()` для каждого элемента в строке, при-

чем строка является и источником, и приемником данных. Вспомните, объекты класса `string` также могут использовать функции STL. Передача и возвращение строки указателем означает, что алгоритм работает с исходной строкой, не создавая копий.

```
string & ToLower(string & st)
{
 transform(st.begin(), st.end(),
 st.begin(), tolower);
 return st;
}
```

Для подсчета того, сколько раз слово встретилось при вводе, можно воспользоваться функцией `count()`. Она получает диапазон и значение в качестве аргументов и возвращает величину, определяющую, сколько раз данное значение встретилось в диапазоне. Можно использовать объект `vector` для обеспечения диапазона и объект `set` для обеспечения списка подсчитываемых слов. Иначе говоря, для каждого слова в множестве нужно подсчитать, сколько раз оно появляется в векторе. Чтобы результат подсчета был ассоциирован с нужным словом, необходимо просто сохранить слово и счетчик как объект `pair<const string, int>` в объекте `map`. Слово будет ключом (только одна копия), а счетчик — значением. Это можно сделать в одном цикле:

```
map<string, int> wordmap;
set<string>::iterator si;
for (si = wordset.begin();
 si != wordset.end(); si++)
 wordmap.insert(pair<string, int>
 (*si, count(words.begin(),
 words.end(), *si)));
```



### ПРЕДОСТЕРЕЖЕНИЕ

Более ранние реализации STL объявляют тип функции `count()` как `void`. Вместо возвращаемого значения применяется четвертый аргумент, который передается указателем, и подсчитанное количество слов добавляется к этому аргументу:

```
int ct = 0;
count(words.begin(),
 words.end(), *si), ct); // count
 // добавляется к ct
```

Класс `map` имеет интересное свойство — для него можно применять форму записи массива, причем индексом для доступа к значению служит ключ. Например, `wordmap["the"]` представляет собой значение, поставленное в соответствие ключу "the", в данном случае оно равно количеству вхождений строки "the". Поскольку в контейнере `wordset` содержатся все ключи, используемые контейнером `wordmap`, можно применить следующий код как альтернативный и более удобный способ сохранения результатов:

```
for (si = wordset.begin();
 si != wordset.end(); si++)
 wordmap[*si] = count(words.begin(),
 words.end(), *si);
```

Поскольку `si` указывает на строку в контейнере `wordset`, `*si` является строкой и может служить ключом для контейнера `wordmap`. Данный код помещает ключи и значения в `wordmap`.

Аналогично можно воспользоваться записью массива для вывода результатов:

```
for (si = wordset.begin();
 si != wordset.end(); si++)
 cout << *si << ":" << wordmap[*si]
 << endl;
```

Если ключ задан неверно, то отвечающее ему значение равно 0.

В листинге 15.13 демонстрируются все эти три идеи и показан код для вывода содержимого трех контейнеров (`vector` — с введенными словами, `set` — со списком слов и `map` — с подсчитанным количеством слов).

 **ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**

Более ранние реализации могут использовать заголовочные файлы `vector.h`, `set.h`, `map.h`, `iterator.h`, `algo.h` и `stl_type.h`. Кроме того, может потребоваться, чтобы в шаблонах `set` и `map` использовался дополнительный параметр шаблона `less<string>`. Может также использоваться `ostream_iterator<string>` вместо `ostream_iterator<string,char>`. Более ранние версии используют тип `Void` для функции `count()`, как упоминалось ранее.

Результаты выполнения программы из листинга 15.13:

```
Enter words (enter quit to quit):
The dog saw the cat and thought the cat fat
The cat thought the cat perfect
quit
```

```
You entered the following words:
The dog saw the cat and thought the cat fat
The cat thought the cat perfect
```

```
Alphabetic list of words:
and cat dog fat perfect saw that the thought
```

```
Word frequency:
and: 1
cat: 4
dog: 1
fat: 1
perfect: 1
saw: 1
that: 1
the: 4
thought: 2
```

Мораль здесь в том, что при использовании STL нужно рассуждать так: сколько усилий, затрачиваемых на создание кода, я смогу сэкономить? Общий и гибкий дизайн STL позволяет избежать выполнения огромного объема работы. Кроме того, разработчики STL по заботились о высокой эффективности библиотеки. Поэтому алгоритмы STL хорошо подобраны и реализованы максимально эффективно.

**Листинг 15.13 Программа usealgo.cpp.**

```
//usealgo.cpp
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <map>
#include <iterator>
#include <algorithm>
#include <cctype>
using namespace std;

string & ToLower(string & st);

int main()
{
 vector<string> words;
 cout << "Enter words (enter quit to quit):\n";
 string input;
 while (cin >> input && input != "quit")
 words.push_back(input);

 cout << "You entered the following words:\n";
 ostream_iterator<string,char> out(cout, " ");
 copy(words.begin(), words.end(), out);
 cout << endl;

 // разместить слова в контейнер set, ПРЕОБРАЗУЯ все буквы в строчные
 set<string> wordset;
 transform(words.begin(), words.end(),
 insert_iterator<set<string> > (wordset, wordset.begin()),
 ToLower);
 cout << "\nAlphabetic list of words:\n";
 copy(wordset.begin(), wordset.end(), out);
 cout << endl;

 // разместить слова и количество их вхождений в map
 map<string, int> wordmap;
 set<string>::iterator si;
 for (si = wordset.begin(); si != wordset.end(); si++)
 wordmap[*si] = count(words.begin(), words.end(), *si);

 // вывести содержимое контейнера map
 cout << "\nWord frequency:\n";
 for (si = wordset.begin(); si != wordset.end(); si++)
 cout << *si << ": " << wordmap[*si] << endl;

 return 0;
}

string & ToLower(string & st)
{
 transform(st.begin(), st.end(), st.begin(), tolower);
 return st;
}
```

## Другие библиотеки

C++ имеет еще несколько библиотек классов, более специализированных, чем примеры, рассмотренные в данной главе. Заголовочный файл `complex` содержит шаблон класса `complex` для работы с комплексными числами, со специализациями типов `float`, `long` и `long double`. Класс обеспечивает стандартные операции с комплексными числами, а также стандартные функции, которые можно выполнять и над комплексными числами.

Заголовочный файл `valarray` содержит шаблон класса `valarray`. Данный шаблон класса разработан для поддержки числовых массивов и обеспечивает множество различных операций, в том числе сложение содержимого двух массивов, выполнение математических функций над каждым элементом массива и выполнение операций линейной алгебры над массивами.

## Резюме

C++ имеет мощный набор библиотек, которые позволяют решить многие типичные задачи программирования, а также инструменты для упрощения решения таких задач. Класс `string` имеет удобные методы, позволяющие работать со строками как с объектами. Класс поддерживает автоматическое управление памятью и множество методов и функций для работы со строками. Некоторые из них, например, позволяют конкатенировать строки, вставлять одну строку в другую, обращать строку, производить поиск символа или подстроки в строке, а также производить операции ввода/вывода.

Шаблон класса `auto_ptr` упрощает управление памятью, выделяемой оператором `new`. При использовании объекта класса `auto_ptr` вместо обычного указателя, содержащего адрес, возвращаемый командой `new`, нет необходимости помнить о последующем применении оператора `delete`. Когда объект `auto_ptr` заканчивает свою работу, его деструктор вызывается оператор `delete` автоматически.

Стандартная библиотека шаблонов, или STL, — это набор шаблонов классов-контейнеров, классов-итераторов, функциональных объектов и алгоритмических функций, представляющих унифицированный дизайн, базирующийся на принципах обобщенного программирования. Алгоритмы используют шаблоны, что делает их универсальными независимо от типа сохраняемого объекта или интерфейса итератора, используемого в контейнере. Итераторы являются обобщением указателей.

В STL используется термин "концепция" для обозначения набора требований. Например, концепция прямого итератора включает в себя требования, состоящие в том, чтобы прямой итератор мог быть разъме-

нован для чтения и записи, а его значение могло быть увеличено. Фактическая реализация концепции называется моделью концепции. Например, концепцию прямого итератора может моделировать обычный указатель или объект, созданный для перемещения по связному списку. Концепции, базирующиеся на других концепциях, называются уточнениями. Например, двусторонний итератор — это уточнение концепции прямого итератора.

Классы-контейнеры `vector` и `set` являются моделями концепций контейнеров, таких как контейнер, последовательность и ассоциативный контейнер. В STL определены несколько шаблонов классов-контейнеров: `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap` и `bitset`. Кроме того, определены классы адаптеров шаблонов: `queue`, `priority_queue` и `stack`. Эти классы адаптируют исходный класс-контейнер, добавляя ему характеристики интерфейса одного из данных классов. Поэтому `stack`, хотя он и базируется на классе `vector`, позволяет выполнить вставку и удаление элементов только из вершины стека.

Некоторые алгоритмы реализованы как методы классов, но большая часть из них реализована как общие, не являющиеся методами классов функции. Это возможно благодаря использованию итераторов как интерфейса между контейнерами и алгоритмами. Одним из преимуществ такого подхода является отсутствие необходимости задавать несколько функций `for_each()` и `copy()` и т.д. для каждого типа контейнера. Еще одно преимущество состоит в том, что алгоритмы STL можно использовать с внешними для STL контейнерами, такими как, например, обычные массивы, объекты класса `string` и любые другие созданные пользователем классы, находящиеся в соответствии с идиомами STL для итератора и контейнера.

И контейнеры, и алгоритмы характеризуются типом итератора, который они используют. Необходимо убедиться в том, что контейнер поддерживает такую концепцию итератора, которая отвечает потребностям алгоритма. Например, алгоритм `for_each()` использует итератор ввода, минимальным требованиям которого удовлетворяют все типы классов-контейнеров в STL. А алгоритм `sort()` для работы требует наличия итератора произвольного доступа, который поддерживает не все классы-контейнеры. Класс-контейнер может содержать специализированный метод дополнительно, если он не удовлетворяет требованиям определенного алгоритма. Например, класс `list` имеет метод `sort()`, базирующийся на двустороннем итераторе, поэтому его можно использовать вместо общей функции.

В STL также содержатся функциональные объекты, или функторы, которые представляют собой классы с перегруженным оператором `()`, т.е. определенным мето-

дом `operator()`). Объекты таких классов можно использовать в функциональной записи, но, кроме этого, они могут нести и добавочную информацию. Например, адаптируемые функциональные объекты в выражении с оператором `typedef` определяют типы аргументов и тип возвращаемого значения для функционального объекта. Этую информацию могут использовать другие компоненты, например, функции-адаптеры.

Библиотека STL, содержащая общие типы контейнеров и множество операций над ними, реализованных эффективными алгоритмами, спроектирована в общей манере и является идеальным источником повторно используемого кода. Задачи программирования можно решить непосредственно с помощью инструментов STL или же используя их как строительные блоки, необходимые для создания решения.

Классы шаблонов `complex` и `valarray` поддерживают числовые операции над комплексными числами и массивами.

## Вопросы для повторения

- Рассмотрите следующее объявление класса:

```
class RQ1
{
private:
 char * st; // указывает на С-строку
public:
 RQ1() {st = new char [1]; strcpy(st,""); }
 RQ1(const char * s)
 { st = new char [strlen(s) + 1];
 strcpy(st, s); }
 RQ1(const RQ1 & rq);
 { st = new char [strlen(rq.st) + 1];
 strcpy(st, rq.st); }
 ~RQ1() { delete [] st; }
 RQ & operator=(const RQ & rq);
 // и т.д.
};
```

- Измените его так, чтобы в объявлении использовался объект класса `string`. Какие методы больше не требуют явного определения?
- Назовите хотя бы два преимущества, которые имеют объекты класса `string` над С-строками с точки зрения простоты использования.
- Создайте функцию, которая получает указатель на строку в качестве аргумента и конвертирует все символы в прописные.
- Какие из приведенных ниже примеров представляют собой некорректное (концептуально или синтаксически) использование объектов `auto_ptr`? (Считайте, что необходимые заголовочные файлы включены.)

```
auto_ptr<int> pia(new int[20]);
auto_ptr<str> (new string);
int rigue = 7;
auto_ptr<int>pr(&rigue);
auto_ptr dbl (new double);
```

- Если бы можно было создать механический эквивалент стека, который содержал бы клюшки для гольфа вместо чисел, почему он был бы плохой (концептуально) сумкой для инвентаря гольфа?
- Почему контейнер `set` был бы плохим выбором для сохранения записей об очках, набранных от лунки к лунке при игре в гольф?
- Если указатели являются итераторами, почему разработчики STL просто не использовали их вместо итераторов?
- Почему разработчики STL не определили просто базовый класс итераторов, использовали метод наследования для производных классов и выразили алгоритмы, используя термины таких классов итераторов?
- Приведите хотя бы три примера преимуществ, которые предоставляет объект класса `vector` по сравнению с обычным массивом.
- Если бы листинг 15.6 был реализован с применением класса `list` вместо класса `vector`, какие части программы стали бы непригодными? Можно ли было бы их легко исправить? Если да, то как?

## Упражнения по программированию

- Палиндром — это строка, читаемая одинаково в обоих направлениях. Например, "tot" и "otto" являются короткими палиндромами. Напишите программу, позволяющую пользователю ввести строку и передающую указатель на строку функции типа `bool`. Функция должна возвращать значение `true`, если строка является палиндромом, и `false` — если нет. В данном случае не заботьтесь о таких сложностях, как регистр, пробелы и пунктуация, т.е. в данной простой версии отвергаются строки типа "Otto" или "Madam, I'm Adam". Просмотрите список методов для работы со строками в приложении F для упрощения задачи.
- Решите ту же задачу, что и в упражнении 1, но позаботьтесь о регистре, пробелах и пунктуации, т.е. строка "Madam, I'm Adam" должна восприниматься как палиндром. Например, тестирующая функция должна сначала сократить строку до "madamimadam", а затем проверить, является ли обратная строка точно такой же. Не забудьте воспользоваться полезной библиотекой `cctype`. Одна

или две функции STL могут оказаться полезными, хотя и необязательными.

3. Напишите функцию, использующую интерфейс старого стиля. Она имеет такой прототип:

```
int reduce(long ar[], int n);
```

Аргументами являются имя массива и количество элементов в нем. Функция сортирует массив, удаляет повторяющиеся значения и возвращает значение, равное количеству элементов в сокращенном массиве. Напишите данную функцию с применением функций STL. (Если вы решите использовать общую функцию `unique()`, заметьте, что она возвращает конец результирующего диапазона.) Протестируйте функцию в короткой программе.

4. Решите ту же задачу, что и в упражнении 3, но примените шаблон функции:

```
template <class T>
int reduce(T ar[], int n);
```

Протестируйте функцию в короткой программе, причем используя в качестве базового типа сначала тип `long`, а затем тип `string`.

5. Перепишите пример, показанный в листинге 11.13, используя шаблон класса STL `queue` вместо класса `Queue`, введенного в главе 11.

6. Распространенной игрой является лотерея. Лотерейный билет имеет пронумерованные ячейки, из которых определенное число выбирается случайным образом. Создайте функцию `Lotto()`, имеющую два аргумента. Первый — это число ячеек на лотерейном билете, а второй — количество ячеек, выбран-

ное случайно. Функция возвращает объект типа `vector<int>`, содержащий в отсортированном порядке числа, выбранные случайно. Например, функцию можно использовать так:

```
vector<int> winners;
winners = Lotto(51, 6);
```

При этом переменной `winners` присваивается вектор, содержащий шесть чисел, случайно выбранных из диапазона от 1 до 51. Заметьте, что простого использования функции `rand()` недостаточно, так как она может возвращать повторяющиеся значения. Подсказка: напишите функцию, которая создает контейнер и заполняет его всеми возможными значениями. Затем воспользуйтесь функцией `random_shuffle()`, а после этого используйте начальный элемент такого вектора для получения значений. Кроме того, создайте программу, которая позволит тестировать функцию.

7. Мэт и Пэт хотят пригласить своих друзей на вечеринку. Они просят вас написать программу, которая может делать следующее:

Позволяет Мэту ввести список имен его друзей. Имена сохраняются в контейнере и затем отображаются в отсортированном порядке.

Позволяет Пэт ввести список имен ее друзей. Имена сохраняются в контейнере и затем отображаются в отсортированном порядке.

Создает третий контейнер, который объединяет два списка, удаляя повторяющиеся элементы, и отображает содержимое этого контейнера.

# Ввод/вывод данных и работа с файлами

**В этой главе рассматривается следующее:**

- Ввод и вывод данных с точки зрения C++
- Семейство классов *iostream*
- Перенаправление
- Методы класса *ostream*
- Форматирование результатов вывода
- Методы класса *istream*
- Состояния потока
- Файловый ввод/вывод
- Использование класса *ifstream* для ввода файлов
- Использование класса *ofstream* для вывода файлов
- Использование класса *fstream* для одновременного ввода/вывода файлов
- Обработка параметров командной строки
- Двоичные файлы
- Произвольный доступ к файлу
- Внутреннее форматирование

Обсуждение ввода/вывода (сокращенно I/O) представляет собой проблему. С одной стороны, практически каждая программа использует ввод и вывод, и знание того, как их применять, — это одна из первых задач для того, кто учит язык программирования. С другой стороны, C++ использует многие из своих сложнейших свойств для реализации ввода/вывода, включая классы, производные классы, перегруженные и виртуальные функции, шаблоны и свойство множественного наследования. Поэтому, чтобы действительно понять, как выполняется ввод/вывод данных в C++, необходимо иметь некоторые знания. Для подготовки в более ранних главах были намечены основные способы использования объекта *cin* класса *istream* и объекта *cout* класса *ostream* для выполнения ввода/вывода. Теперь мы подробнее рассмотрим классы ввода/вывода C++, узнаем, как они разработаны, и научимся управлять форматом вывода. (Если вы пропустили несколько глав и просто хотите получить более подробную информацию о форматировании, можете бегло просмотреть разделы по этому вопросу, обращая внимание на методы и пропуская объяснения.)

Средства ввода/вывода файлов C++ основаны на тех же базовых определениях классов, на которых основаны *cin* и *cout*, поэтому изучение файлового ввода/вывода начнем с рассмотрения консольного ввода/вывода (клавиатура и экран).

Комитет по стандартизации ANSI/ISO C++ поработал над тем, чтобы сделать текущий ввод/вывод C++ лучше совместимым с существующим вводом/выводом C, что привело к некоторым изменениям в традиционной практике C++.

## Обзор ввода/вывода данных в C++

В большинстве языков программирования средства ввода/вывода встроены в сам язык. Например, если посмотреть на список ключевых слов таких языков, как BASIC или Pascal, можно увидеть операторы *PRINT*, *writeln* и подобные им в словаре языка. Но ни в C, ни в C++ функции ввода/вывода не встроены. Если посмотреть на ключевые слова языка, можно увидеть операторы *for* и *if*, но при этом полностью отсутствуют операторы, связанные с вводом/выводом. Первоначально в языке C реализация ввода/вывода была оставлена на усмотрение разработчиков компиляторов. В результате разработчики получили творческую свободу при создании функций ввода/вывода данных, которые наилучшим образом смогут удовлетворять аппаратные запросы компьютера. Практически большинство разработчиков в качестве основы для ввода/вывода использовали набор функций ввода/вывода, первоначально разработанных для среды UNIX. Стандарт ANSI C формализовал применение этого пакета, названного пакетом Стандартного ввода/вывода.

вода (Standart Input/Output), сделав его обязательным компонентом стандартной библиотеки С. Язык С++ также поддерживает данный пакет, поэтому тот, кто знаком с семейством функций С, объявленных в файле `stdio.h`, может использовать их и в программах, написанных на С++. (Более новые реализации компиляторов применяют заголовочный файл `cstdio` для поддержки этих функций.)

Однако в С++ чаще используется ввод/вывод в стиле С++, а не в стиле языка С. Это решение представляется собой набор классов, определенных в заголовочных файлах `iostream` (ранее `iostream.h`) и `fstream` (ранее `fstream.h`). Данная библиотека классов не является частью формального определения языка (`cin` и `istream` не являются ключевыми словами); ведь, в конце концов, язык программирования определяет правила для того, как выполнять какие-то действия, например, создавать классы, но не определяет, что именно будет создано при использовании этих правил. Подобно тому как реализации С включают стандартную библиотеку функций, в С++ есть стандартная библиотека классов. Вначале эта стандартная библиотека была неформальным стандартом, состоящим в основном из классов, определенных в заголовочных файлах `iostream` и `fstream`. Комитет ANSI/ISO C++ решил формализовать эту библиотеку как библиотеку стандартных классов и добавить еще несколько стандартных классов, наподобие тех, что обсуждались в главе 15. В этой главе рассматривается стандартный ввод/вывод С++. Но сначала мы изучим концептуальную основу ввода/вывода С++.

## Потоки и буферы

Программа, написанная на С++, представляет ввод и вывод как поток байтов. При вводе программа читает байты из потока ввода, при выводе вставляет байты в поток вывода. Для программы, ориентированной на работу с текстом, каждый байт может быть представлением определенного символа. В общем случае байты могут формировать двоичное представление символьных или числовых данных. Байты потока ввода могут поступать с клавиатуры или такого устройства сохранения данных, как жесткий диск, либо из другой программы. Подобным образом байты потока вывода могут выводиться на экран, принтер, устройство сохранения данных или пересыпаться в другую программу. Поток действует как посредник между исходной точкой и точкой назначения потока. Такой подход позволяет программе, созданной с помощью средств языка С++, обрабатывать поток ввода с клавиатуры так же, как и ввод из файла; программа, написанная на языке С++, только рассматривает поток байтов. При этом ей не нужно знать, откуда эти байты появляются. Аналогично, используя по-

токи, программа, написанная на С++, может обрабатывать поток вывода независимо от того, куда направляются байты. Таким образом, управление вводом предполагает две стадии:

- Присоединить поток ввода к программе
- Связать поток с файлом

Другими словами, поток ввода должен иметь две связи, по одной с каждой стороны. Связь со стороны файла обеспечивает для потока источник, а связь со стороны программы обеспечивает направление потока в программу. (Связь со стороны файла может выступать в роли файла, но также она может быть и устройством, например, клавиатурой.) Подобным образом управление выводом требует связи потока вывода с программой и присоединения некоторой точки назначения вывода к потоку. Это очень похоже на прокладку труб, по которым вместо воды перетекают байты (рис. 16.1).

Обычно ввод/вывод можно осуществлять более эффективно с помощью *буфера*. Буфер — это блок памяти, используемый как промежуточное временное средство для хранения данных с последующей их пересылкой от устройства к программе или от программы к устройству. Обычно такие устройства, как диски, пересыпают информацию блоками по 512 или более байтов, в то время как программа часто обрабатывает информацию по одному байту за один раз. Буфер помогает выровнять две эти несоизмеримые скорости пересылки информации. Например, предположим, что программа должна подсчитать количество знаков доллара в файле, находящемся на жестком диске. Программа может читать один символ из файла, обрабатывать его, читать следующий символ и т.д.

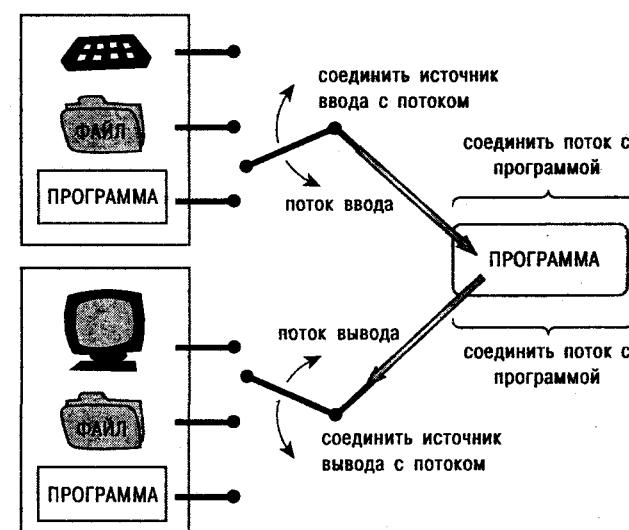


РИСУНОК 16.1 Ввод и вывод в языке С++.

Чтение файла по одному символу требует частого обращения к жесткому диску и является медленным. При буферизованном подходе считывается большой блок данных с диска, сохраняется в буфере, а затем происходит чтение из буфера по одному символу за один раз. Поскольку читать символы по одному из памяти гораздо быстрее, чем с жесткого диска, этот подход является более быстрым и простым в случае использования аппаратных средств. Конечно, когда при выполнении программы достигается конец буфера, ей необходимо прочесть следующий блок данных с диска. Этот принцип подобен принципу работы резервуара с водой, который собирает мегагаллоны избыточной воды во время большого ливня, а затем обеспечивает равномерную подачу поставку воды в дома (рис. 16.2). Аналогично при выводе программа должна сначала заполнить буфер, а затем переслать весь блок данных на жесткий диск, очистив буфер для следующей порции выводимых данных. Этот процесс называется *очисткой буфера*. Возможно, у вас есть своя водопроводная аналогия для этого процесса.

Ввод с клавиатуры обеспечивает передачу одного символа за один раз, поэтому в данном случае программе не нужен буфер для выравнивания различных показателей скоростей пересылки данных. Однако буферизованный ввод с клавиатуры позволяет пользователю двигаться в обратном направлении и корректировать ввод, прежде чем переслать его программе. Программа, написанная на C++, обычно очищает буфер ввода при нажатии клавиши Enter. Поэтому в примерах из данной книги обработка результатов ввода не начинается до тех пор, пока не будет нажата клавиша Enter. Для вывода на

экран программа, написанная на C++, обычно очищает буфер вывода при пересылке символа новой строки. В зависимости от реализации программы может очищать буфер и при других обстоятельствах, например, при приближении ввода. Так, когда программа достигает выражения, включающего оператор ввода, она очищает все текущие буферы вывода. Реализации C++, совместимые со стандартом ANSI C, должны вести себя именно таким образом.

## Потоки, буферы и файл *iostream*

Реализация потоков и буферов может быть несколько сложнее, но включение файла *iostream* (прежнее название *iostream.h*) дает несколько классов, разработанных для реализации и управления потоками и буферами. Новейшая версия ввода/вывода C++ фактически определяет шаблоны классов для поддержки обоих типов данных — *char* и *wchar\_t*. Используя свойство оператора *typedef*, C++ заставляет специализации типа *char* в этих шаблонах имитировать традиционную, нешаблонную реализацию ввода/вывода. Вот некоторые из этих классов (рис. 16.3):

- Класс *streambuf* предоставляет память для буферов, а также методы для заполнения буфера, доступа к содержимому буфера, очистке буфера и управлению памятью для буфера.
- Класс *ios\_base* представляет общие свойства потока, такие как признак открытости для чтения, а также определение того, двоичный это или текстовый поток.

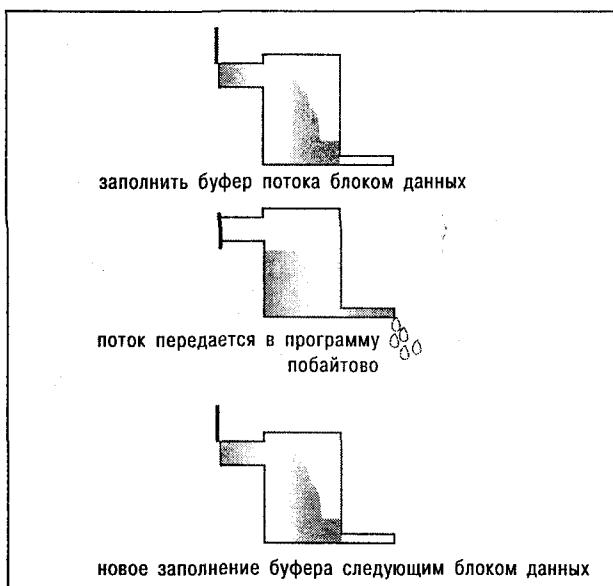


РИСУНОК 16.2 Буферизованный поток.

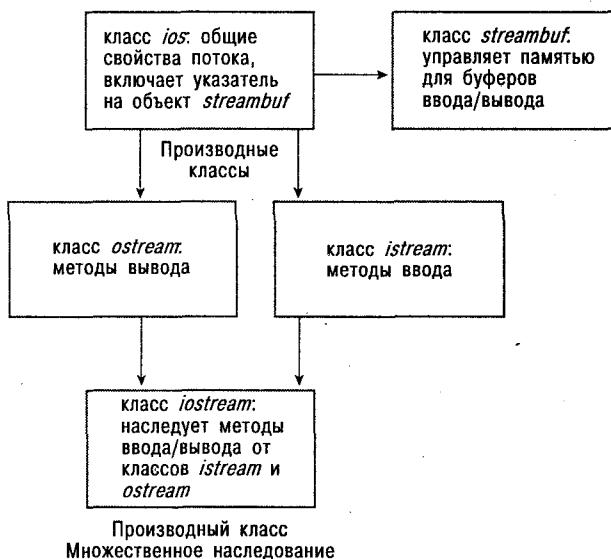


РИСУНОК 16.3 Некоторые классы ввода/вывода.

- Класс **ios** основан на классе **ios\_base** и включает элементы, являющиеся указателями на объекты класса **streambuf**.
- Классы **ostream** и **istream** являются производными от класса **ios** и поддерживают соответственно методы вывода и ввода.
- Класс **iostream** основан на классах **istream** и **ostream** и поэтому наследует и методы ввода, и методы вывода.

Чтобы воспользоваться этими средствами, необходимо применить объекты соответствующих классов. Например, объект **cout** класса **ostream** нужно использовать для работы с потоком вывода. При создании такого объекта открывается поток, автоматически создается буфер, который ставится в соответствие потоку. Это также делает доступными и функции-элементы данного класса.

#### ПЕРЕОПРЕДЕЛЕНИЕ ВВОДА/ВЫВОДА

Стандарт ISO/ANSI C++ изменяет поток ввода/вывода несколькими способами. Во-первых, произошло изменение имени файла с **ostream.h** на **ostream**, в нем к именам классов добавляется **std**. Во-вторых, классы ввода/вывода были переписаны. Чтобы C++ был интернациональным языком, он должен поддерживать интернациональную кодировку, для которой требуется 16-разрядный символьный тип. Поэтому в реализации языка был добавлен 16-разрядный тип **wchar\_t** (от слова "wide" – широкий) к традиционному 8-разрядному типу **char**. Каждый тип данных нуждается в своих собственных средствах ввода/вывода. Вместо того чтобы создавать два отдельных набора классов, Комитет по стандартизации разработал набор шаблонов классов ввода/вывода, включающий **basic\_istream<charT, traits<charT>>** и **basic\_ostream<charT, traits<charT>>**. Шаблон **traits<charT>**, в свою очередь, является классом шаблонов, определяющим такие специфические особенности символьного типа, как способ сравнения значений на предмет равенства и значение символа EOF. Стандарт устанавливает специализации типов **char** и **wchar\_t** для классов ввода/вывода. Например, **istream** и **ostream** определены оператором **typedef** для специализации типа **char**. Аналогично **wistream** и **wostream** определены для специализации типа **wchar\_t**. Например, есть объект **wcout**, предназначенный для вывода потоков 16-разрядных символов. Заголовочный файл **ostream** содержит все эти определения.

Некоторая не зависящая от типа информация, которая находилась в базовом классе **ios**, была перемещена в новый класс **ios\_base**. Она включает в себя различные константы форматирования, например, **ios::fixed**, которая превратилась теперь в **ios\_base::fixed**. Кроме того, **ios\_base** содержит свойства, которых не было ранее в классе **ios**.

В некоторых случаях изменение в именах файлов отвечает изменению в определении классов. В Microsoft Visual C++ 5.0 можно включить файл **iostream.h** и пользоваться старыми определениями классов или включить файл **iostream** и использовать новые определения классов. Однако такая синхронизация не стала еще общим правилом.

Библиотека C++ класса **iostream** заботится о многих деталях, освобождая от этого пользователя. Например, включение файла **iostream** в программу автоматически создает восемь потоковых объектов (четыре для потоков 8-разрядных символов и четыре — для 16-разрядных символов):

- Объект **cin** отвечает стандартному потоку ввода. По умолчанию этот поток поставлен в соответствие стандартному устройству ввода, обычно клавиатуре. Объект **wcin** работает подобным образом, но для типа **wchar\_t**.
- Объект **cout** отвечает стандартному потоку вывода. По умолчанию этот поток поставлен в соответствие стандартному устройству вывода, обычно монитору. Объект **wcout** работает подобным образом, но для типа **wchar\_t**.
- Объект **cerr** отвечает стандартному потоку вывода ошибок, которым можно воспользоваться для вывода ошибок. По умолчанию этот поток поставлен в соответствие стандартному устройству вывода, обычно монитору, причем данный поток не буферизован. Это значит, что информация пересыпается напрямую на экран без ожидания заполнения буфера или появления символа новой строки. Объект **wcerr** работает подобным образом, но для типа **wchar\_t**.
- Объект **clog** также отвечает стандартному потоку вывода ошибок. По умолчанию этот поток поставлен в соответствие стандартному устройству вывода, обычно монитору, причем в данном случае поток буферизован. Объект **wclog** работает подобным образом, но для типа **wchar\_t**.

Что же означает выражение: объект представляет поток? Например, когда файл **iostream** объявляет для программы объект **cout**, этот объект имеет элементы, содержащие такую информацию, связанную с выводом, как ширина полей для отображения данных, количество цифр после десятичной точки, система счисления для вывода целых чисел и адрес объекта **streambuf**, описывающего буфер для работы с потоком вывода. Выражение наподобие

```
cout << "Bjarne free";
```

помещает символы из строки "Bjarne free" в буфер, управляемый **cout** с помощью указателя на объект **streambuf**. Класс **ostream** определяет функцию **operator<<()**, используемую в выражении. Кроме того, этот класс поддерживает элементы данных объекта **cout** с помощью разных методов, один из которых рассматривается далее. Более того, C++ определяет, что вывод из буфера направляется в стандартный поток вывода,

обычно на монитор, с помощью средств, обеспечивающих операционной системой. Итак, один конец потока подключен к программе, другой — к стандартному потоку вывода, а объект `cout` с помощью объекта типа `streambuf` управляет передачей байтов в потоке.

## Перенаправление

Стандартные потоки ввода/вывода обычно имеют отношение к клавиатуре и монитору. Но многие операционные системы, включая UNIX и MS DOS, поддерживают перенаправление — средство, позволяющее изменить ассоциации для стандартных потоков ввода и вывода. Представим себе, например, что имеется выполняемая программа для системы DOS на C++ под названием `counter.exe`, которая подсчитывает количество символов в потоке ввода и выдает результат. После выполнения программы отображается следующее:

```
C>counter
Hello
and goodbye!
Control-Z — Комбинация, соответствующая
 признаку конца файла
Input contained 19 characters.
C>
```

Здесь результаты ввода поступали с клавиатуры, а вывод направлялся на экран.

Однако выполнив перенаправление ввода (<) и вывода (>), можно использовать ту же самую программу для подсчета количества символов в файле `oklahoma` и помещения результата в файл `cow_cnt`:

```
C>counter <oklahoma >cow_cnt
C>
```

Часть командной строки `<oklahoma` ассоциирует стандартный поток ввода с файлом `oklahoma`, заставляя объект `cin` читать результаты ввода из файла, а не с клавиатуры. Другими словами, операционная система изменяет связь на входном конце потока ввода, в то время как выходной конец остается связанным с программой. Часть командной строки `>cow_cnt` ассоциирует стандартный поток вывода с файлом `cow_cnt`, заставляя объект `cout` посыпать вывод в файл, а не на экран. Таким образом, операционная система изменяет подключение на выходном конце потока вывода, в то время как входной конец остается связанным с программой. Обе системы, и DOS (версия 2.0 или более поздняя), и UNIX, автоматически распознают синтаксис перенаправления. (UNIX и DOS 3.0 или более поздней версии также разрешают использовать добавочные пробелы между операторами перенаправления и именами файлов.)

Стандартный поток вывода, представленный объектом `cout`, — это обычный канал вывода программы.

Стандартные потоки вывода ошибок (обозначенные `cerr` и `clog`) предназначены для вывода сообщений об ошибках. По умолчанию все три потока посыпаются на монитор. Перенаправление стандартного потока вывода не влияет на `cerr` или `clog`. Таким образом, если один из этих объектов используется для вывода сообщений об ошибках, программа выведет сообщение об ошибке на экран, даже если стандартный поток вывода `cout` перенаправлен в другое место. Например, рассмотрим следующий фрагмент кода:

```
if (success)
 cout << "Here come the goodies!\n";
else
{
 cerr << "Something horrible has
 happened.\n";
 exit(1);
}
```

Если перенаправление не действует, любое из приведенных выше сообщений будет выведено на экран. Однако, если вывод программы перенаправлен в файл, первое сообщение при выполнении условного оператора попадет в файл, а второе сообщение будет выведено на экран. Кстати, некоторые операционные системы позволяют перенаправлять также и стандартный поток вывода ошибок. В системе UNIX, например, оператор `2>` перенаправляет стандартный поток ошибок.

Фактически классы `istream` и `ostream` необязательно могут обеспечивать перенаправление. В реализации Borland C++ 3.1, например, есть производный от `istream` класс `istream_withassign` для обработки перенаправления, а `cin` является объектом класса `istream_withassign`. Аналогично объект `cout` принадлежит к классу `ostream_withassign`, который является производным и позволяет перенаправлять поток вывода. В других случаях эти стандартные объекты используют те же самые методы, что и соответствующие им базовые классы. Для простоты далее `cin` будет обозначать объект класса `istream`, а `cout` — объект класса `ostream`.

## ВЫВОД С ПОМОЩЬЮ cout

Как уже говорилось, C++ рассматривает вывод как поток байтов. (Если используется поток `wostream`, это могут быть 16-разрядные единицы информации, объединяющие два байта.) Но многие типы данных в программе реализованы в виде больших единиц, чем отдельный байт. Например, тип `int` может быть представлен 2- или 4-байтовым двоичным значением. А значение типа `double` может быть представлено 8 байтами двоичных данных. Но, когда последовательность байтов выводится на экран, каждый байт должен представлять символьное значение. Это значит, чтобы вывести число **-2.34** на

экран, необходимо переслать пять символов - , 2, ., 3 и 4, а не внутреннее 8-байтовое представление этого значения. Поэтому одна из наиболее важных задач класса **ostream** — преобразовывать такие числовые типы, как **int** или **float**, в поток символов, которые представляют это значение в текстовой форме. Таким образом, класс **ostream** переводит внутреннее представление данных из двоичного битового шаблона в символьные байты потока вывода. (Когда-нибудь мы все будем иметь бионические имплантанты, которые позволят нам напрямую обрабатывать двоичные данные. Их разработку мы оставляем в качестве упражнения читателю.) Чтобы выполнить эту задачу перевода, класс **ostream** имеет несколько методов. Далее мы рассмотрим их, суммируя методы, использованные ранее в книге, и описывая дополнительные методы, обеспечивающие более тонкое управление потоком вывода.

## Перегруженная операция <<

Чаще всего в этой книге объект **cout** используется вместе с операцией **<<**, называемой *операцией вставки*:

```
int clients = 22;
cout << clients;
```

В C++, как и в С, символ операции **<<** по умолчанию обозначает поразрядный сдвиг влево (см. приложение Е). Выражение наподобие **x<<3** обозначает, что в бинарном представлении **x** все разряды сдвигаются влево на три ячейки. Очевидно, здесь нет ничего общего с выводом. Но класс **ostream** переопределяет операцию **<<**, и она становится перегруженной для вывода с помощью операции из класса **ostream**. В этом облике операция **<<** называется *операцией вставки*, а не операцией сдвига влево. (Операция сдвига влево получила эту новую роль, исходя из визуальных соображений, так как она предполагает движение потока информации влево.) Операция вставки перегружена так, чтобы распознавать все базовые типы языка C++:

- **unsigned char**
- **signed char**
- **char**
- **short**
- **unsigned short**
- **int**
- **unsigned int**
- **long**
- **unsigned long**
- **float**
- **double**
- **long double**

В классе **ostream** задано определение функции **operator<<()** для каждого из приведенных выше типов. (Функции, в именах которых присутствует слово **operator**, используются для перегрузки операций, как

известно из главы 10.) Поэтому если есть выражение вида

```
cout << value;
```

и если **value** имеет один из предыдущих типов, то программа, написанная на C++, может найти ему соответствие — функцию-оператор с определенной сигнатурой. Например, выражение **cout << 88** совпадает со следующим прототипом метода:

```
ostream & operator<<(int);
```

Данный прототип обозначает, что функция **operator<<()** имеет один аргумент типа **int**. Это та часть, которая совпадет с числом **88** в предыдущем выражении. Прототип также показывает, что функция возвращает указатель на объект класса **ostream**. Это свойство позволяет объединять вывод как в старой популярной рок-песне:

```
cout << "I'm feeling sedimental over "
 << boundary << "\n";
```

Если вы программист, работающий с языком С, и вам надоели многочисленные спецификаторы типа **%** и проблемы, возникающие при несовпадении типа спецификатора и переменной, убедитесь, что использование **cout** гораздо проще.

## Вывод и указатели

Класс **ostream** определяет также оператор вставки для следующих типов указателей:

- **const signed char \***
- **const unsigned char \***
- **const char \***
- **void \***

Не забудьте, что C++ представляет строку указателем на ее местонахождение. Указатель может принимать форму имени массива типа **char** или явного указателя на тип **char** или строки в кавычках. Поэтому все следующие выражения отображают строки:

```
char name [20] = "Dudly Diddlemore";
char * pn = "Violet D'Amore";
cout << "Hello!";
cout << name;
cout << pn;
```

Эти методы используют нулевой символ конца строки, чтобы определить, когда нужно прекратить вывод символов.

Язык C++ согласовывает указатель любого другого типа с типом **void \*** и выводит числовое представление адреса. Если нужен адрес строки, его необходимо привести к другому типу.

```
int eggs = 12;
char * amount = "dozen";
```

```

cout << &eggs; // выводит на экран адрес
 // переменной eggs
cout << amount; // выводит на экран
 // строку "dozen"
cout << (void *) amount; // выводит на
 // экран адрес строки "dozen"

```

### ПРИМЕЧАНИЕ

Не все текущие реализации C++ имеют прототип с аргументом типа **void \***. Поэтому необходимо привести указатель к типу **unsigned** или, возможно, к типу **unsigned long**, если необходимо вывести на экран значение адреса.

### Конкатенация вывода

Все воплощения оператора вставки определены так, чтобы возвращать значение типа **ostream &**, т.е. прототипы имеют форму:

```
ostream & operator<<(type);
```

(Здесь **type** — это выводимый тип.) Возвращаемый тип **ostream &** означает, что при использовании этого оператора возвращается указатель на объект **ostream**. Какой объект? Определение функции говорит о том, что это указатель на объект, который вызывает оператор. Другими словами, возвращаемое значение операторной функции совпадает с передаваемым ей объектом. Например, **cout << "potluck"** возвращает объект типа **cout**. Именно это свойство позволяет конкатенировать вывод, используя оператор вставки. Например, рассмотрим следующее выражение:

```

cout << "We have" << count
 << "unhatched chickens.\n";

```

Выражение **cout << "We have"** выводит на экран строку и возвращает объект типа **cout**, сокращая выражение до следующего:

```
cout << count << "unhatched chickens.\n";
```

Затем выражение **cout << count** выводит на экран значение переменной **count** и возвращает **cout**, который затем обрабатывает финальный аргумент выражения (рис. 16.4). Такой метод разработки является поистине удобным свойством, поэтому все примеры перегрузки операции **<<** в предыдущих главах имитировали его.

### Другие методы класса ostream

Несмотря на многочисленные функции **operator<<()**, класс **ostream** включает метод **put()** для отображения символов и метод **write()** для отображения строк.

Метод **put()** практически не используется. Традиционно он имеет следующий прототип:

```
ostream & put(char);
```

В текущем стандарте он имеет эквивалентную форму, за тем лишь исключением, что в этом случае сущест-

ствует шаблон для типа **wchar\_t**. Его можно применять, используя обычную запись метода класса:

```
cout.put('W'); // вывести символ W
```

Здесь **cout** — это объект, а **put()** — функция-элемент класса. Как и операторные функции **<<**, данная функция возвращает указатель на вызываемый объект, поэтому с ее помощью тоже можно конкатенировать вывод:

```
cout.put('I').put('t'); // вывод слова It
 // путем двух вызовов функции put()
```

В результате вызова функции **cout.put('I')** возвращается объект **cout**, который затем действует как вызываемый объект для функции **put('t')**.

При наличии соответствующего прототипа можно использовать функцию **put()** с аргументами таких числовых типов, как **int**, а функция сама преобразует аргумент в правильное значение типа **char**. Например, можно сделать следующее:

```
cout.put(65); // вывести символ A
cout.put(66.3); // вывести символ B
```

В первом выражении значение **65** типа **int** преобразовано в значение типа **char**, а затем выводится символ, имеющий ASCII-код **65**. Аналогично во втором выражении значение **66.3** типа **double** преобразуется в тип **char** и выводится символ с соответствующим кодом.

Такое поведение было удобным до C++ версии 2.0; в то время язык представлял символьные константы значениями типа **int**. Поэтому в выражении вроде

```
cout << 'W';
```

```
char * name = "Bingo"
cout << "What ho! " << name << "!\n";
```

направляет строку *What ho!* в буфер вывода и возвращает объект **cout**

```
cout << name << "!\n";
```

направляет строку *Bingo* в буфер вывода и возвращает объект **cout**

```
cout << "!\n";
```

направляет строку *!* в буфер вывода и возвращает объект **cout** (возвращаемое значение не используется)

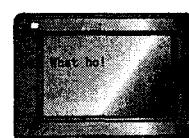


РИСУНОК 16.4 Конкатенация вывода.

'W' интерпретируется как значение типа `int` и поэтому выводится как целое число 87, т.е. отображается ASCII-код символа. Но выражение

```
cout.put('W');
```

срабатывает отлично. Поскольку теперь C++ представляет символьные константы как тип `char`, можно пользоваться обоими методами.

Проблема реализации состоит в том, что некоторые компиляторы перегружают функцию `put()` для трех типов аргументов: `char`, `unsigned char` и `signed char`. Это делает использование `put()` с аргументом типа `int` неоднозначным, поскольку значение типа `int` должно быть преобразовано к одному из этих трех типов.

Метод `write()` выводит целую строку и имеет следующий прототип:

```
basic_ostream<charT,traits>& write(const
char_type* s, streamsize n);
```

Первый аргумент функции `write()` представляет собой адрес строки, а второй — количество выводимых символов. Использование `cout` с методом `write()` требует специализации типа `char`, поэтому возвращаемое значение имеет тип `ostream &`. В листинге 16.1 показано, как работает метод `write()`.

### Листинг 16.1 Программа write.cpp.

```
// write.cpp - использование cout.write()
#include <iostream>
using namespace std;
#include <cstring> // или string.h

int main()
{
 const char * state1 = "Ohio";
 const char * state2 = "Utah";
 const char * state3 = "Euphoria";

 int len = strlen(state2);
 cout << "Increasing loop index:\n";
 int i;
 for (i = 1; i <= len; i++)
 {
 cout.write(state2,i);
 cout << "\n";
 }

// конкатенация вывода
 cout << "Decreasing loop index:\n";
 for (i = len; i > 0; i--)
 cout.write(state2,i) << "\n";

// выход за пределы строки
 cout << "Exceeding string length:\n";
 cout.write(state2, len + 5) << "\n";

 return 0;
}
```

Результаты выполнения программы:

```
U
Ut
Uta
Utah
Utah
Uta
Ut
U
Utah E
```

Заметьте, что в результате вызова функции `cout.write()` возвращается объект `cout`. Так происходит потому, что метод `write()` возвращает указатель на объект, который его вызывает, в данном случае это объект `cout`. Такое свойство делает возможным конкатенировать вывод, поскольку `cout.write()` заменяется возвращаемым им значением, т.е. `cout`:

```
cout.write(state2,i) << "\n";
```

Обратите также внимание, что метод `write()` не перестает выводить символы автоматически при достижении нулевого символа. Он просто выводит столько символов, сколько ему задано, даже если при этом выходит за пределы строки! В данном случае программа группирует строку "Utah" с двумя другими строками, поэтому прилегающие области памяти также содержат данные. Компиляторы отличаются в том, как они сохраняют данные в памяти и как они выравнивают значения в памяти. Например, строка "Utah" занимает пять байтов, но данный компилятор выравнивает длину строки так, чтобы она была кратна четырем байтам, поэтому строка "Utah" увеличена до восьми байтов.

Метод `write()` можно также использовать и с числовыми данными. Он не преобразует число в соответствующий набор символов; вместо этого он выводит битовое представление числа, как оно сохраняется в памяти. Например, 4-байтовое значение типа `long` 560031841 будет выведено как четыре независимых байта. Устройство вывода, такое как монитор, будет интерпретировать каждый байт как ASCII-код. Поэтому 560031841 отображается на экране как 4-символьная комбинация, скорее всего, бессмысленная. Однако метод `write()` обеспечивает удобный и точный способ записи данных в файл. К этой возможности мы еще вернемся позже в данной главе.

### Очистка буфера вывода

Представьте себе, что происходит, когда программа использует `cout` для пересылки байтов в стандартный поток вывода. Поскольку вывод класса `ostream` буферизован объектом `cout`, то результаты вывода не попадают в точку назначения немедленно. Вместо этого они накапливаются в буфере до тех пор, пока тот не заполнится. Затем программа очищает буфер, пересыпая его содер-

жимое дальше и очищая его для новых данных. Обычно размер буфера равен или кратен 512 байтам. Буферизация позволяет очень сильно экономить время в том случае, когда стандартный поток вывода соответствует файлу, находящемуся на жестком диске. В конце концов, не хотелось бы, чтобы программа 512 раз обращалась к жесткому диску, чтобы записать 512 байтов. Гораздо эффективнее собрать 512 байтов в буфер и записать их на диск во время одной дисковой операции.

Однако для вывода на экран заполнение буфера не является столь решающим. Наоборот, это было бы даже неудобно, если бы пришлось повторять сообщение "Press any key to continue" ("Нажмите любую клавишу для продолжения") до тех пор, пока в буфере не набрались бы необходимые 512 байтов. К счастью, в случае вывода на экран программа не обязательно будет ожидать заполнения буфера. Например, вывод символа новой строки обычно заставляет программу очистить буфер. Кроме того, как было отмечено ранее, большинство реализаций очищает буфер при ожидании ввода. Представьте себе следующий код:

```
cout << "Enter a number: ";
float num;
cin >> num;
```

Тот факт, что программа ожидает ввода, заставляет ее вывести сообщение объекта `cout` (т.е. очистить буфер от сообщения "Enter a number:") немедленно, даже если строка вывода не имеет символа новой строки. Без этого свойства программа бы ждала выполнения ввода, даже не выведя пользователю сообщение объекта `cout`.

Если конкретная реализация не очищает буфер вывода тогда, когда это необходимо, можно форсировать очистку буфера с помощью одного из двух манипуляторов. Манипулятор `flush` очищает буфер, а манипулятор `endl` очищает буфер и добавляет символ новой строки. Эти манипуляторы можно использовать так же, как и обычные имена переменных:

```
cout << "Hello, good-looking! " << flush;
cout << "Wait just a moment, please."
<< endl;
```

Манипуляторы фактически являются функциями. Например, можно очистить буфер `cout` путем прямого вызова функции `flush()`:

```
flush(cout);
```

Однако класс `ostream` перегружает операцию вставки `<<` таким образом, чтобы выражение

```
cout << flush
```

заменилось вызовом функции `flush(cout)`. Поэтому с успехом можно пользоваться более удобной записью для очистки буфера.

## Форматирование вывода с помощью `cout`

Операции вставки класса `ostream` преобразуют значения в текстовый вид. По умолчанию они форматируют значения следующим образом:

- Значение типа `char`, если оно представляется собой символ, который можно вывести, печатается как символ в поле, размер которого составляет один символ.
- Числовые значения целого типа выводятся как десятичные целые числа в поле, размер которого достаточен для вывода всех цифр и при необходимости для вывода знака "минус".
- Строки выводятся в поле, размер которого равен длине строки.

Поведение, заданное по умолчанию, при выводе значений чисел с плавающей точкой изменилось. В следующем списке представлены различия в старой и новой реализациях:

- (Старый стиль) Типы чисел с плавающей точкой выводятся с шестью полями после десятичной точки; замыкающие нули не выводятся. (Заметьте, что количество выводимых цифр никак не связано с точностью, с которой число хранится в памяти.) Число выводится в записи с фиксированной десятичной точкой или в научной записи (см. главу 3) — в зависимости от величины числа. Снова, как и в предыдущем случае, поле достаточно велико, чтобы поместить число и, если нужно, знак "минус".
- (Новый стиль) Типы чисел с плавающей точкой выводятся в шести полях, замыкающие нули не выводятся. (Заметьте, что количество выводимых цифр никак не связано с точностью, с которой число хранится в памяти.) Число выводится в записи с фиксированной десятичной точкой или в научной записи (см. главу 3) — в зависимости от величины числа. В частности, научная запись используется в том случае, если показатель имеет значение 6 и больше или -5 и меньше. Снова поле является достаточно большим, чтобы поместить число и при необходимости знак "минус". Поведение, заданное по умолчанию, отвечает использованию функции `printf()` стандартной библиотеки C со спецификатором `%g`.

Поскольку каждое значение выводится в поле, размер которого соответствует этому значению, необходимо явно указывать пробелы между значениями, иначе последовательные значения сольются вместе.

Существует несколько небольших различий между форматированием раннего C++ и текущим стандартом; они приведены в табл. 16.3 далее в этой главе.

Листинг 16.2 иллюстрирует вывод, заданный по умолчанию. После каждого значения выводится двоеточие (:), что позволяет увидеть размер поля, используемого в каждом случае. В программе используется выражение  $1.0 / 9.0$ , что позволяет получить бесконечную дробь и увидеть, сколько значений реально отображается.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Не все компиляторы генерируют вывод, форматированный в соответствии с текущим стандартом. Кроме того, текущий стандарт позволяет вносить региональные вариации. Например, в европейской реализации может использоваться запятая вместо точки в десятичных дробях, т.е. программа будет выводить **2,54** вместо **2.54**. В библиотеке локальных настроек (заголовочный файл **locale**) имеется механизм для задания определенного стиля для ввода или вывода, поэтому один и тот же компилятор может содержать разные локальные настройки. В данной главе используются локальные настройки для США.

### Листинг 16.2 Программа **defaults.cpp**.

```
// defaults.cpp - форматы cout,
// заданные по умолчанию
#include <iostream>
using namespace std;

int main()
{
 cout << "12345678901234567890\n";
 char ch = 'K';
 int t = 273;
 cout << ch << ":\n";
 cout << t << ":\n";
 cout << -t << ":\n";

 double f1 = 1.200;
 cout << f1 << ":\n";
 cout << (f1 + 1.0 / 9.0) << ":\n";

 double f2 = 1.67E2;
 cout << f2 << ":\n";
 f2 += 1.0 / 9.0;
 cout << f2 << ":\n";
 cout << (f2 * 1.0e4) << ":\n";

 double f3 = 2.3e-4;
 cout << f3 << ":\n";
 cout << f3 / 10 << ":\n";
 return 0;
}
```

Результаты выполнения программы:

12345678901234567890

K:

273:

-273:

1.2:

1.31111:

167:

167.111:

1.67111e+006:

0.00023:

2.3e-005:

Каждое значение заполняет свое поле. Заметьте, что замыкающие нули в числе 1.200 не выводятся, но значения с плавающей точкой без таких нулей выводятся в шести полях. Кроме того, в данной реализации отводится три поля для значения показателя степени.

### Изменение системы счисления при выводе

Класс **ostream** наследуется из класса **ios**, который, в свою очередь, наследуется из класса **ios\_base**. В классе **ios\_base** сохраняется информация, необходимая для форматирования вывода. Например, определенные биты в одном из элементов класса показывают, какая система счисления используется для вывода, а другой элемент класса определяет ширину поля. Используя манипуляторы, можно управлять системой счисления при выводе целых чисел. С помощью функций-элементов класса **ios\_base** можно контролировать ширину и количество полей, отводимых для отображения цифр, находящихся справа от десятичной точки. Поскольку класс **ios\_base** является косвенным базовым классом для класса **ostream**, можно использовать его методы вместе с такими объектами класса **ostream** (или его потомков), как **cout**.

### ПРИМЕЧАНИЕ

Методы и элементы, которые находятся в классе **ios\_base**, ранее содержались в классе **ios**. Теперь класс **ios\_base** является базовым для класса **ios**. В новой системе класс **ios** представляет собой класс-шаблон со специализациями **char** и **wchar\_t**, а класс **ios\_base** содержит нешаблонные свойства.

Давайте посмотрим, как установить систему счисления для вывода целых чисел. Чтобы вывести целые числа в десятичной, шестнадцатиричной или восьмеричной системе счисления, можно воспользоваться манипуляторами **dec**, **hex** или **oct**. Например, при вызове функции

**hex(cout);**

устанавливается шестнадцатиричная система счисления для форматирования в объекте **cout**. После однократного вызова программа будет выводить числа в шестнадцатиричной форме до тех пор, пока состояние форматирования не будет изменено. Заметьте, что манипуляторы не являются функциями-элементами класса, поэтому их нельзя вызвать с помощью объектной записи.

Несмотря на то что манипуляторы реально являются функциями, их обычно используют следующим образом:

**cout << hex;**

Класс **ostream** перегружает операцию **<<** так, чтобы ее использование было эквивалентно вызову функции **hex(cout)**. Листинг 16.3 иллюстрирует использование этих манипуляторов. Программа выводит целое значе-

ние и его квадрат в трех различных системах счисления. Заметьте, что манипулятор можно использовать отдельно или же в наборе с несколькими операциями вставки.

### Листинг 16.3 Программа manip.cpp.

```
//manip.cpp - использование манипуляторов формата
#include <iostream>
using namespace std;
int main()
{
 cout << "Enter an integer: ";
 int n;
 cin >> n;

 cout << "n n*n\n";
 cout << n << " "
 << n * n << " (decimal)\n";

// установить шестнадцатиричный режим
 cout << hex;
 cout << n << " ";
 cout << n * n << " (hexadecimal)\n";

// установить восьмеричный режим
 cout << oct << n << " "
 << n * n << " (octal)\n";

// альтернативный способ использования
// манипулятора
 dec(cout);
 cout << n << " " << n * n
 << " (decimal)\n";

 return 0;
}
```

Результаты выполнения программы:

```
Enter an integer: 13
n n*n
13 169 (decimal)
d a9 (hexadecimal)
15 251 (octal)
13 169 (decimal)
```

### Установка ширины полей

Можно заметить, что колонки в предыдущем примере не выровнены; так происходит потому, что числа размещены в полях с разной шириной. Можно воспользоваться функцией-элементом `width`, чтобы поместить различные числа в поля одинаковой ширины. Метод имеет следующие прототипы:

```
int width();
int width(int i);
```

Первая форма возвращает текущую установку для ширины полей. Вторая устанавливает ширину поля равной `i` пробелам и возвращает предыдущее значение ширины поля. Это позволяет сохранить предыдущее значение в том случае, если оно может понадобиться в будущем.

Метод `width()` влияет только на следующий вывод, а после него ширина поля возвращается к предыдущему

значению. Например, рассмотрим следующее выражение:

```
cout << '#';
cout.width(12);
cout << 12 << "#" << 24 << "#\n";
```

Поскольку `width()` является функцией-элементом, необходимо использовать объект (`cout` в данном случае) для его вызова. Выражение отображает на экране следующую информацию:

```
12#24#
```

Число 12 выводится в поле из 12 символов, выровненное по правой стороне. Это называется выравниванием справа. После этого ширина поля возвращается к значению, заданному по умолчанию, и два символа `#` и число 24 выводятся в полях необходимого для них размера.

### ПОМНИТЕ

Метод `width()` влияет только на следующий отображаемый элемент, а после этого значение ширины поля принимает предыдущее значение.

Язык C++ никогда не урезает отображаемые данные, поэтому, если попытаться вывести на печать значение из семи символов в поле из двух символов, C++ расширит поле, чтобы вместить данные. (Некоторые языки программирования просто заполняют поле звездочками, если данные не помещаются в него. Философия C/C++ заключается в том, что вывод данных гораздо важнее, чем аккуратность выводимых колонок; C++ считает, что содержимое важнее формы.) Листинг 16.4 показывает, как работает функция-элемент `width()`.

### Листинг 16.4 Программа width.cpp.

```
// width.cpp - использование метода width
#include <iostream>
using namespace std;
int main()
{
 int w = cout.width(30);
 cout << "default field width = "
 << w << ":\n";

 cout.width(5);
 cout << "N" << ':';
 cout.width(8);
 cout << "N * N" << ":\n";

 for (long i = 1; i <= 100; i *= 10)
 {
 cout.width(5);
 cout << i << ':';
 cout.width(8);
 cout << i * i << ":\n";
 }

 return 0;
}
```

Результаты выполнения программы:

```
default field width = 0:
N: N * N:
1: 1:
10: 100:
100: 10000:
```

При выводе значения выровнены справа и помещены в соответствующие поля. Часть вывода заполнена пробелами, т.е. объект `cout` получает полную ширину поля, добавляя пробелы. При выравнивании справа пробелы вставляются с левой стороны. Символ, используемый для добавления к полю, называется *символом-заполнителем*. По умолчанию выполняется выравнивание справа.

Заметьте, что программа устанавливает ширину поля равной 30 только для строки в первом выражении с `cout`, но не для значения `w`. Так происходит, потому что метод `width()` влияет только на вывод, следующий за его вызовом строки. Обратите также внимание, что переменная `w` имеет значение 0. Это объясняется тем, что `cout.width(30)` возвращает предыдущую ширину поля, а не ту, которая была только что установлена. Тот факт, что `w` равно нулю, означает, что и ширина поля равна нулю. Поскольку C++ расширяет поле, чтобы уместить данные, эта ширина подходит для всех типов данных. В заключение программа использует метод `width()` для выравнивания заголовков колонок и данных, устанавливая ширину, равную пяти символам для первой колонки и восьми символам для второй колонки.

## Символы-заполнители

По умолчанию `cout` заполняет поля пробелами. Для изменения символа-заполнителя можно использовать метод `fill()`. Например, в результате вызова метода

```
cout.fill('*');
```

символ-заполнитель заменяется на звездочку. Это может быть удобно, например, для печати чеков, чтобы получатель не мог добавить несколько цифр к числу. Листинг 16.5 иллюстрирует применение этого метода.

## Листинг 16.5 Программа fill.cpp.

```
// Изменение символа-заполнителя для полей
#include <iostream>
using namespace std;
int main()
{
 cout.fill('*');
 char * staff[2] = { "Waldo Whipsnade",
 "Wilmarie Wooper"};
 long bonus[2] = { 900, 1350 };
 for (int i = 0; i < 2; i++)
 {
 cout << staff[i] << ": $";
 cout.width(7);
 cout << bonus[i] << "\n";
 }
 return 0;
}
```

Результаты выполнения программы:

```
Waldo Whipsnade: *****900
Wilmarie Wooper: *****1350
```

Заметьте, что, в отличие от ширины полей, новый символ заполнения будет действовать до тех пор, пока его не изменить явно.

## Установка точности при выводе чисел с плавающей точкой

Значение точности чисел с плавающей точкой зависит от режима вывода. В режиме, заданном по умолчанию, она обозначает количество выводимых цифр. В фиксированном или научном режиме, которые обсуждаются далее, точность обозначает количество цифр выводимых после десятичной точки. Точность, установленная для C++ по умолчанию, равна 6. (Вспомните, что замыкающие нули отбрасываются.) Функция-элемент `precision()` позволяет выбрать другие значения. Например, выражение

```
cout.precision(2);
```

заставляет объект `cout` использовать точность, равную 2. В отличие от случая с функцией `width()`, но подобно случаю с функцией `fill()`, новые установки точности остаются действовать до их явного обновления. Листинг 16.6 иллюстрирует эти свойство.

## Листинг 16.6 Программа precise.cpp.

```
// precise.cpp - установка точности
#include <iostream>
using namespace std;
int main()
{
 float price1 = 20.40;
 float price2 = 1.9 + 8.0 / 9.0;
 cout << "\"Furry Friends\" is $"
 << price1 << "!\n";
 cout << "\"Fiery Fiends\" is $"
 << price2 << "!\n";
 cout.precision(2);
 cout << "\"Furry Friends\" is $"
 << price1 << "!\n";
 cout << "\"Fiery Fiends\" is $"
 << price2 << "!\n";
 return 0;
}
```

 **ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**  
Более ранние версии C++ интерпретируют заданную точность как количество цифр после десятичной точки, а не как общее количество цифр.

Результаты выполнения программы:

```
"Furry Friends" is $20.4!
"Fiery Fiends" is $2.78889!
"Furry Friends" is $20!
"Fiery Fiends" is $2.8!
```

Заметьте, что в третьей строке замыкающие нули после десятичной точки не выводятся. Кроме того, в четвертой строке выводятся только две цифры.

### Вывод замыкающих нулей и десятичной точки

Некоторые формы вывода, например цены или цифры в колонках, выглядят лучше, если замыкающие нули присутствуют. Например, вывод в листинге 16.6 выглядел бы лучше, если бы имел вид \$20.40 а не \$20.4. Семейство классов `iostream` не включает функцию, которая могла бы выполнить эту задачу. Однако класс `ios_base` содержит функцию `setf()` (сокращение от слов *set flag* — установить флаг), которая управляет определенными свойствами форматирования. Класс также определяет несколько констант, которые можно использовать в качестве аргументов этой функции. Например, вызов функции

```
cout.setf(ios_base::showpoint);
```

заставляет `cout` вывести десятичную точку. Ранее (но не на данный момент) он также приводил и к выводу на экран замыкающих нулей. Так, вместо того чтобы число 2.00 вывести как 2, объект `cout` выводит его в виде 2.000000 (старое форматирование C++) или 2. (текущее форматирование), если точность, заданная по умолчанию, равна 6. В листинге 16.7 это выражение добавлено к программе из листинга 16.6.

 **ПРЕДОСТЕРЕЖЕНИЕ**  
Если используемый компилятор включает заголовочный файл `iostream.h` вместо `iostream`, то, скорее всего, необходимо использовать класс `ios` вместо класса `ios_base` в качестве аргумента функции `setf()`.

Если запись `ios_base::showpoint` вам кажется непонятной, знайте, что `showpoint` — это статическая константа в диапазоне доступа класса `ios_base`, определенная в объявлении класса. Диапазон доступа класса указывает, что нужно использовать оператор диапазона доступа `(::)` с именем константы, если это имя применяется за пределами метода класса. Таким образом, `ios_base::showpoint` — это имя константы, определенной в классе `ios_base`.

### Листинг 16.7 Программа showpt.cpp.

```
// showpt.cpp — устанавливает точность,
// отображает десятичную точку
#include <iostream>
using namespace std;

int main()
{
 float price1 = 20.40;
 float price2 = 1.9 + 8.0 / 9.0;

 cout.setf(ios_base::showpoint);
 cout << "\"Furry Friends\" is $"
 << price1 << "!\n";
 cout << "\"Fiery Fiends\" is $"
 << price2 << "!\n";

 cout.precision(2);
 cout << "\"Furry Friends\" is $"
 << price1 << "!\n";
 cout << "\"Fiery Fiends\" is $"
 << price2 << "!\n";

 return 0;
}
```

Результаты выполнения программы с текущим форматированием приведены ниже. Заметьте, что замыкающие нули не выводятся, но десятичная точка в третьей строке отображена.

```
"Furry Friends" is $20.4!
"Fiery Fiends" is $2.78889!
"Furry Friends" is $20.!
"Fiery Fiends" is $2.8!
```

Как можно вывести на печать замыкающие нули? Для ответа на этот вопрос нам необходимо более подробно рассмотреть функцию `setf()`.

### Подробнее о функции setf()

Метод `setf()` управляет несколькими свойствами форматирования, поэтому его необходимо рассмотреть подробнее. В классе `ios_base` содержится защищенный элемент, в котором отдельные биты (называемые в данном контексте *флагами*) управляют различными аспектами форматирования, включая систему счисления и вывод замыкающих нулей. Процесс отметки флага называется *установкой флага* (или бита) и означает, что значение бита установлено равным 1. (Если вам когда-нибудь приходилось устанавливать DIP-переключатели при конфигурировании аппаратного обеспечения, знайте, что битовые флаги являются их программным эквивалентом.) Манипуляторы `hex`, `dec` и `oct`, например, устанавливают биты, отвечающие за систему счисления. Функция `setf()` обеспечивает другой способ установки флагов.

Функция `setf()` имеет два прототипа. Первый из них:

```
fmtflags setf(fmtflags);
```

Здесь `fmtflags` — это имя для оператора `typedef` типа **битовая маска**, используемого для хранения флагов формата. Имя определено в классе `ios_base`. Эта версия функции `setf()` используется для установки информации о форматировании, контролируемой одним битом. Аргументом является значение `fmtflags`, в котором установлены необходимые биты. Возвращаемое значение определяется предыдущей установкой флагов. Это значение можно сохранить, если в будущем понадобится восстановить исходные настройки. Какое значение необходимо передать функции `setf()`? Если нужно установить бит 11 равным 1, то необходимо передать функции число, в котором бит 11 равен 1. В возвращаемом значении бит 11 будет иметь предшествующее вызову функции состояние. Проверка значений необходимых битов выглядит (и является) утомительной. Однако этого и не нужно делать, поскольку в классе `ios_base` определены констан-

ты, представляющие битовые значения. В табл. 16.1 показаны некоторые из определений.

### ПРИМЕЧАНИЕ

Тип битовой маски используется для сохранения значений отдельных битов. Это может быть целый тип, тип `enum` или STL-контейнер `bitset`. Главная идея заключается в том, что возможен индивидуальный доступ к каждому биту. Пакет `iostream` использует тип битовой маски для сохранения информации о состоянии.

Поскольку константы форматирования определены внутри класса `ios_base`, перед ними необходимо вставлять оператор диапазона доступа, т.е. нужно писать `ios_base::uppercase`, а не просто `uppercase`. Изменения остаются в силе до тех пор, пока новые значения не будут вновь указаны явно. Листинг 16.8 иллюстрирует использование некоторых из этих констант.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В более ранних реализациях может использоваться класс `ios` вместо `ios_base`, а также может не поддерживаться константа `boolalpha`.

Таблица 16.1 Константы форматирования.

| Константа                        | Значение                                                                                       |
|----------------------------------|------------------------------------------------------------------------------------------------|
| <code>ios_base::boolalpha</code> | Задает ввод и вывод значений типа <code>bool</code> как <code>true</code> и <code>false</code> |
| <code>ios_base::showbase</code>  | Задает применение базовых префиксов C++ (0,0x) при выводе                                      |
| <code>ios_base::showpoint</code> | Задает печать замыкающей десятичной точки                                                      |
| <code>ios_base::uppercase</code> | Задает применение прописных букв для шестнадцатиричного вывода и экспоненциальной записи       |
| <code>ios_base::showpos</code>   | Задает отображение знака “плюс” перед положительными числами                                   |

Листинг 16.8 Программа `setf.cpp`.

```
// setf.cpp – использование функции setf() для контроля форматирования
#include <iostream>
using namespace std;

int main()
{
 int temperature = 63;

 cout << "Today's water temperature: ";
 cout.setf(ios_base::showpos); // вывести знак "плюс"
 cout << temperature << "\n";

 cout << "For our programming friends, that's\n";
 cout << hex << temperature << "\n"; // использование 16-ричной системы счисления
 cout.setf(ios_base::uppercase); // использование прописных букв для обозначения
 // шестнадцатиричных чисел

 cout.setf(ios_base::showbase); // использование префикса 0X для обозначения 16-ричных чисел
 cout << "or\n";
 cout << temperature << "\n";
 cout << "How " << true << "! oops-How ";
 cout.setf(ios_base::boolalpha);
 cout << true << "!\n";

 return 0;
}
```

Результаты выполнения программы из листинга 16.8:

```
"Furry Friends" is $20.4!
"Fiery Fiends" is $2.78889!
"Furry Friends" is $20.!
"Fiery Fiends" is $2.8!
```

Заметьте, что знак "плюс" используется только в десятичной системе счисления. C++ рассматривает шестнадцатиричные и восьмеричные значения как беззнаковые, поэтому знак им не требуется. (Однако в некоторых реализациях может отображаться знак "плюс".)

Второй прототип функции `setf()` имеет два аргумента и возвращает значение предыдущих установок:

```
fmtflags setf(fmtflags , fmtflags);
```

Перегруженная форма этой функции используется для выбора такого форматирования, которое управляется несколькими битами. Первый аргумент, как и ранее, представляет собой значение `fmtflags`, содержащее необходимые настройки. Второй аргумент содержит такое значение, которое сначала очищает соответствующие биты. Например, пусть установка бита 3 равным 1 означает использование десятичной системы счисления, установка бита 4 равным 1 — восьмеричной, а бита 5 равным 1 — шестнадцатиричной. Пусть поток вывода использует десятичную систему, которую необходимо заменить шестнадцатиричной. Для этого необходимо не только установить бит 5 равным 1, но и установить значение бита 3 равным 0, это называется *очисткой бита*. Манипулятор `hex` выполняет обе эти задачи автоматически. Применение функции `setf()` требует большей работы, поскольку первый аргумент показывает, какие биты необходимо установить, а второй — какие очистить. Это не так сложно, как может показаться, поскольку именно для этой цели класс `ios_base` определяет набор констант (показанных в табл. 16.2). В частности, нужно использовать константу `ios_base::basefield` как второй аргумент функции, а `ios_base::hex` — как первый при изменении системы счисления.

Таблица 16.2 Аргументы функции `setf(long, long)`.

| Первый аргумент                    | Второй аргумент                   | Значение                                                                   |
|------------------------------------|-----------------------------------|----------------------------------------------------------------------------|
| <code>ios_base::basefield</code>   | <code>ios_base::dec</code>        | Задает использование десятичной системы счисления                          |
|                                    | <code>ios_base::oct</code>        | Задает использование восьмеричной системы счисления                        |
|                                    | <code>ios_base::hex</code>        | Задает использование шестнадцатиричной системы счисления                   |
| <code>ios_base::floatfield</code>  | <code>ios_base::fixed</code>      | Задает использование записи в виде десятичной дроби                        |
|                                    | <code>ios_base::scientific</code> | Задает применение экспоненциальной записи                                  |
| <code>ios_base::adjustfield</code> | <code>ios_base::left</code>       | Задает выравнивание слева                                                  |
|                                    | <code>ios_base::right</code>      | Задает выравнивание справа                                                 |
|                                    | <code>ios_base::internal</code>   | Знак или префикс системы счисления выровнять слева, число выровнять справа |

Иначе говоря, вызов функции

```
cout.setf(ios_base::hex,
ios_base::basefield);
```

имеет тот же эффект, что и применение манипулятора `hex`.

Класс `ios_base` определяет три набора флагов для форматирования, с которыми можно работать таким способом. Каждый набор состоит из одной константы, используемой как первый аргумент, и двух или трех констант, используемых как второй. Второй аргумент очищает набор битов; затем первый аргумент устанавливает один из битов равным 1. В табл. 16.2 показаны имена констант, используемых в качестве второго аргумента функции `setf()`, соответствующий выбор констант для первого аргумента и их значения. Например, для установки выравнивания слева можно воспользоваться `ios_base::adjustfield` как вторым аргументом и `ios_base::left` — как первым. Выравнивание слева означает, что число начинает выводиться с левой стороны поля, а выравнивание справа означает, что число заканчивает выводиться с правого края. Внутреннее выравнивание означает, что все префиксы или знаки помещаются слева, а само число — справа. (К сожалению, редактор языка C++ не имеет режима автовыравнивания.)

Запись с десятичной точкой означает, что для чисел с плавающей точкой используется стиль 123.4 независимо от величины числа, а экспоненциальная запись означает, что независимо от величины числа используется стиль 1.23e04.

В стандарте оба типа записи имеют следующие свойства:

- Заданная точность означает количество цифр после десятичной точки, а не общее количество цифр.
- Выводятся замыкающие нули.

В старой реализации языка замыкающие нули не выводились, если только не был установлен флаг

`ios::showpoint`. Кроме того, точность всегда означала количество цифр справа от точки, даже в режиме, заданном по умолчанию.

Функция `setf()` является элементом класса `ios_base`. Поскольку это базовый класс для класса `ostream`, то эту функцию можно вызвать с помощью объекта `cout`. Например, при необходимости выравнивания слева выполняется такой вызов функции:

#### Листинг 16.9 Программа `setf2.cpp`.

```
// setf2.cpp - для управления форматированием используется функция setf() с двумя аргументами
#include <iostream>
using namespace std;
#include <cmath>

int main()
{
 // выполнить выравнивание слева, вывести знак "плюс",
 // показать замыкающие нули с точностью 3
 cout.setf(ios_base::left, ios_base::adjustfield);
 cout.setf(ios_base::showpos);
 cout.setf(ios_base::showpoint);
 cout.precision(3);

 // использовать степенную запись и сохранить предыдущие настройки форматирования
 ios_base::fmtflags old = cout.setf(ios_base::scientific, ios_base::floatfield);
 cout << "Left Justification:\n";
 long n;
 for (n = 1; n <= 41; n+= 10)
 {
 cout.width(4);
 cout << n << "|";
 cout.width(12);
 cout << sqrt(n) << "|\\n";
 }

 // изменить выравнивание на внутреннее
 cout.setf(ios_base::internal, ios_base::adjustfield);
 // восстановить исходный стиль вывода чисел с плавающей точкой
 cout.setf(old, ios_base::floatfield);

 cout << "Internal Justification:\n";
 for (n = 1; n <= 41; n+= 10)
 {
 cout.width(4);
 cout << n << "|";
 cout.width(12);
 cout << sqrt(n) << "|\\n";
 }

 // выполнить выравнивание справа и запись с десятичной точкой
 cout.setf(ios_base::right, ios_base::adjustfield);
 cout.setf(ios_base::fixed, ios_base::floatfield);
 cout << "Right Justification:\n";
 for (n = 1; n <= 41; n+= 10)
 {
 cout.width(4);
 cout << n << "|";
 cout.width(12);
 cout << sqrt(n) << "|\\n";
 }
}

return 0;
```

```
ios_base::fmtflags old =
 cout.setf(ios::left, ios::adjustfield);
```

Для возврата к предыдущим настройкам выполняется следующее:

```
cout.setf(old, ios::adjustfield);
```

Листинг 16.9 иллюстрирует несколько примеров использования функции `setf()` с двумя аргументами.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Эта программа использует математическую функцию, а некоторые компиляторы C++ не производят автоматического поиска математической библиотеки. Например, в некоторых разновидностях системы UNIX необходимо задать следующую команду:

```
$ CC setf2.C -lm
```

Опция `-lm` указывает редактору связей, что необходимо найти математическую библиотеку.

Результаты выполнения программы из листинга 16.9:

```
Left Justification:
+1 |+1.000e+000 |
+11 |+3.317e+000 |
+21 |+4.583e+000 |
+31 |+5.568e+000 |
+41 |+6.403e+000 |

Internal Justification:
+ 1 |+ 1.00 |
+ 11 |+ 3.32 |
+ 21 |+ 4.58 |
+ 31 |+ 5.57 |
+ 41 |+ 6.40 |

Right Justification:
+1 | +1.000 |
+11 | +3.317 |
+21 | +4.583 |
+31 | +5.568 |
+41 | +6.403 |
```

Заметьте, как точность, равная 3, приводит к тому, что выводятся на печать три цифры числа (использова-

лась для внутреннего выравнивания в программе), в то время как в записи с десятичной точкой и в степенной записи выводятся три цифры после десятичной точки.

Эффекты, вносимые функцией `setf()`, можно отменить с помощью функции `unsetf()`, имеющей следующий прототип:

```
void unsetf(fmtflags mask);
```

Здесь `mask` является битовым шаблоном. Все биты, установленные в шаблоне равными 1, очищают соответствующие биты в настройках. Это значит, что функция `setf()` устанавливает биты равными 1, а функция `unsetf()` вновь устанавливает их равными 0.

### Стандартные манипуляторы

Использование `setf()` — это не самый удобный для пользователя подход к форматированию, поэтому C++ имеет несколько манипуляторов, которые вызывают `setf()`, автоматически задавая необходимые аргументы. Ранее уже упоминались `dec`, `hex` и `oct`. Эти манипуляторы, большинство из которых не было представлено в более ранних реализациях языка, работают так же, как `hex`. Например, оператор

```
cout << left << fixed;
```

задает выравнивание слева и запись с десятичной точкой. В табл. 16.3 перечислены различные манипуляторы.

Таблица 16.3 Некоторые стандартные манипуляторы.

| Манипулятор              | Вызов                                                         |
|--------------------------|---------------------------------------------------------------|
| <code>boolalpha</code>   | <code>setf(ios_base::boolalpha)</code>                        |
| <code>noboolalpha</code> | <code>unsetf(ios_base::noboolalpha)</code>                    |
| <code>showbase</code>    | <code>setf(ios_base::showbase)</code>                         |
| <code>noshowbase</code>  | <code>unsetf(ios_base::showbase)</code>                       |
| <code>showpoint</code>   | <code>setf(ios_base::showpoint)</code>                        |
| <code>noshowpoint</code> | <code>unsetf(ios_base::showpoint)</code>                      |
| <code>showpos</code>     | <code>setf(ios_base::showpos)</code>                          |
| <code>noshowpos</code>   | <code>unsetf(ios_base::showpos)</code>                        |
| <code>uppercase</code>   | <code>setf(ios_base::uppercase)</code>                        |
| <code>nouppercase</code> | <code>unsetf(ios_base::uppercase)</code>                      |
| <code>internal</code>    | <code>setf(ios_base::internal, ios_base::adjustfield)</code>  |
| <code>left</code>        | <code>setf(ios_base::left, ios_base::adjustfield)</code>      |
| <code>right</code>       | <code>setf(ios_base::right, ios_base::adjustfield)</code>     |
| <code>dec</code>         | <code>setf(ios_base::dec, ios_base::basefield)</code>         |
| <code>hex</code>         | <code>setf(ios_base::hex, ios_base::basefield)</code>         |
| <code>oct</code>         | <code>setf(ios_base::oct, ios_base::basefield)</code>         |
| <code>fixed</code>       | <code>setf(ios_base::fixed, ios_base::floatfield)</code>      |
| <code>scientific</code>  | <code>setf(ios_base::scientific, ios_base::floatfield)</code> |

**СОВЕТ**

Если используемая система поддерживает эти манипуляторы, необходимо пользоваться их преимуществами, если нет, то в любой момент можно воспользоваться, по крайней мере, функцией `setf()`.

**Заголовочный файл `iomanip`**

Установка некоторых значений форматирования, например ширины поля, была бы сложной с использованием только инструментов класса `iostream`. C++ имеет несколько дополнительных манипуляторов в заголовочном файле `iomanip`. Они обеспечивают выполнение тех же задач, которые рассматривались ранее, но позволяют делать это более удобным с точки зрения записи способом. Вот три наиболее часто используемых манипулятора: `setprecision()` — для установки точности, `setfill()` — для установки символа-заполнителя и `setw()` — для установки ширины поля. В отличие от манипуляторов, о которых шла речь ранее, эти манипуляторы имеют аргументы. Манипулятор `setprecision()` имеет аргумент целого типа, задающий точность, `setfill()` имеет аргумент типа `char`, являющийся символом заполнения, а `setw()` — аргумент, задающий ширину поля. Поскольку они являются манипуляторами, их можно конкатенировать в операторе вывода `cout`. Это делает манипулятор `setw()` удобным при выводе нескольких колонок значений. Листинг 16.10 иллюстрирует это изменением ширины поля и символа заполнения для выводимой строки. Кроме того, в нем используются некоторые более новые стандартные манипуляторы.

**Листинг 16.10 Программа `iomanip.cpp`.**

```
// iomanip.cpp - использование манипуляторов
// из заголовочного файла iomanip
// В некоторых системах необходимо явно
// связать математическую библиотеку

#include <iostream>
using namespace std;
#include <iomanip>
#include <cmath>

int main()
{
 //использование новых стандартных манипуляторов
 cout << showpoint << fixed << right;

 //использование манипуляторов из
 //заголовочного файла iomanip
 cout << setw(6) << "N" << setw(14)
 << "square root"
 << setw(15) << "fourth root\n";

 double root;
 for (int n = 10; n <= 100; n += 10)
 {
 root = sqrt(n);
 cout << setfill('0') << setw(6) << setprecision(3)
 << root << endl;
 }
}
```

```
cout << setw(6) << setfill('.') << n
<< setfill(' ') << setw(12)
<< setprecision(3) << root
<< setw(14) << setprecision(4)
<< sqrt(root) << "\n";
}
```

```
}
return 0;
```

Результаты выполнения программы:

| N      | square root | fourth root |
|--------|-------------|-------------|
| ....10 | 3.162       | 1.7783      |
| ....20 | 4.472       | 2.1147      |
| ....30 | 5.477       | 2.3403      |
| ....40 | 6.325       | 2.5149      |
| ....50 | 7.071       | 2.6591      |
| ....60 | 7.746       | 2.7832      |
| ....70 | 8.367       | 2.8925      |
| ....80 | 8.944       | 2.9907      |
| ....90 | 9.487       | 3.0801      |
| ...100 | 10.000      | 3.1623      |

**ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**

Эта программа использует математическую функцию, а некоторые компиляторы C++ не производят автоматического поиска математической библиотеки. Например, в некоторых разновидностях системы UNIX необходимо задать следующую команду:

```
$ CC setf2.C -lm
```

Опция `-lm` указывает редактору связей, что необходимо найти математическую библиотеку. Кроме того, старые компиляторы могут не распознавать такие новые стандартные манипуляторы, как `showpoint`. В этом случае необходимо пользоваться эквивалентом в виде функции `setf()`.

Теперь можно обеспечить вывод с аккуратными колонками. Заметьте, что эта программа производит одинаковое форматирование и для предыдущих, и для текущих реализаций языка. Использование манипулятора `showpoint` приводит к выводу замыкающих нулей в старой реализации, а в новой реализации тот же эффект достигается с помощью манипулятора `fixed`. Использование манипулятора `fixed` заставляет в обеих реализациях выводить числа с плавающей точкой в записи с десятичной точкой, а в текущей реализации это приводит еще и к тому, что точностью считается количество цифр после десятичной точки. В более ранних системах точность всегда имела такой смысл, независимо от режима вывода чисел с плавающей точкой.

В табл. 16.4 показаны некоторые из различий между более ранним форматированием C++ и текущей реализацией. Вывод, который можно сделать, просмотрев эту таблицу, состоит в том, что не нужно удивляться, если при запуске программы, увиденной где-то, форматирование вывода получается другим.

Таблица 16.4 Изменения в форматировании.

| Свойство                                                 | Более ранняя реализация C++                                | Текущий C++                                                                                                                                              |
|----------------------------------------------------------|------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>precision(n)</code>                                | Отображается $n$ цифр после десятичной точки               | Отображается всего $n$ цифр в режиме, заданном по умолчанию, и $n$ цифр после десятичной точки в режиме с фиксированной точкой и в экспоненциальном виде |
| <code>ios::showpoint</code>                              | Отображаются замыкающая десятичная точка и замыкающие нули | Отображается замыкающая десятичная точка                                                                                                                 |
| <code>ios::fixed,</code><br><code>ios::scientific</code> |                                                            | Отображаются замыкающие нули<br>(см. комментарий к <code>precision()</code> )                                                                            |

## Ввод данных с помощью `cin`

Теперь пришло время заняться вводом данных и рассмотреть, как программа получает данные. Объект `cin` представляет стандартный ввод как поток байтов. Обычно поток символов вводится с клавиатуры. Если ввести последовательность символов 1998, объект `cin` извлечет эти символы из входного потока. Вводимые данные могут быть частью строки, значением типа `int`, `float` или любого другого типа. Поэтому извлечение символов требует также и преобразования типов. Объект `cin`, управляемый типом переменной, которая указана для получения значения, должен использовать свои методы для преобразования этой последовательности символов в значение требуемого типа.

Обычно `cin` используется следующим образом:

```
cin >> value_holder;
```

Здесь переменная `value_holder` указывает на область в памяти, где необходимо сохранить введенное значение. Это может быть имя переменной, указатель, разыменованный указатель либо элемент структуры или класса. Как `cin` интерпретирует ввод, зависит от типа переменной `value_holder`. Класс `istream`, определенный в заголовочном файле `iostream`, переопределяет оператор извлечения `>>` так, чтобы он распознавал следующие базовые типы:

- `signed char &`
- `unsigned char &`
- `char &`
- `short &`
- `unsigned short &`
- `int &`
- `unsigned int &`
- `long &`
- `unsigned long &`
- `float &`
- `double &`
- `long double &`

Эти функции называются функциями форматированного ввода, поскольку они преобразуют введенное значение в тип, определяемый переменной назначения.

Типичная операторная функция имеет следующий прототип:

```
istream & operator>>(int &);
```

И аргумент, и возвращаемое значение являются указателями. Аргумент-указатель (см. главу 9) означает, что оператор наподобие

```
cin >> staff_size;
```

заставляет функцию `operator>>()` работать непосредственно с переменной `staff_size`, а не с ее копией, как происходит в случае с обычным аргументом. Поскольку тип аргумента — указатель, объект `cin` имеет возможность напрямую изменить значение переменной. В выражении, приведенном выше, изменяется значение переменной `staff_size`. Вскоре мы поговорим о важности того, что возвращаемое значение также является ссылкой. Сначала рассмотрим аспект преобразования типов оператора извлечения. Для аргумента каждого из приведенных ранее типов операция извлечения преобразует символьный ввод в указанный тип значения. Например, пусть переменная `staff_size` имеет тип `int`. Тогда компилятор заменяет выражение

```
cin >> staff_size;
```

следующим прототипом:

```
istream & operator>>(int &);
```

Функция, отвечающая этому прототипу, читает поток символов, который пересыпается программе, например, символы 2, 3, 1, 8 и 4. Для систем, использующих 2-байтовый тип `int`, функция затем преобразует эти символы в 2-байтовое двоичное представление целого числа 23184. Если в другом случае переменная `staff_size` имеет тип `double`, то `cin` использует функцию `operator>>(double &)` для преобразования такого же самого ввода в 8-байтовое представление числа с плавающей точкой 23184.0.

В связи с этим можно использовать манипуляторы `hex`, `oct` и `dec` вместе с объектом `cin` для указания того, что целочисленный ввод нужно интерпретировать как шестнадцатиричный, восьмеричный или десятичный. Например, оператор

```
cin >> hex;
```

приводит к тому, что результат ввода числа **12** или **0x12** будет восприниматься как шестнадцатиричное число **12** или десятичное число **18**, а **ff** или **FF** будет восприниматься как десятичное число **255**.

Кроме того, класс `istream` перегружает операцию извлечения `>>` для следующих типов символьных указателей:

- `signed char *`
- `char *`
- `unsigned char *`

Для такого типа аргумента оператор извлечения читает следующее слово из потока ввода и помещает его по указанному адресу, добавляя нулевой символ для того, чтобы сформировать строку. Например, рассмотрим следующий программный код:

```
cout << "Enter your first name:\n";
char name[20];
cin >> name;
```

В ответ на запрос ввести **Hilary** операция извлечения поместит символы **Hilary\0** в массив `name`. (Как обычно, **\0** представляет собой нулевой символ конца строки.) Идентификатор `name`, будучи именем массива типа `char`, действует как адрес первого элемента массива, задавая тип `name` как `char *` (указатель-на-тип-`char`).

Тот факт, что каждая операция извлечения возвращает указатель на объект, для которого он был вызван, позволяет конкатенировать ввод так же, как это делалось для вывода:

```
char name[20];
float fee;
int group;
cin >> name >> fee >> group;
```

Здесь, например, объект `cin`, возвращаемый операцией `cin >> name`, становится объектом, который обрабатывает переменную `fee`.

## Как `cin >>` рассматривает поток ввода

Различные версии операции извлечения имеют общий способ рассмотрения потока ввода. Они пропускают управляющие символы (пробелы, символы новой строки и символы табуляции) до тех пор, пока не будет обнаружен непустой символ. Это справедливо даже для режимов ввода одного символа (таких, для которых аргументом является тип `char`, `unsigned char` или `signed char`), но несправедливо для функций символьного вво-

да С (рис. 16.5). В односимвольных режимах операция `>>` считывает символ и помещает его в указанное место. В других режимах эта операция читает один элемент указанного типа, т.е. читается все от первого непустого символа до первого символа, который не подходит типу назначения.

Например, рассмотрим следующий программный код:

```
int elevation;
cin >> elevation;
```

Пусть введены следующие символы:

```
-123Z
```

Оператор прочитает символы **-**, **1**, **2** и **3**, поскольку они соответствуют типу целого числа. А символ **Z** не является таковым, поэтому последним принятым для ввода символом будет **3**. Символ **Z** останется в потоке ввода, и следующий оператор `cin` начнет чтение из этой точки. Тем временем оператор преобразует последовательность символов **-123** в целое число и присвоит это значение переменной `elevation`.

Может получиться, что ввод не отвечает ожиданиям программы, например, если вместо **-123Z** был введен символ **Z**. В этом случае операция извлечения оставляет значение переменной `elevation` неизменным и возвращает значение нуль. (Используя терминологию технического языка, можно сказать, что оператор `if` или `while` обрабатывает объект `istream` как `false`, если у объекта установлено состояние ошибки. Все это мы обсудим более подробно далее в этой главе.) Возвращаемое значение `false` позволяет программе проверить, удовлетворяет ли ввод ее требованиям, как показано в листинге 16.11.

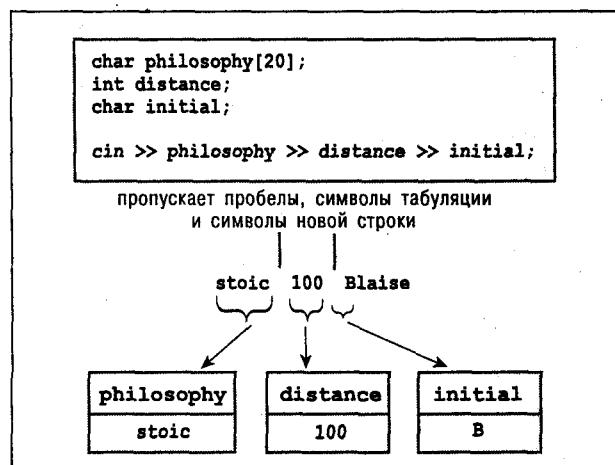


РИС. 16.5 Оператор `cin >>` пропускает служебные символы.

**Листинг 16.11 Программа check\_it.cpp.**

```
#include <iostream>
using namespace std;

int main()
{
 cout.precision(2);
 cout << showpoint << fixed;
 cout << "Enter numbers: ";

 double sum = 0.0;
 double input;
 while (cin >> input)
 {
 sum += input;
 }

 cout << "Last value entered = "
 << input << "\n";
 cout << "Sum = " << sum << "\n";
 return 0;
}
```

**ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ**

Если используемый компилятор не поддерживает манипуляторы `showpoint` и `fixed`, необходимо пользоваться эквивалентами функции `setf()`.

Результаты выполнения программы в случае, когда в поток ввода попали неподходящие символы (-123Z):

```
Enter numbers: 200.0
1.0E1 -50 -123Z 60
Last value entered = -123.00
Sum = 37.00
```

Поскольку ввод буферизован, вторая строка ввода с клавиатуры не пересыпается программе до тех пор, пока не будет нажата клавиша `<Enter>` в конце строки. Но цикл прекратит обработку потока ввода при достижении символа `Z`, поскольку он не подходит для формата чисел с плавающей точкой. Несовпадение ввода с ожидаемым форматом, в свою очередь, приводит к тому, что выражение `cin >> input` принимает значение `false` и цикл `while` прерывается.

**Состояние потока**

Теперь мы подробнее рассмотрим, что происходит при вводе неподходящих символов. Объекты `cin` и `cout` содержат элемент данных (наследуемый от класса `ios_base`), описывающий *состояние потока*.

Состояние потока (определенное как тип `iosstate`, который, в свою очередь, является типом битовой маски, наподобие рассмотренного ранее) состоит из трех элементов класса `ios_base`: `eofbit`, `badbit` и `failbit`. Каждый элемент является отдельным битом, который может принимать значение 1 (*установлен*) или 0 (*не установлен*). Когда при выполнении чтения с помощью `cin` достигается конец файла, устанавливается бит `eofbit`. Если при выполнении операции `cin` не может прочесть ожидаемый

символ, как в приведенном выше примере, устанавливается бит `failbit`. Такие действия ввода/вывода, как попытка чтения недоступного файла или попытка записи на защищенную дискету, также приводят к установке значения бита `failbit` равным 1. Элемент `badbit` установлен тогда, когда в потоке произошла какая-либо недиагностируемая ошибка. (В реализациях не всегда согласовано, какие события устанавливают `failbit`, а какие — `badbit`.) Если все три из этих битов состояния установлены равными 0, значит, все происходит благополучно. В программах может быть встроена проверка состояния потока для того, чтобы решить, что делать дальше. В табл. 16.5 перечислены эти биты вместе с некоторыми методами класса `ios_base` для отчета или изменения состояния потока. (Более ранние компиляторы не поддерживают два метода `exceptions()`.)

**Установка состояния**

Два метода из приведенных в табл. 16.5, `clear()` и `setstate()`, похожи. Оба переустанавливают состояние, но различным образом. Метод `clear()` устанавливает состояние по своему аргументу. Поэтому в результате вызова

```
clear();
```

используется аргумент, заданный по умолчанию (равный 0), который очищает все три бита состояния (`eofbit`, `badbit` и `failbit`). Аналогично в результате вызова

```
clear(eofbit);
```

состояние задается битом `eofbit`; т.е. `eofbit` установлен, а два других бита очищены.

Однако метод `setstate()` влияет только на биты, которые заданы в его аргументе. Поэтому в результате вызова

```
setstate(eofbit);
```

устанавливается бит `eofbit`, но не затрагиваются другие биты. Если бит `failbit` установлен, то он и остается таким же.

Почему возникает необходимость переустановить состояние потока? Для программиста наиболее общей причиной для использования функции `clear()` без аргументов может послужить необходимость заново открыть поток ввода после того, как состоялся неверный ввод или был достигнут конец файла. Нужно это делать или нет, в большой степени зависит от того, что программа пытается выполнить. Вскоре мы рассмотрим некоторые примеры. Главная цель функции `setstate()` — обеспечить функциям ввода/вывода метод для изменения состояния потока. Например, если переменная `num` имеет тип `int`, в результате вызова

```
cin >> num; // считать значение типа int
```

Таблица 16.5 Состояние потока.

| Элемент                       | Описание                                                                                                                                                                                                                                                 |
|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>eofbit</b>                 | Установлен равным 1, если достигнут конец файла.                                                                                                                                                                                                         |
| <b>badbit</b>                 | Установлен равным 1, если поток поврежден; например, при ошибке чтения из файла.                                                                                                                                                                         |
| <b>failbit</b>                | Установлен равным 1, если при выполнении операции ввода был считан неподходящий символ или если при выполнении операции вывода не был отображен ожидаемый символ.                                                                                        |
| <b>goodbit</b>                | Еще один способ установить 0.                                                                                                                                                                                                                            |
| <b>good()</b>                 | Возвращает значение <b>true</b> , если поток можно использовать (все биты сброшены).                                                                                                                                                                     |
| <b>eof()</b>                  | Возвращает значение <b>true</b> , если установлен бит <b>eofbit</b> .                                                                                                                                                                                    |
| <b>bad()</b>                  | Возвращает значение <b>true</b> , если установлен бит <b>badbit</b> .                                                                                                                                                                                    |
| <b>fail()</b>                 | Возвращает значение <b>true</b> , если установлен бит <b>badbit</b> или <b>failbit</b> .                                                                                                                                                                 |
| <b>rdstate()</b>              | Возвращает состояние потока.                                                                                                                                                                                                                             |
| <b>exceptions()</b>           | Возвращает битовую маску, указывающую, какие флаги отвечают за прерывание.                                                                                                                                                                               |
| <b>exceptions(iostate ex)</b> | Устанавливает, какие состояния приведут к тому, что функция <b>clear()</b> будет генерировать прерывание; например, если <b>ex</b> — это <b>eofbit</b> , тогда функция <b>clear()</b> будет генерировать прерывание, если установлен бит <b>eofbit</b> . |
| <b>clear(iostate s)</b>       | Устанавливает состояние потока равным <b>s</b> ; состояние <b>s</b> по умолчанию определено как 0 ( <b>goodbit</b> ); генерирует прерывание <b>basic_ios::failure</b> , если <b>rdstate() &amp; exceptions() != 0</b> .                                  |
| <b>setstate (iostate s)</b>   | Вызывает функцию <b>clear(rdstate()   s)</b> . При этом устанавливаются те биты состояния потока, значение которых соответствует <b>s</b> ; другие биты состояния остаются без изменений.                                                                |

функция **operator>>(int &)** использует функцию **setstate()** для установки бита **failbit** или **eofbit**.

### Ввод/вывод и прерывания

Представим себе, что функция ввода устанавливает бит **eofbit**. Будет ли при этом генерироваться прерывание? По умолчанию нет. Однако можно воспользоваться методом **exceptions()** для управления прерываниями.

Сначала обратимся к базовым знаниям. Метод **exceptions()** возвращает битовый шаблон с тремя битами, отвечающими **eofbit**, **failbit** и **badbit**. Изменение состояния потока достигается либо с помощью функции **clear()**, либо с помощью функции **setstate()**, использующей **clear()**. После изменения состояния потока метод **clear()** сравнивает текущее состояние потока со значением, которое возвращает метод **exceptions()**. Если бит возвращаемом значении был установлен, а также установлен соответствующий бит для текущего состояния, то **clear()** генерирует прерывание **basic\_ios::failure**. Это произойдет, например, если для обоих значений установлен бит **badbit**. Следовательно, если функция **exceptions()** возвращает **goodbit**, то никакие прерывания не генерируются.

Установка, заданная по умолчанию для функции **exceptions()** как раз и соответствует биту **goodbit**, т.е. прерываний не происходит. Однако управлять этим поведением позволяет перегруженная функция **exceptions(iostate)**:

```
cin.exceptions(badbit); //установка бита badbit
//приводит к возникновению прерывания
```

Поразрядный оператор ИЛИ () (см. приложение Е) позволяет задать несколько битов. Например, оператор

```
cin.exceptions(badbit | eofbit);
```

приводит к тому, что прерывание генерируется в случае, если установлен либо бит **badbit**, либо бит **eofbit**.

### Эффекты состояния потока

Оператор **if** или **while** в выражении

```
while (cin >> input)
```

дает значение **true**, если поток имеет хорошее состояние (все биты сброшены). Если проверка не проходит, можно использовать функции-элементы, приведенные в табл. 16.5, чтобы выявить возможную причину этого. Например, можно изменить центральную часть программы из листинга 16.11 следующим образом:

```
while (cin >> input)
{
 sum += input;
}
if (cin.eof())
 cout << "Loop terminated because
 EOF encountered\n";
```

Установка бита состояния потока имеет очень важное следствие: поток закрыт для последующей записи или чтения до тех пор, пока бит не будет сброшен. Например, следующий код неверный:

```

while (cin >> input)
{
 sum += input;
}
cout << "Last value entered = "
 << input << "\n";
cout << "Sum = " << sum << "\n";
cout << "Now enter a new number: ";
cin >> input; // не работает

```

Если нужно, чтобы программа считывала последующий ввод после того, как было установлено состояние потока, необходимо его вновь установить с помощью правильного значения. Это можно сделать, вызвав метод `clear()`:

```

while (cin >> input)
{
 sum += input;
}
cout << "Last value entered = "
 << input << "\n";
cout << "Sum = " << sum << "\n";
cout << "Now enter a new number: ";
cin.clear(); // инициализировать состояние потока
while (!isspace(cin.get()))
 continue; // избавиться от результатов
 // неверного ввода
cin >> input; // теперь это работает

```

Заметьте, что этого недостаточно для инициализации состояния потока. Символ при прерывании цикла ввода все еще находится во входной очереди, и программе необходимо пропустить его. Один из способов, позволяющих это сделать, — продолжать считывание символов до тех пор, пока не будет достигнут управляющий символ. Функция `isspace()` (см. главу 6) — это функция типа `cctype`, которая возвращает значение `true`, если ее аргумент является управляющим символом. Или же можно пропустить не только следующее слово, но и всю строку:

```

while (cin.get() != '\n')
 continue; // пропуск строки

```

В этом примере предполагается, что цикл прервался из-за несоответствующего ввода. Предположим теперь, что цикл прервался из-за достижения конца файла или аппаратной ошибки. Тогда новый код, избавляющий от результатов неверного ввода, становится бессмысленным. Это можно исправить, воспользовавшись методом `fail()` для проверки правильности этого предположения. Поскольку `fail()` возвращает значение `true`, если бит `failbit` или `badbit` установлен, в программе необходимо исключить последнюю возможность.

```

while (cin >> input)
{
 sum += input;
}
cout << "Last value entered = "
 << input << "\n";
cout << "Sum = " << sum << "\n";
cout << "Now enter a new number: ";

```

```

if (cin.fail() && !cin.bad()) // ошибка
 // при неверном вводе
{
 // переустановить состояние потока
 cin.clear();
 while (!isspace(cin.get()))
 continue; // избавиться от
 // неверного ввода
}
else // иначе выйти из программы
{
 cout << "I cannot go on!\n";
 exit(1);
}
cout << "Now enter a new number: ";
cin >> input; // теперь это работает

```

## Другие методы класса `istream`

В главах 3, 4 и 5 рассматривались методы `get()` и `getline()`. Вспомните, что они обеспечивают следующие дополнительные возможности при вводе:

- Методы `get(char &)` и `get(void)` обеспечивают ввод одного символа и не пропускают служебные символы.
- Функции `get(char *, int, char)` и `getline(char *, int, char)` читают все содержимое строки, заданное по умолчанию.

Эти функции называются *функциями неформатированного ввода*, поскольку они просто считывают символьный поток ввода, не пропуская служебные символы и не выполняя преобразования типов данных.

Рассмотрим две группы таких методов класса `istream`.

### Односимвольный ввод

Метод `get()`, используемый с аргументом типа `char` или без аргумента вообще, считывает следующий введенный символ, даже если это пробел — символ табуляции или символ новой строки. Версия `get(char & ch)` присваивает введенный символ своему аргументу, а версия `get(void)` использует введенный символ, преобразованный в целый тип, обычно `int`, как возвращаемое значение.

Сначала рассмотрим функцию `get(char &)`. Предположим, что в программе имеется следующий цикл:

```

int ct = 0;
char ch;
cin.get(ch);
while (ch != '\n')
{
 cout << ch;
 ct++;
 cin.get(ch);
}
cout << ct << '\n';

```

Далее предположим, что произведен следующий ввод:

**I C++ clearly.<Enter>**

При нажатии клавиши **Enter** введенная строка посылается программе. Фрагмент программы сначала прочитает символ **I**, отобразит его с помощью **cout** и увеличит значение **ct** до 1. Затем он прочтет пробел, следующий за символом **I**, выведет его и увеличит значение **ct** до 2. Этот процесс будет продолжаться до тех пор, пока программа не обработает нажатие клавиши **Enter** как символ новой строки и не прервет цикл. Главное здесь то, что использование функции **get(ch)** позволяет читать, выводить и подсчитывать пробелы наравне с другими символами.

Предположим, что теперь в программе будет использована операция **>>**:

```
int ct = 0;
char ch;
cin >> ch;
while (ch != '\n') // НЕ РАБОТАЕТ
{
 cout << ch;
 ct++;
 cin >> ch;
}
cout << ct << '\n';
```

Прежде всего, код пропустит пробелы, не подсчитывая их и сжав фразу до:

**IC++clearly.**

Хуже то, что цикл никогда не закончится! Поскольку операция извлечения пропускает символ новой строки, это значение никогда не сможет быть присвоено переменной **ch** и в ходе проверки условия цикла эта ситуация не изменяется.

Функция-элемент **get(char &)** возвращает указатель на объект класса **istream**, для которого она была вызвана. Это значит, что можно конкатенировать операции извлечения, следующие за функцией **get(char &)**:

```
char c1, c2, c3;
cin.get(c1).get(c2) >> c3;
```

Вначале **cin.get(c1)** присваивает значение первого введенного символа переменной **c1** и возвращает собственный объект, которым является **cin**. При этом код сокращается до оператора **cin.get(c2) >> c3**, который присваивает второй введенный символ переменной **c2**. При вызове функции вновь возвращается объект **cin** и код сокращается до **cin >> c3**. При этом следующий непустой символ присваивается переменной **c3**. Заметьте, что переменным **c1** и **c2** могут быть присвоены пустые символы, а переменной **c3** — нет.

Если функция **cin.get(char &)** достигает конца файла, или реального, или эмулируемого с клавиатуры (**Ctrl+Z** для DOS, **Ctrl+D** в начале строки для UNIX), она

не присваивает значения своему аргументу. Это вполне разумно, поскольку, раз программа достигла конца файла, следовательно, больше нет значений для присвоения. Более того, метод вызывает функцию **setstate(failbit)**, которая заставляет объект **cin** возвращать значение **false**:

```
char ch;
while (cin.get(ch))
{
 // обработка результатов ввода
}
```

До тех пор пока ввод является правильным, функция **cin.get(ch)** возвращает объект **cin**, который рассматривается как значение **true**, поэтому цикл продолжается. При достижении конца файла возвращаемое значение становится равным **false** и цикл прерывается.

Функция-элемент **get(void)** также считывает управляющие символы, но для передачи ввода программе она использует возвращаемое ею значение. Поэтому ее можно применить следующим образом:

```
int ct = 0;
char ch;
ch = cin.get(); // использование
 // возвращаемого значения
while (ch != '\n')
{
 cout << ch;
 ct++;
 ch = cin.get();
}
cout << ct << '\n';
```

Некоторые более ранние реализации функций в C++ не имеют именно этой функции-элемента.

Функция-элемент **get(void)** возвращает значение типа **int** (или какой-либо больший целый тип, в зависимости от кодировки и локальных настроек). Поэтому следующим код является нерабочим:

```
char c1, c2, c3;
cin.get().get() >> c3; // не работает
```

Здесь функция **cin.get()** возвращает значение типа **int**. Поскольку это возвращаемое значение не является объектом класса, то для него нельзя применить оператор класса, иначе это приведет к синтаксической ошибке при компиляции. Однако функцию **get()** можно использовать в конце последовательности операций извлечения:

```
char c1;
cin.get(c1).get(); // работает
```

Тот факт, что функция **get(void)** возвращает значение типа **int**, свидетельствует о том, что за ней не может следовать операция извлечения. Но поскольку функция **cin.get(c1)** возвращает объект **cin**, она служит разрешенным префиксом для **get()**. Данный код прочтет первый

символ, присвоит его переменной `c1`, затем прочтет второй символ и отбросит его.

При достижении конца файла, реального или эмулированного, функция `cin.get(void)` возвращает значение `EOF`, которое является символьской константой, определенной в заголовочном файле `iostream`. Это свойство позволяет применять следующий код для чтения результатов ввода:

```
int ch;
while ((ch = cin.get()) != EOF)
{
 // обработка ввода
}
```

Для переменной `ch` необходимо указать тип `int` вместо `char`, так как значение `EOF` не может быть представлено типом `char`.

В главе 5 эти функции описаны более детально; в табл. 16.6 представлены свойства функций односимвольного ввода.

### Выбор формы односимвольного ввода

Имея функции `>>`, `get(char &)` и `get(void)`, как определить, какую нужно использовать? Прежде всего необходимо решить, будут ли при вводе пропускаться служебные символы или нет. Если пропуск служебного символа вас устраивает, то необходимо воспользоваться операцией извлечения `>>`. Например, пропуск служебного символа удобен при выборе пунктов меню:

```
cout << "a. annoy client b. bill client\n"
 << "c. calm client d. deceive client\n"
 << "q. \n";
cout << "Enter a, b, c, d, or q: ";
char ch;
cin >> ch;
while (ch != 'q')
{
 switch(ch)
 {
 ...
 }
 cout << "Enter a, b, c, d, or q: ";
 cin >> ch;
}
```

Чтобы ввести, например, вариант выбора `b`, необходимо ввести `b` и нажать клавишу `Enter`, производя, таким образом, двусимвольный ввод `b\n`. Если воспользоваться формой `get()`, то нужно добавить код для обработки символа `\n` на каждой итерации цикла, в то время как операция извлечения пропускает его. (Если вы программирували на языке С, то, вероятно, сталкивались с ситуацией, когда появление символа новой строки рассматривается в качестве результата неверного ввода. Эту проблему легко решить, но она представляет собой существенное неудобство.)

Если нужно, чтобы программа получала каждый символ, необходимо воспользоваться одним из методов `get()`. Например, программа подсчета слов использует пустой символ, чтобы определить конец слова. Из двух методов `get()` метод `get(char &)` имеет более удобный интерфейс. Наибольшее достоинство метода `get(void)` состоит в том, что он имеет близкое сходство со стандартной функцией С `getchar()` и позволяет легко преобразовать программу, написанную на С, в программу на C++. Для этого потребуется включить заголовочный файл `iostream` вместо `stdio.h` и заменить во всем тексте функцию `getchar()` на `cin.get()`, а `putchar(ch)` — на `cout.put(ch)`.

### Строковый ввод: `getline()`, `get()` и `ignore()`

Далее мы рассмотрим функции строкового ввода, представленные в главе 4. Функция-элемент `getline()` и третья версия функции `get()` могут считывать строки, и обе имеют одинаковую сигнатуру (упрощенную здесь по сравнению с более общим объявлением шаблона):

```
istream & get(char *, int, char = '\n');
istream & getline(char *, int, char = '\n');
```

Вспомните, что первый аргумент — это адрес в памяти, по которому необходимо разместить введенную строку. Второй аргумент — это число, на единицу большее, чем максимальное количество символов, которое нужно прочитать. (Дополнительный символ оставлен для нулевого символа, чтобы сохранить ввод как строку.) Если опустить третий аргумент, каждая функция будет считывать вводимые символы, пока не будет достигнуто необходимое количество символов или пока не встретится символ новой строки.

Таблица 16.6 Функции `cin.get(ch)` и `cin.get()`.

| Свойство                                                   | <code>cin.get(ch)</code>                        | <code>ch = cin.get()</code>                                                                    |
|------------------------------------------------------------|-------------------------------------------------|------------------------------------------------------------------------------------------------|
| Метод, используемый для передачи символа                   | Присвоить аргументу <code>ch</code>             | Задает использование значения, возвращаемое функцией для присвоения переменной <code>ch</code> |
| Значение, возвращаемое функцией при символьном вводе       | Указатель на объект класса <code>istream</code> | Код символа, заданный типом <code>int</code>                                                   |
| Значение, возвращаемое функцией при достижении конца файла | Значение <code>false</code>                     | <code>EOF</code>                                                                               |

Например, код

```
char line[50];
cin.get(line, 50);
```

считывает в символьный массив `line`. Функция `cin.get()` прекратит чтение при достижении 49 символов или, по умолчанию, при прочтении символа новой строки, в зависимости от того, что наступит раньше. Основное различие между `get()` и `getline()` заключается в том, что функция `get()` оставляет символ новой строки в потоке, делая его первым символом для последующей операции чтения, в то время как функция `getline()` извлекает символ новой строки из потока ввода и пропускает его.

В главе 4 рассказывается об использовании двух форм этих функций-элементов, заданных по умолчанию. Сейчас мы ознакомимся с третьим аргументом, который изменяет поведение функции, заданное по умолчанию. Третий аргумент, который имеет принятое по умолчанию значение '\n', — это символ завершения строки. Прочтение символа завершения строки заставляет функцию обработки ввода остановиться, даже если не было достигнуто максимальное количество символов. Поэтому по умолчанию оба метода прекращают считывание потока входных данных при достижении символа новой строки до того, как считывается указанное количество символов. Как и в случае, определенном по умолчанию,

функция `get()` оставляет последний символ во входной очереди, а функция `getline()` — нет.

Листинг 16.12 демонстрирует, как работают функции `getline()` и `get()`. Там же представлена и функция `ignore()`. Она имеет два аргумента: число, задающее количество символов для чтения, и символ, действующий как символ завершения строки. Например, в результате вызова функции

```
cin.ignore(80, '\n');
```

считываются и отбрасываются либо следующие 80 символов, либо все символы до первого встретившегося символа новой строки. Прототип задает значения по умолчанию 1 и EOF для двух аргументов, и функция возвращает значение типа `istream &`:

```
istream & ignore(int = 1, int = EOF);
```

В результате вызова функции возвращается объект. Это позволяет конкатенировать вызовы функций следующим образом:

```
cin.ignore(80, '\n').ignore(80, '\n');
```

Здесь в результате первого вызова метода `ignore()` считывается и пропускается одна строка, а в результате второго вызова — вторая строка. В результате выполнения двух вызовов пропускаются две строки.

Теперь рассмотрим листинг 16.12.

### Листинг 16.12 Программа `get_fun.cpp`.

```
// get_fun.cpp - использование функций get() и getline()
#include <iostream>
using namespace std;
const int Limit = 80;

int main()
{
 char input[Limit];
 cout << "Enter a string for getline() processing:\n";
 cin.getline(input, Limit, '#');
 cout << "Here is your input:\n";
 cout << input << "\nDone with phase 1\n";

 char ch;
 cin.get(ch);
 cout << "The next input character is " << ch << "\n";

 if (ch != '\n')
 cin.ignore(Limit, '\n'); // пропустить до конца строки

 cout << "Enter a string for get() processing:\n";
 cin.get(input, Limit, '#');
 cout << "Here is your input:\n";
 cout << input << "\nDone with phase 2\n";

 cin.get(ch);
 cout << "The next input character is " << ch << "\n";
 return 0;
}
```

## ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Версия функции `getline()` класса `iostream` в Microsoft Visual C++ 5.0 содержит ошибку, которая приводит к тому, что вывод следующей строки задерживается до тех пор, пока не будут введены данные, запрашиваемые еще не выведенной строкой. Однако версия из файла `iostream.h` работает правильно.

Результаты выполнения программы из листинга 16.12:

```
Enter a string for getline() processing:
Please pass
me a #3 melon!
Here is your input:
Please pass
me a
Done with phase 1
The next input character is 3
Enter a string for get() processing:
I still
want my #3 melon!
Here is your input:
I still
want my
Done with phase 2
The next input character is #
```

Заметьте, что функция `getline()` отбрасывает символ завершения строки `#` при вводе, а функция `get()` не делает этого.

### Неожиданный строковый ввод

Некоторые формы ввода для функций `get(char *, int)` и `getline()` влияют на состояние потока. Как и в случае с более ранними функциями, при достижении конца файла устанавливается бит `eofbit`, а все, что делает поток непригодным для дальнейшей работы, например ошибка устройства, приводит к установке бита `badbit`. Два других специальных случая — это отсутствие ввода и ввод количества символов, превышающего или равного указанному, при вызове функции. Сейчас мы рассмотрим эти случаи.

Если любой из двух методов не выполняет извлечение символов, то в результате этого помещается нулевой символ в строку ввода, а затем используется функция `setstate()` для установки бита `failbit`. (В более ранних реализациях C++ `failbit` в такой ситуации не устанавливается.) В каком же случае метод не может прочитать символы? Одна из возможностей заключается в том, что метод сразу же достигает конца файла. Для функции `get(char *, int)` вторая возможность состоит в вводе пустой строки:

```
char temp[80];
while (cin.get(temp, 80)) // прерывается при
// вводе пустой строки
...
```

Интересно, что пустая строка не заставляет функцию `getline()` устанавливать бит `failbit`, поскольку `getline()` все равно извлекает символ новой строки, хотя и не сохраняет его. Если нужно, чтобы цикл с функцией `getline()` мог прерваться при вводе пустой строки, его можно написать следующим образом:

```
char temp[80];
while (cin.getline(temp, 80) &&
 temp[0] != '\0') // прерывается при
// вводе пустой строки
```

Теперь предположим, что количество символов во входной очереди равно или превышает количество символов, указанных при вызове метода. Сначала рассмотрим функцию `getline()` и следующий фрагмент кода:

```
char temp[30];
while (cin.getline(temp, 30))
```

Метод `getline()` считывает последовательность символов из входной очереди, помещая их в массив `temp` до тех пор, пока (в целях тестирования) не будет достигнут символ `EOF`, символ новой строки или же пока не будет сохранено 29 символов. Если встречается признак конца файла `EOF`, то будет установлен бит `eofbit`. Если следующий считываемый символ является символом новой строки, он будет отброшен. И если считано 29 символов, то будет установлен бит `failbit`, при условии, что следующим символом не является символ новой строки. Таким образом, строка из 30 или более символов приведет к прекращению ввода.

Теперь рассмотрим метод `get(char *, int)`. Сначала он проверяет количество символов, затем определяет признак конца файла и затем символ новой строки. Он не устанавливает флаг `failbit` при прочтении максимального количества символов. Тем не менее, ему можно указать, после прочтения скольких символов необходимо прекратить чтение. Можно воспользоваться функцией `peek()` (см. следующий раздел для проверки следующего, еще не прочитанного символа). Если это символ новой строки, тогда метод `get()` должен прочесть полную строку. Если это не символ новой строки, то `get()` должен остановиться до того, как будет достигнут конец строки. Этот метод не обязательно работает с функцией `getline()`. Поскольку `getline()` читает и отбрасывает символ новой строки, просмотр следующего символа ничего не даст. Но если используется метод `get()`, есть возможность предпринять какое-либо действие, если читается меньше символов, чем содержит полная строка. В следующем разделе описываются примеры такого подхода. В табл. 16.7 показаны некоторые различия между более ранними методами ввода C++ и текущим стандартом.

Таблица 16.7 Изменения в поведении методов ввода.

| Метод                   | Ранний C++                                                                                                                                                    | Текущий C++                                                                                                                                                                                                                                     |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>getline()</b>        | Не устанавливает бит <b>failbit</b> , если не считаны никакие символы.<br><br>Не устанавливает бит <b>failbit</b> , если считано максимальное число символов. | Устанавливает бит <b>failbit</b> , если не прочитано никаких символов (но символ новой строки считается прочитанным).<br><br>Устанавливает бит <b>failbit</b> , если прочитано максимальное число символов, а часть символов осталась в строке. |
| <b>get(char *, int)</b> | Не устанавливает бит <b>failbit</b> , если не прочитано никаких символов.                                                                                     | Устанавливает бит <b>failbit</b> , если не прочитано никаких символов.                                                                                                                                                                          |

## Другие методы класса *istream*

Другие методы класса *istream* включают в себя **read()**, **peek()**, **gcount()** и **putback()**. Функция **read()** читает указанное число байтов и сохраняет их по указанному адресу. Например, оператор

```
char gross[144];
cin.read(gross, 144);
```

читает 144 символа из стандартного потока ввода и помещает их в массив **gross**. В отличие от функций **getline()** и **get()**, функция **read()** не добавляет нулевой символ к результатам ввода, не преобразуя его в строку. Метод **read()** предназначается в основном не для ввода с клавиатуры. Чаще всего он используется в сочетании с методом **write()** класса **ostream** для файлового ввода/вывода. Метод возвращает значение типа **istream &**, поэтому эти функции можно конкатенировать следующим образом:

```
char gross[144];
char score[20];
cin.read(gross, 144).read(score, 20);
```

Функция **peek()** просматривает следующий символ из потока ввода, не извлекая его из потока, т.е. она позволяет выяснить, какой символ будет считан следующим. Предположим, что нужно считывать поток ввода до первого символа новой строки или точки, в зависимости от того, что встретится раньше. Можно использовать функцию **peek()**, чтобы по следующему символу определить, нужно продолжать чтение или нет:

```
char great_input[80];
char ch;
int i = 0;
while ((ch = cin.peek()) != '.' && ch != '\n')
 cin.get(great_input[i++]);
great_input[i] = '\0';
```

При вызове **cin.peek()** просматривается следующий вводимый символ и присваивается переменной **ch**. Затем условие цикла **while** проверяет, не является ли **ch**

точкой или символом новой строки. Если нет, то оператор внутри цикла помещает этот символ в массив и увеличивает индекс массива. Когда цикл прекращает выполняться, точка или символ новой строки остается в потоке ввода и будет первым символом для следующей операции обработки ввода. Затем этот код добавляет в конец массива нулевой символ, превращая его, таким образом, в строку.

Метод **gcount()** возвращает количество символов, считанное последним методом неформатированного ввода, т.е. методом **get()**, **getline()**, **ignore()** или **read()**, но не операцией извлечения (**>>**), которая форматирует ввод для определенных типов данных. Например, предположим, что для передачи данных в массив **myarray** использовался оператор **cin.get(myarray, 80)**, а затем необходимо узнать, сколько символов было считано. Можно воспользоваться функцией **strlen()** для подсчета символов в массиве, но лучше использовать функцию **cin.gcount()**, которая просто показывает, сколько символов было только что считано из потока ввода.

Функция **putback()** возвращает символ в поток ввода. В этом случае возвращенный символ становится первым символом, прочитанным следующей операцией обработки ввода. Метод **putback()** имеет один аргумент типа **char**, который представляет собой возвращенный символ, и возвращает значение типа **istream &**, которое позволяет конкатенировать его с другими методами класса *istream*. Использование функции **peek()** похоже на использование **get()**. Она выполняет чтение символа с последующим вызовом функции **putback()** для помещения символа назад в поток ввода. Однако функция **putback()** дает возможность поместить в поток символ, отличный от только что прочитанного.

В листинге 16.13 используются два подхода, реализующие считывание и вывод символов до тех пор, пока не встретится символ **#**. При реализации первого подхода символ **#** считывается, а затем с помощью **putback()** помещается назад в поток ввода. При втором подходе используется **peek()** для просмотра символов до начала считывания входного потока.

## Листинг 16.13 Программа peeker.cpp.

```

// peeker.cpp - некоторые методы класса istream
#include <iostream>
using namespace std;
#include <cstdlib> // или stdlib.h

int main()
{
 // считать и вывести весь входной поток символов, пока не встретится символ #
 char ch;

 while(cin.get(ch)) // обрывается при достижении признака EOF (#)
 {
 if (ch != '#')
 cout << ch;
 else
 {
 cin.putback(ch); // вернуть символ во входной поток
 break;
 }
 }

 if (!cin.eof())
 {
 cin.get(ch);
 cout << '\n' << ch << " is next input character.\n";
 }
 else
 {
 cout << "End of file reached.\n";
 exit(0);
 }

 while(cin.peek() != '#') // проверка на наличие символа # во входном потоке
 {
 cin.get(ch);
 cout << ch;
 }

 if (!cin.eof())
 {
 cin.get(ch);
 cout << '\n' << ch << " is next input character.\n";
 }
 else
 cout << "End of file reached.\n";
}

return 0;
}

```

Результаты выполнения программы:

```

I used a #3 pencil when I should have used a #2.
I used a
is next input character.
3 pencil when I should have used a
is next input character.

```

#### Примечания к программе

Давайте внимательно рассмотрим некоторые участки программного кода. При первом подходе используется цикл `while` для чтения входного потока. При выполнении выражения (`cin.get(ch)`) возвращается 0, когда достигнут конец файла, поэтому ввод признака конца файла

с клавиатуры прерывает цикл. Если будет считан символ #, программа возвратит его назад в поток ввода и прервёт цикл с помощью оператора `break`.

```

while(cin.get(ch)) // обрывается при
 // достижении признака EOF
{
 if (ch != '#')
 cout << ch;
 else
 {
 cin.putback(ch); // вернуть символ
 // во входной поток
 break;
 }
}

```

Второй подход проще:

```
while(cin.peek() != '#') // проверка на
 // наличие символа #
{
 cin.get(ch);
 cout << ch;
}
```

Программа просматривает, какой символ будет следующим. Если это не символ #, то программа считывает следующий символ, отображает его и просматривает следующий. Так продолжается до тех пор, пока не встретится последний символ входного потока.

Теперь, как было обещано ранее, давайте рассмотрим пример (листинг 16.14), в котором используется функция peek() для определения того, была ли прочитана вся строка или нет. Как только массив заполняется символами, программа отбрасывает оставшуюся часть строки.

#### Листинг 16.14 Программа truncate.cpp.

```
// truncate.cpp - используется get() для
// усечения строки ввода, если это необходимо
#include <iostream>
using namespace std;
const int SLEN = 10;
inline void eatline()
{
 while (cin.get() != '\n') continue;
}
int main()
{
 char name[SLEN];
 char title[SLEN];
 cout << "Enter your name: ";
 cin.get(name,SLEN);
 if (cin.peek() != '\n')
 cout << "Sorry, we only
 ↵have enough room for "
 << name << endl;
 eatline();
 cout << "Dear " << name
 << ", enter your title: \n";
 cin.get(title,SLEN);
 if (cin.peek() != '\n')
 cout << "We were forced to
 ↵truncate your title.\n";
 eatline();
 cout << " Name: " << name
 << "\nTitle: " << title << endl;
}
return 0;
}
```

Результаты выполнения программы:

```
Enter your name: Stella Starpride
Sorry, we only have enough room for Stella St
Dear Stella St, enter your title:
Astronomer Royal
We were forced to truncate your title.
Name: Stella St
Title: Astronome
```

Заметьте, что следующий код имеет смысл независимо от того, считывается ли вся строка или нет:

```
while (cin.get() != '\n') continue;
```

Если функция get() считывает всю строку, она оставляет символ новой строки во входном потоке, который затем считывается и игнорируется. Если же функция get() читает только часть строки, оставшаяся часть строки отбрасывается, так как в противном случае следующее выражение, обрабатывающее ввод, начнет чтение с нее. В данном примере это приведет к тому, что в массив title программы поместит слово arpride.

## Ввод/вывод файлов

Большинство компьютерных программ работает с файлами. Текстовые процессоры создают файлы документов. Программы баз данных создают файлы и производят в них поиск информации. Компиляторы считывают файлы с исходным кодом и генерируют выполняемые файлы. Сам по себе файл — это набор байтов, сохраняемых на некотором устройстве, возможно, на магнитной ленте, на оптическом диске, диске или на жестком диске. Обычно операционная система управляет файлами, следя за их местоположением, размером, датой создания и т.д. Если только вы не программируете на уровне операционной системы, об этих вопросах можно не беспокоиться. Все, что нужно, — задать способ связи программы с файлом, а также методы, используемые программой при чтении содержимого файла, записи в файл, а также при создании нового файла и записи в него. Перенаправление обеспечивает некоторую обработку файлов, но является гораздо более ограниченным, чем явный файловый ввод/вывод внутри программы. Кроме того, перенаправление является свойством операционной системы, а не языка C++, поэтому оно доступно не для всех систем. Сейчас мы рассмотрим, как C++ работает с явным вводом/выводом в программе.

Пакет классов C++ работает с файловым вводом/выводом во многом так же, как он делает это со стандартным вводом/выводом. Чтобы записать данные в файл, необходимо создать потоковый объект и воспользоваться такими методами класса ostream, как операция вставки << или функция write(). Для чтения файла создается потоковый объект и применяются такие методы класса istream, как операция извлечения >> или функция get(). Однако файлы требуют более высокого уровня управления, чем стандартные потоки ввода/вывода. Например, новый открытый файл необходимо поставить в соответствие какому-либо потоку. Файлы можно открывать в режиме только для чтения, только для записи и в режиме чтения/записи. При записи в файл может возникнуть необходимость создать новый файл, заме-

нить старый или добавить данные к старому файлу. Существует также возможность перемещаться по файлу в разных направлениях. Чтобы помочь справиться с этими задачами, в C++ определены несколько классов в заголовочном файле **fstream** (ранее — **fstream.h**), включая класс **ifstream** для выполнения файлового ввода и класс **ofstream** — для файлового вывода. C++ также определяет класс **fstream** для одновременного файлового ввода/вывода. Эти классы являются производными от классов в заголовочном файле **iostream**, поэтому объекты этих новых классов могут использовать те методы, которые мы только что изучили.

## Простой файловый ввод/вывод

Предположим, что программе необходимо записать информацию в файл. Для этого требуется выполнить следующее:

- Создать объект класса **ofstream** для управления потоком вывода.
- Поставить этот объект в соответствие определенному файлу.
- Использовать объект так же, как использовался объект **cout**; единственная разница заключается в том, что информация выводится в файл, а не на экран.

Для выполнения этой задачи начнем с включения заголовочного файла **fstream**. При включении этого файла автоматически включается файл **iostream** почти для всех реализаций языка C++, поэтому **iostream** не нужно включать явно. Затем необходимо объявить объект класса **ofstream**:

```
//создать объект класса ofstream с именем fout
ofstream fout;
```

Имя объекта может быть любым допустимым именем C++, например, **fout**, **outFile**, **cgate** или **didi**.

Затем необходимо поставить в соответствие объекту определенный файл. Это можно сделать, воспользовавшись методом **open()**. Предположим, что необходимо открыть для записи файл, например, **cookies**. Для этого нужно сделать следующее:

```
// поставить в соответствие объекту fout
// файл cookies
fout.open("cookies");
```

Два этих этапа (создание объекта и его ассоциация с файлом) можно объединить в одном операторе, воспользовавшись другим конструктором:

```
// создать объект fout и поставить ему
// в соответствие файл cookies
ofstream fout("cookies");
```

После этого объект **fout** (или объект с любым другим именем) можно использовать точно так же, как и объект **cout**. Например, если требуется записать слова **Dull Data** в файл, то для этого необходимо выполнить следующее:

```
fout << "Dull Data";
```

В самом деле, поскольку класс **ostream** является базовым для класса **ofstream**, можно использовать методы класса **ostream**, включая различные определения операций вставки, а также методы и манипуляторы форматирования. Класс **ofstream** работает с буферизованным выводом, поэтому программа выделяет место для буфера вывода при создании такого объекта класса **ofstream**, как **fout**. Если создаются два объекта класса **ofstream**, то программа создает два буфера — отдельно для каждого объекта. Объект класса **ofstream** наподобие **fout** считывает побайтно вывод, сгенерированный программой. Затем, когда буфер заполняется, содержимое буфера целиком передается в файл назначения. Поскольку дисковые накопители предназначены для передачи данных большими блоками, а не побайтово, буферизованный подход значительно повышает скорость пересылки данных от программы к файлу.

Открытие файла для вывода таким способом позволяет создать новый файл, если файла с таким именем не существует. Если же файл с таким именем существует, то до открытия для вывода этот файл урезается до нулевого размера и информация начинает выводиться в пустой файл. Далее будет рассмотрено, как можно открыть существующий файл, сохранив его содержимое.

### ПРЕДОСТЕРЕЖЕНИЕ

При открытии файла для вывода в режиме, заданном по умолчанию, автоматически урезается его размер до нуля и удаляется его предыдущее содержимое.

Требования к чтению файла очень похожи на требования к записи:

1. Создать объект класса **ifstream** для управления потоком ввода.
2. Поставить этот объект в соответствие определенному файлу.
3. Использовать объект так же, как объект **cin**.

Этапы выполнения этих действий совпадают с этапами записи файла. Сначала конечно же необходимо включить заголовочный файл **fstream**, затем объявить объект класса **ifstream** и поставить ему в соответствие файл. Это можно сделать с помощью двух операторов или одного:

```
// два оператора
//создать объект класса ifstream с именем fin
ifstream fin;
```

```
// открыть файл jellyjar.dat для чтения
fin.open("jellyjar.dat");

// один оператор

// создать объект ifs и поставить ему
// в соответствие файл jamjar.dat
ifstream ifs("jamjar.dat");
```

Теперь объекты `fin` или `ifs` можно использовать так же, как объект `cin`. Например, можно сделать следующее:

```
char ch;

// считать символ из файла jellyjar.dat
fin >> ch;

char buf[80];

// считать слово из файла
fin >> buf;

// считать строку из файла
fin.getline(buf, 80);
```

Ввод, как и вывод, буферизован, поэтому при создании объекта класса `ifstream` наподобие `fin` создается буфер ввода, управляемый этим объектом. Как и при выводе, буферизация позволяет работать с данными гораздо быстрее, чем при однобайтовом обмене.

Связь с файлом прекращается автоматически, если истекает срок действия объектов потока ввода/вывода, например, по окончании выполнения программы. Кроме того, связь с файлом можно прервать явно, воспользовавшись методом `close()`:

```
// прервать связь с файлом для вывода
fout.close();

// прервать связь с файлом для ввода
fin.close();
```

При прерывании такой связи не ликвидируется поток, он просто отключается от файла. Однако аппарат управления потоком остается. Например, объект `fin` все еще существует вместе с буфером, которым он управляется. Как станет видно в дальнейшем, поток можно вновь соединить с тем же самым или с другим файлом.

Тем временем рассмотрим короткий пример. Программа из листинга 16.15 запрашивает имя файла. Затем она создает файл с таким именем, записывает в него некоторую информацию и закрывает файл. При закрытии файла очищается буфер, гарантуя, таким образом, что файл будет обновлен. Затем программа открывает тот же файл для чтения и выводит его содержимое на экран. Заметьте, что программа использует `fin` и `fout` точно так же, как и объекты `cin` и `cout`.

### Листинг 16.15 Программа file.cpp.

```
// file.cpp — сохранение в файле
// не нужно для многих систем

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
 char filename[20];

 cout << "Enter name for new file: ";
 cin >> filename;

 // создать объект потока вывода для нового файла и назвать его fout
 ofstream fout(filename);

 fout << "For your eyes only!\n"; // записать в файл
 cout << "Enter your secret number: "; // вывести на экран
 float secret;
 cin >> secret;
 fout << "Your secret number is " << secret << "\n";
 fout.close(); // закрыть файл

 // создать объект потока ввода для нового файла и назвать его fin
 ifstream fin(filename);
 cout << "Here are the contents of " << filename << ":\n";
 char ch;
 while (fin.get(ch)) // считать символ из файла и
 cout << ch; // вывести его на экран
 cout << "Done\n";
 fin.close();

 return 0;
}
```

Результаты выполнения программы:

```
Enter name for new file: pythag
Enter your secret number: 3.14159
Here are the contents of pythag:
For your eyes only!
Your secret number is 3.14159
Done
```

Если посмотреть в каталог, содержащий программу, то там можно найти файл **pythag**, а любой текстовый редактор должен показать такое же содержимое файла, которое программа выводила на экран.

## Открытие нескольких файлов

Может возникнуть необходимость в том, чтобы программа открывала сразу более одного файла. Стратегия открытия нескольких файлов зависит от того, как они будут использоваться. Если нужно, чтобы одновременно были открыты два файла, следует создать отдельный поток для каждого файла. Например, программа, которая сравнивает два отсортированных файла и записывает результаты в третий, должна создать два объекта класса **ifstream** для двух входных файлов и один объект класса **ofstream** для выходного файла. Число файлов, которые можно открыть одновременно, зависит от операционной системы, но обычно оно составляет 20.

Однако можно обрабатывать группу файлов последовательно. Например, может возникнуть необходимость подсчитать, сколько раз имя появляется в наборе из десяти файлов. Для этого можно открыть только один поток и ставить его в соответствие только одному файлу, обрабатывая файлы по очереди. При этом ресурсы компьютера используются более эффективно, чем при открытии отдельного потока для каждого файла. Для использования этого подхода необходимо объявить потоковый объект без его инициализации и затем использовать метод **open()** для соединения потока с файлом. Например, вот как можно обработать два файла последовательно:

```
// создать поток, используя конструктор,
// заданный по умолчанию
ifstream fin;

// поставить в соответствие потоку файл fat.dat
fin.open("fat.dat");
... // обрабатывать информацию

// закрыть связь потока и файла fat.dat
fin.close();

// переинициализировать fin
// (может быть необязательным)
fin.clear();

// поставить в соответствие потоку файл rat.dat
fin.open("rat.dat");
...
fin.close();
```

Вскоре мы рассмотрим этот пример, но сначала обратимся к методу, который позволяет программе обрабатывать набор файлов так, чтобы можно было воспользоваться циклом для их обработки.

## Работа в режиме командной строки

Программы, обрабатывающие файлы, часто используют аргументы командной строки как способ указания файлов. Аргументы командной строки — это аргументы, которые появляются в командной строке при наборе команды. Например, для подсчета слов в некоторых файлах в системе UNIX можно набрать в командной строке UNIX такую команду:

```
wc report1 report2 report3
```

Здесь **wc** — это имя программы, а **report1**, **report2** и **report3** — это имена файлов, передаваемые программе как аргументы командной строки.

Язык C++ имеет механизм, позволяющий программе получить доступ к аргументам командной строки. Для этого можно использовать следующий альтернативный заголовок функции **main()**:

```
int main(int argc, char *argv[])
```

Аргумент **argc** представляет собой количество аргументов командной строки. Этот счетчик включает и само имя программы. Переменная **argv** — это указатель на тип **char**. Переменную **argv** можно трактовать как массив указателей на аргументы командной строки, причем **argv[0]** — это указатель на первый символ строки, содержащей имя команды, **argv[1]** — это указатель на первый символ строки, содержащей первый аргумент командной строки, и т.д. Иначе говоря, **argv[0]** — это первая строка из командной строки, **argv[1]** — вторая строка и т.д. Например, предположим, что командная строка имеет следующий вид:

```
wc report1 report2 report3
```

Тогда **argc** равно 4, **argv[0]** равно **wc**, **argv[1]** равно **report1** и т.д. В следующем цикле каждый элемент командной строки выводится в отдельной строке:

```
for (int i = 1; i < argc; i++)
 cout << argv[i] << "\n";
```

Если цикл начинается с **i = 1**, то будут выведены только аргументы командной строки, а если цикл начинается с **i = 0**, будет отображено также имя команды.

Аргументы командной строки аналогичны операционным системам командной строки типа DOS или UNIX. Другие системы, тем не менее, также позволяют использовать аргументы командной строки:

- Многие IDE (интегрированные среды разработки) в DOS и Windows позволяют задавать параметры в ко-

мандной строке. Обычно необходимо пройти через несколько меню, чтобы вывести окно, в котором можно будет задать аргументы командной строки. Точный набор требуемых этапов зависит от производителя среды и может меняться от версии к версии, поэтому необходимо обратиться к документации.

- В IDE для DOS и во многих IDE для Windows могут создаваться выполняемые файлы, которые можно запускать в DOS или в окне DOS в стандартном режиме командной строки.
- В Symantec C++ для Macintosh и в Metrowerks CodeWarrior для Macintosh можно эмулировать аргументы командной строки, поместив в программу следующий блок кода:

```
...
#include <console.h> // для эмуляции
// аргументов командной строки
int main(int argc, char * argv[])
{
 argc = ccommand(&argv); // да, именно
 // ccommand, а не command
 ...
```

#### Листинг 16.16 Программа count.cpp.

```
// count.cpp - подсчитывает количество символов в файлах по списку
#include <iostream>
using namespace std;
#include <fstream>
#include <cstdlib>
// #include <console.h> // или stdlib.h // для Macintosh
int main(int argc, char * argv[])
{
 // argc = ccommand(&argv); // для Macintosh
 if (argc == 1) // выйти из программы, если аргументы не заданы
 {
 cerr << "Usage: " << argv[0] << " filename[s]\n";
 exit(1);
 }

 ifstream fin; // открыть поток
 long count;
 long total = 0;
 char ch;

 for (int file = 1; file < argc; file++)
 {
 fin.open(argv[file]); // соединить поток с файлом argv[file]
 count = 0;
 while (fin.get(ch))
 count++;
 cout << count << " characters in " << argv[file] << "\n";
 total += count;
 fin.clear(); // необходимо в некоторых реализациях
 fin.close(); // отсоединить файл
 }
 cout << total << " characters in all files\n";
 return 0;
}
```

Когда программа запускается, функция `ccommand()` выводит диалоговое окно, в котором можно задать аргументы командной строки. Эта функция позволяет также эмулировать перенаправление.

В листинге 16.16 скомбинированы некоторые методы обработки аргументов командной строки с методами обработки файлового потока для подсчета символов в файлах, указанных в командной строке.

#### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

В одних реализациях языка C++ требуется использовать функцию `fin.clear()`, в других нет. Это зависит от того, будет ли в результате связи нового файла с объектом `fstream` автоматически переустанавливаться состояние потока или нет. Использование функции `fin.clear()` не наносит никакого ущерба, даже если в нем нет необходимости.

В системе DOS, например, можно скомпилировать программу из листинга 16.16 в выполняемый файл `count.exe`. Результат его выполнения будет выглядеть следующим образом:

```
C>count
Usage: c:\count.exe filename[s]
C>count paris rome
3580 characters in paris
4886 characters in rome
8466 characters in all files
C>
```

Заметьте, что программа использует поток `cerr` для вывода сообщений об ошибках. Кроме того, в сообщении используется `argv[0]`, а не `count.exe`:

```
cerr << "Usage: " << argv[0]
<< " filename[s]\n";
```

Таким образом, если имя исполняемого файла изменяется, программа автоматически использует новое имя.

Предположим, что программе подсчета символов передано имя несуществующего файла. Тогда оператор ввода `fin.get(ch)` потерпит неудачу, сразу же прервав цикл, и программа выдаст результат: 0 символов. Однако программу можно изменить так, чтобы она могла проверить, было ли успешно выполнено соединение потока с файлом. Это один из вопросов, которые мы рассмотрим в следующем разделе.

## Проверка потока и функция `is_open()`

Классы файловых потоков C++ наследуют элемент, показывающий состояние потока, от класса `ios_base`. Этот элемент, как известно, хранит информацию, отражающую состояние потока: все ли в порядке, достигнут ли конец файла, произошла ли ошибка операции ввода/вывода и т.д. Если все в порядке, то состояние потока определяется нулем (отсутствие новостей — хорошие новости). Другие различные состояния устанавливают соответствующие биты равными 1. Классы файловых потоков наследуют методы класса `ios_base`, которые выводят информацию о состоянии потока и которые были приведены ранее в табл. 16.5. С помощью этих методов можно проверить условия состояния потока. Например, можно использовать метод `good()` для того, чтобы проверить, все ли биты состояния очищены. Однако новые реализации C++ дают более удобный метод проверки того, был ли открыт файл. Этот метод называется `is_open()`. Программу из листинга 16.16 можно модифицировать так, чтобы она осуществляла проверку того, было ли задано имя несуществующего файла, и пропускала его, чтобы обработать следующий файл. Для этого необходимо добавить вызов функции `fin.is_open()` в цикле:

```
for (int file = 1; file < argc; file++)
{
 fin.open(argv[file]);
// добавить вызов функции
 if (!fin.is_open())
```

```
{
 cerr << "Couldn't open file "
 << argv[file] << "\n";
 continue;
}
// конец добавления

count = 0;
while (fin.get(ch))
 count++;
cout << count << " characters in "
 << argv[file] << "\n";
total += count;
fin.clear();
fin.close(); // отсоединить файл
}
```

В результате вызова функции `fin.is_open()` возвращается значение `false`, если вызов функции `fin.open()` был неудачным. В таком случае программа выдает предупреждение об этой проблеме и оператор `continue` заставляет программу пропустить остаток цикла и начать новую итерацию.

### ПРЕДОСТЕРЕЖЕНИЕ

В прошлом обычные тесты на успешное открытие файла выглядели следующим образом:

```
if(!fin.good()) ... // неудача при открытии
if (!fin) ... // неудача при открытии
```

Объект `fin`, когда используется тестовое условие, принимает значение `false`, если функция `fin.good()` возвращает значение `false`, в противном случае он принимает значение `true` поэтому эти две формы эквивалентны. Однако есть обстоятельство, которое этот тест не может обнаружить, — речь идет об использовании неподходящего режима открытия файла (см. раздел "Режимы файлов"). Метод `is_open()` обрабатывает этот тип ошибки вместе с теми, которые обрабатывает метод `good()`. Однако более ранние реализации не поддерживают метод `is_open()`.

## Режимы файлов

Режим файла описывает, как используется файл: для чтения, для записи, для добавления и т.д. Когда поток ассоциируется с файлом, а также при инициализации файлового потокового объекта именем файла или при работе с методом `open()`, можно использовать и второй аргумент, задающий режим файла:

```
// конструктор с аргументом режима
ifstream fin("banjo", mode1);
ofstream fout();
// метод open() с аргументом режима
fout.open("harp", mode2);
```

Класс `ios_base` определяет тип `openmode` для представления режима файла; как и типы `fmtflags` и `iostate`, это тип битовой маски (ранее это был тип `int`). В классе `ios_base` определено несколько констант, которыми можно воспользоваться для указания режима файла. В табл.

16.8 приведен список констант и их значений. Файловый ввод/вывод в C++ претерпел некоторые изменения, чтобы стать совместимым с файловым вводом/выводом ANSI C.

**Таблица 16.8 Константы режимов файла.**

| Константа                     | Значение                                                |
|-------------------------------|---------------------------------------------------------|
| <code>ios_base::in</code>     | Открыть файл для чтения.                                |
| <code>ios_base::out</code>    | Открыть файл для записи.                                |
| <code>ios_base::ate</code>    | Переместить указатель в конец файла после его открытия. |
| <code>ios_base::app</code>    | Добавить информацию к концу файла.                      |
| <code>ios_base::trunc</code>  | Урезать файл, если он существует.                       |
| <code>ios_base::binary</code> | Двоичный файл.                                          |

Если конструкторы классов `ifstream` и `ofstream`, а также методы `open()` имеют по два аргумента, как же нам удалось использовать их с одним аргументом в предыдущих примерах? Нетрудно догадаться, что прототипы для этих функций-элементов класса обеспечивают для второго аргумента (аргумента режима файла) значения, заданные по умолчанию. Например, метод `open()` класса `ifstream` и его конструктор используют константу `ios_base::in` (открыть для чтения) как значение, заданное по умолчанию для аргумента режима файла, в то время как метод `open()` класса `ofstream` и его конструктор используют `ios_base::out` | `ios_base::trunc` (открыть для записи и урезать файл) как значение, заданное по умолчанию. Поразрядный оператор ИЛИ () используется для сложения двух битовых значений в одно, в котором будут установлены оба бита. Класс `fstream` не обеспечивает режим файла, заданный по умолчанию, поэтому необходимо указывать режим явно, причем это необходимо делать при создании объекта данного класса.

Заметьте, флаг `ios_base::trunc` означает, что существующий файл урезается при открытии для программного вывода; таким образом, предыдущее содержимое файла удаляется. Это позволяет значительно снизить опасность переполнения дискового пространства. Однако можно себе представить ситуацию, когда совсем необязательно удалять содержимое файла при его открытии. C++, разумеется, обеспечивает и другие возможности. Если, например, необходимо сохранить содержимое файла и добавить новые данные к концу файла, можно воспользоваться режимом `ios_base::app`:

```
ofstream fout("bagels",
 ios_base::out | ios_base::app);
```

Вновь в программном коде используется оператор | для создания комбинации режимов. Таким образом, `ios_base::out` | `ios_base::app` означает, что будут включены и режим `out`, и режим `app` (рис. 16.6).

Среди более ранних реализаций языка C++ существуют некоторые различия. Например, в одних реализациях позволяет пропускать константу `ios_base::out` в предыдущем примере, а в других — нет. Если не используется режим, заданный по умолчанию, то наиболее безопасный подход заключается в том, чтобы указать все режимы явно. Одни компиляторы не поддерживают все возможности, представленные в табл. 16.7, а другие могут предлагать и дополнительные возможности, не указанные в таблице. Последствием этих различий является то, что для проверки приведенных примеров в своей системе может понадобиться внести в них некоторые изменения. Хорошая новость заключается в том, что при разработке стандарта C++ обеспечивается более высокая степень универсальности.

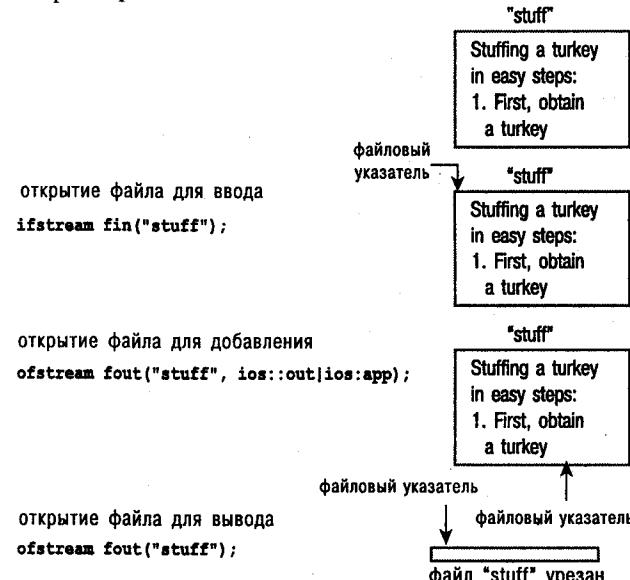
Стандарт C++ определяет компоненты ввода/вывода в языке C++, используя термины из стандарта ANSI C. Следующий оператор C++:

```
ifstream fin(filename, c++mode);
```

реализуется так, как если бы он использовал функцию C `fopen()`:

```
fopen(filename, cmode);
```

Здесь `c++mode` имеет тип значения `openmode`, как `ios_base::in` в ANSI C, а `cmode` отвечает строке режима файла из C, например, строке "r". В табл. 16.9 показано соответствие между режимами в C++ и C. Заметьте, что константа `ios_base::out` сама по себе задает усечение файла, но этого не происходит, если за ней следует `ios_base::in`. Не приведенные в таблице комбинации, например `ios_base::in` | `[vn] ios_base::trunc`, не позволяют открыть файл.



**РИСУНОК 16.6 Некоторые режимы открытия файлов.**

Таблица 16.9 Режимы открытия файлов C++ и C.

| Режим C++                                                   | Режим C  | Значение                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------------|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ios_base::in</code>                                   | "r"      | Открыть для чтения.                                                                                                                                                                                                                                                                        |
| <code>ios_base::out</code>                                  | "w"      | (То же самое, что и <code>ios_base::out   ios_base::trunc</code> ).                                                                                                                                                                                                                        |
| <code>ios_base::out   ios_base::trunc</code>                | "w"      | Открыть для записи, урезав длину файла до нуля, если он уже существует.                                                                                                                                                                                                                    |
| <code>ios_base::out   ios_base::app</code>                  | "a"      | Открыть файл для записи, но только для добавления к файлу.                                                                                                                                                                                                                                 |
| <code>ios_base::in   ios_base::out</code>                   | "r+"     | Открыть для чтения и записи, причем запись разрешена в любом месте файла.                                                                                                                                                                                                                  |
| <code>ios_base::in   ios_base::out   ios_base::trunc</code> | "w+"     | Открыть файл для чтения и записи, вначале урезав длину файла до нуля, если он уже существует.                                                                                                                                                                                              |
| <code>c++mode   ios_base::binary</code>                     | "cmodeb" | Открыть файл в режиме <code>c++mode</code> или в отвечающем ему режиме <code>cmode</code> , а также в двоичном режиме; например, <code>ios_base::in   ios_base::binary</code> становится "rb".                                                                                             |
| <code>c++mode   ios_base::ate</code>                        | "cmode"  | Открыть в указанном режиме и перейти к концу файла. В языке выполняется вызов отдельной функции вместо указания режима. Например, <code>ios_base::in   ios_base::ate</code> превращается в вызов функции C с указанием режима перемещения по файлу <code>fseek(file, 0, SEEK_END)</code> . |

Заметьте, что и `ios_base::ate`, и `ios_base::app` перемещают файловый указатель в конец только что открытого файла. Разница между ними заключается в том, что режим `ios_base::app` позволяет добавлять данные только в конце файла, в то время как режим `ios_base::ate` позволяет просто помещать указатель в конец файла.

Очевидно, есть много различных комбинаций режимов. Мы рассмотрим только некоторые из них.

### Добавление к файлу

Начнем с программы, которая добавляет данные к концу файла. Программа обрабатывает файл, содержащий список гостей. Вначале она выводит текущее содержимое файла, если таковой существует. После попытки открытия файла используется метод `is_open()`, чтобы убедиться, что файл существует. Затем программа открывает файл для вывода, используя режим `ios_base::app`. Затем она запрашивает ввод с клавиатуры, чтобы добавить его к файлу, и в конце выводит содержимое измененного файла. Листинг 16.17 иллюстрирует, каким образом эти цели будут достигнуты. Отметьте, как программа использует метод `is_open()` для проверки того, был ли файл открыт успешно.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Файловый ввод/вывод был, возможно, аспектом C++, стандартизованным в последнюю очередь при создании языка, поэтому многие устаревшие компиляторы не до конца поддерживают текущий стандарт. Некоторые, например, используют режимы наподобие `poscreate`, которые не поддерживаются текущим стандартом. Кроме того, только некоторые из компиляторов требуют реализации вызова функции `fin.clear()` перед открытием того же самого файла для чтения.

Результаты выполнения программы из листинга 16.17. В этот момент файл `guests.dat` еще не создан, поэтому программа не выводит содержимое файла.

```
Enter guest names (enter a blank line to quit):
Sylvester Ballone
Phil Kates
Bill Ghan
```

Вот новое содержимое файла `guests.dat`:

```
Sylvester Ballone
Phil Kates
Bill Ghan
```

При следующем запуске программы файл `guests.dat` уже существует, поэтому программа выводит его содержимое. Кроме того, она добавляет новые данные к существующему файлу, а не заменяет их.

```
Sylvester Ballone
Phil Kates
Bill Ghan
Enter guest names (enter a blank line to quit):
Greta Greppo
LaDonna Mobile
Fannie Mae
```

Here are the new contents of the `guests.dat` file:

```
Sylvester Ballone
Phil Kates
Bill Ghan
Greta Greppo
LaDonna Mobile
Fannie Mae
```

Содержимое файла `guest.dat` можно прочесть с помощью любого текстового редактора, включая и тот, в котором создается исходный код.

## Листинг 16.17 Программа append.cpp.

```

// append.cpp - запись информации в файл

#include <iostream>
using namespace std;
#include <fstream>
#include <cstdlib> // ((или stdlib.h) для функции exit()

const char * file = "guests1.dat";
const int Len = 40;
int main()
{
 char ch;

// вывести первоначальное содержимое файла
 ifstream fin;
 fin.open(file);

 if (fin.is_open())
 {
 cout << "Here are the current contents of the "
 << file << " file:\n";
 while (fin.get(ch))
 cout << ch;
 }
 fin.close();

// добавить новые имена
 ofstream fout(file, ios::out | ios::app);
 if (!fout.is_open())
 {
 cerr << "Can't open " << file << " file for output.\n";
 exit(1);
 }

 cout << "Enter guest names (enter a blank line to quit):\n";
 char name[Len];
 cin.get(name, Len);
 while (name[0] != '\0')
 {
 while (cin.get() != '\n')
 continue; // исключить \n и длинные строки
 fout << name << "\n";
 cin.get(name, Len);
 }
 fout.close();

// вывести измененный файл
 fin.clear(); // необязательно для некоторых компиляторов
 fin.open(file);
 if (fin.is_open())
 {
 cout << "Here are the new contents of the "
 << file << " file:\n";
 while (fin.get(ch))
 cout << ch;
 }
 fin.close();

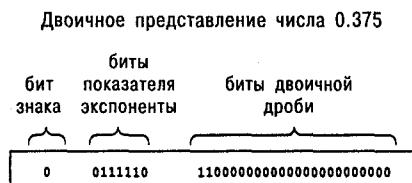
 return 0;
}

```

## Двоичные файлы

Когда данные сохраняются в файле, их можно сохранить в текстовой форме или двоичном формате. Текстовая форма означает, что все данные сохраняются как текст, даже числа. Например, сохранение значения -2.324216e+07 в текстовой форме означает сохранение 13 символов, из которых состоит данное число. Для этого требуется, чтобы внутреннее представление компьютера для числа с плавающей точкой было преобразовано в текстовую форму, с этой целью выполняется операция вставки <<. Двоичный формат означает, что число сохраняется во внутреннем представлении, т.е. вместо символов сохраняется 64-разрядное (обычно) представление числа типа **double**. Для символа двоичное представление совпадает с его текстовым — двоичным представлением ASCII-кода (или его эквивалента) символа. Однако для чисел двоичное представление очень сильно отличается от их текстового представления (рис. 16.7).

Каждый формат имеет свои достоинства. Текстовый формат прост для чтения, поэтому для чтения и редактирования текстового файла можно воспользоваться обычным редактором или текстовым процессором. Текстовый файл можно с легкостью перенести с одной компьютерной системы на другую. В двоичном же формате числа сохраняются более точно, поскольку он позволяет сохранить точное внутреннее представление числа. Не происходит ошибок преобразования или округления. Сохранение данных в двоичном формате может происходить быстрее, поскольку при этом не происходит преобразования и данные можно сохранять большими блоками. Кроме того, двоичный формат обычно занимает меньше места (в зависимости от природы данных). Однако перенос данных в другую систему может быть проблемой, если новая система использует другое внутреннее представление данных. В этом случае может оказаться необходимым создать программу, которая преобразует один формат данных в другой.



Текстовое представление числа 0.375

|            |            |            |            |            |
|------------|------------|------------|------------|------------|
| 00110000   | 00101110   | 00110011   | 00110111   | 00110111   |
| символ "0" | символ "," | символ "3" | символ "7" | символ "5" |

РИСУНОК 16.7 Двоичное и текстовое представление числа с плавающей точкой.

Рассмотрим более конкретный пример. Представьте себе следующее определение и объявление структуры:

```
struct planet
{
 char name[20]; // название планеты
 double population; // ее население
 double g; // ускорение свободного
 // падения на ней
};
```

Чтобы сохранить содержимое структуры **pl** в текстовой форме, нужно сделать следующее:

```
ofstream fout("planets.dat", ios_base::app);
fout << pl.name << " " << pl.population
<< " " << pl.g << "\n";
```

Заметьте, что каждый элемент структуры необходимо указать явно, используя операцию членства, и, кроме того, для удобства нужно разделить данные при выводе. Если бы структура содержала, скажем, 30 элементов, это было бы довольно утомительно.

Чтобы сохранить ту же самую информацию в двоичном формате, можно сделать следующее:

```
ofstream fout("planets.dat",
ios_base::app | ios_base::binary);
fout.write((char *) &pl, sizeof pl);
```

Этот код сохраняет структуру целиком как одну единицу данных, используя внутреннее представление данных. Данный файл нельзя прочесть как текст, но информация в нем сохранена более компактно и более точно, чем, например, в текстовом формате. И это, конечно, проще с точки зрения написания кода. Такой подход привел к выполнению двух изменений, которые заключались в использовании:

- Двоичного режима записи в файл.
- Функции-элемента **write()**.

Рассмотрим эти изменения более подробно.

Некоторые системы, например DOS, поддерживают два формата файлов: текстовый и двоичный. Если необходимо сохранить данные в двоичном формате, лучше использовать двоичный формат файлов. В C++ это делается путем использования константы **ios\_base::binary** для режима файла. Если вы хотите узнать, почему это нужно делать в системе DOS, прочтите в разделе "Двоичные и текстовые файлы".

### ДВОИЧНЫЕ И ТЕКСТОВЫЕ ФАЙЛЫ

Использование двоичного режима файла приводит к тому, что программа пересыпает данные из памяти в файл или в обратном направлении без какого-либо их преобразования. Однако это совсем необязательно выполняется в случае текстового режима. Например, рассмотрим текстовые файлы DOS. В них символ новой строки представлен двухсимвольной комбинацией: символами возврата карет-

ки и конца строки. В текстовых файлах Macintosh символ новой строки представлен символом возврата каретки. В файлах UNIX символ новой строки представлен символом конца строки. В C++, который возник в рамках операционной системы UNIX, символ новой строки также представлен символом конца строки. Для обеспечения переносимости DOS-программа, написанная на C++, автоматически переводит символ новой строки C++ в комбинацию "возврат каретки/конец строки" при записи в текстовый файл. Аналогично, Macintosh-программа, написанная на C++, переводит символ новой строки в символ возврата каретки. При чтении текстовых файлов эти программы преобразуют локальный символ новой строки обратно в форму C++. Текстовый формат может вызвать проблемы с двоичными данными, так как байт из представления числа с плавающей точкой может иметь такой же код, как ASCII-код для символа новой строки. Кроме того, есть различия в том, как определяется конец файла. Поэтому для записи данных в двоичном формате необходимо всегда использовать двоичный режим. (Системы UNIX имеют только один файловый режим, поэтому двоичный режим совпадает с текстовым.)

Чтобы сохранить данные в двоичной форме, а не текстовой, можно воспользоваться методом `write()`. Вспомните, что этот метод копирует определенное число байтов из памяти в файл. Мы использовали его ранее для копирования текста, но он может копировать любой тип данных байт в байт, не производя преобразования. Например, если ему передать адрес переменной типа `long` и указать скопировать 4 байта, то данный метод скопирует 4 байта, "дословно" передав значение типа `long` и не производя его преобразования. Единственное неудобство заключается в том, что адрес переменной необходимо преобразовать к типу указатель-на-`char`. Точно такой же подход можно использовать для копирования структуры `planet` целиком. Чтобы узнать ее размер в байтах, необходимо воспользоваться оператором `sizeof`.

```
fout.write((char *) &p1, sizeof p1);
```

Данный оператор копирует 36 байтов (значение выражения `sizeof p1`), начиная с указанного адреса, в файл, связанный с объектом `fout`.

Чтобы прочесть информацию из файла, нужно использовать соответствующий метод `read()` с объектом `ifstream`:

```
ifstream fin("planets.dat",
ios_base::binary);
fin.read((char *) &p1, sizeof p1);
```

Данный блок кода копирует количество байтов `sizeof p1` из файла в структуру `p1`. Этот же самый подход можно применить для классов, не использующих виртуальные функции. В этом случае сохраняются только элементы данных, а не методы класса. Если же класс содержит виртуальные функции, тогда скрытый указатель на таблицу, содержащую указатели на виртуальные функции,

также копируется. Поскольку при следующем запуске программы таблица виртуальных функций может быть расположена по другому адресу, копирование информации о старых указателях в объект из файла может привести к путанице. (См. также примечание к упражнению по программированию 6.)



### COBET

Методы `read()` и `write()` дополняют друг друга. Используйте `read()` для чтения данных из файла, которые были записаны с помощью `write()`.

В листинге 16.18 эти методы используются для создания и чтения двоичного файла. По форме программа подобна программе из листинга 16.17, но в ней используются методы `write()` и `read()` вместо операции вставки и метода `get()`. Кроме того, для форматирования вывода на экран используются манипуляторы.



### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Хотя концепция двоичного файла является частью ANSI C, некоторые реализации C и C++ не обеспечивают поддержки двоичного режима файлов. Причина этой оплошности состоит в том, что некоторые системы имеют только один тип файла, поэтому можно пользоваться такими двоичными операциями, как `read()` и `write()`, для стандартного формата файлов. Поэтому, если используемая вами реализация отвергает `ios_base::binary` как неподходящую константу, ее необходимо просто убрать из программы. Если ваша реализация не поддерживает манипулятор `fixed`, можно воспользоваться оператором `cout.setf(ios_base::fixed, ios_base::floatfield)`. Symantec C++ версии 8 требует заменить два оператора

```
while (fin.read((char *) &p1, sizeof p1))
```

следующими:

```
while (fin.read((char *) &p1, sizeof p1) &&
!fin.eof())
```

Результаты первого выполнения программы из лист. 16.18:

```
Enter planet name (enter a blank line to quit):
Earth
Enter planetary population: 5962000000
Enter planet's acceleration of gravity: 9.81
Enter planet name (enter a blank line to quit):
Here are the newcontents of the planets.dat file:
Earth: 5932000000 9.81
```

Результаты следующего выполнения программы:

```
Here are the current contents of the
planets.dat file:
Earth: 5932000000 9.81
Enter planet name (enter a blank line to quit):
Bill's Planet
Enter planetary population: 23020020
Enter planet's acceleration of gravity: 8.82
Enter planet name (enter a blank line to quit):
Here are the new contents of the planets.dat file:
Earth: 5932000000 9.81
Bill's Planet: 23020020 8.82
```

## Листинг 16.18 Программа binary.cpp.

```

#include <iostream> // не требуется для многих систем
using namespace std;
#include <fstream>
#include <iomanip>
#include <cstdlib> // (или stdlib.h) для функции exit()

inline void eatline() { while (cin.get() != '\n') continue; }

struct planet
{
 char name[20]; // имя планеты
 double population; // ее население
 double g; // ускорение свободного падения на ней
};

const char * file = "planets.dat";

int main()
{
 planet pl;
 cout << fixed << right;

// вывести начальное содержимое
 ifstream fin;
 fin.open(file, ios::in | ios::binary); // двоичный файл
 //ПРИМЕЧАНИЕ: некоторые системы не поддерживают режим ios::binary
 if (fin.is_open())
 {
 cout << "Here are the current contents of the "
 << file << "\n";
 while (fin.read((char *) &pl, sizeof pl))
 {
 cout << setw(20) << pl.name << ":"
 << setprecision(0) << setw(12) << pl.population
 << setprecision(2) << setw(6) << pl.g << "\n";
 }
 }
 fin.close();

// добавление новых данных
 ofstream fout(file, ios::out | ios::app | ios::binary);
 //Примечание: некоторые системы не поддерживают режим ios::binary
 if (!fout.is_open())
 {
 cerr << "Can't open " << file << " file for output:\n";
 exit(1);
 }

 cout << "Enter planet name (enter a blank line to quit):\n";
 cin.get(pl.name, 20);
 while (pl.name[0] != '\0')
 {
 eatline();
 cout << "Enter planetary population: ";
 cin >> pl.population;
 cout << "Enter planet's acceleration of gravity: ";
 cin >> pl.g;
 eatline();
 fout.write((char *) &pl, sizeof pl);
 cout << "Enter planet name (enter a blank line "
 "to quit):\n";
 cin.get(pl.name, 20);
 }
 fout.close();
}

```

```

// вывести измененный файл
fin.clear(); // не требуется для некоторых реализаций, но все же не помешает
fin.open(file, ios::in | ios::binary);
if (fin.is_open())
{
 cout << "Here are the new contents of the "
 << file << " file:\n";
 while (fin.read((char *) &p1, sizeof p1))
 {
 cout << setw(20) << p1.name << ": "
 << setprecision(0) << setw(12) << p1.population
 << setprecision(2) << setw(6) << p1.g << "\n";
 }
}
fin.close();
return 0;
}

```

Мы уже рассмотрели основные особенности программы, но давайте вновь вернемся к ним. В программе использован следующий код (в форме функции `eatline()`) после чтения параметров планеты:

```
while (cin.get() != '\n') continue;
```

Этот блок читает и отбрасывает оставшуюся часть строки до символа новой строки. Рассмотрим следующий оператор ввода внутри цикла:

```
cin.get(p1.name, 20);
```

Если символ новой строки остался на месте, этот оператор прочтет символ новой строки как пустую строку, прерывая, таким образом, цикл.

## Произвольный доступ к файлам

Как последний пример работы с файлами рассмотрим произвольный доступ. Он представляет возможность переместиться в любое место файла сразу, вместо последовательного передвижения по нему. Подход с произвольным доступом часто используется при обработке файлов баз данных. Программа поддерживает отдельный индексный файл, содержащий информацию о местоположении данных в основном файле. Затем она может перейти непосредственно к этому местоположению, прочесть данные и, возможно, изменить их. Этот подход проще реализовать, если файл состоит из набора записей одинакового размера. Каждая запись представляет собой набор связанных данных. Например, в предыдущем примере каждая запись представляла собой данные об определенной планете. Файловая запись наиболее естественно отвечает программной структуре или классу.

Мы будем базироваться на примере программы для работы с двоичным файлом из листинга 16.18, используя то преимущество, что структура `planet` обеспечивает удобный шаблон файловой записи. В примере файл бу-

дет открыт в режиме "чтения-записи", поэтому можно будет сразу и читать, и модифицировать запись. Это можно сделать, создав объект класса `fstream`. Класс `fstream` является производным от класса `iostream`, который, в свою очередь, базируется на классах `istream` и `ostream`, поэтому он наследует методы их обоих. Кроме того, он наследует два буфера: один — для ввода и один — для вывода, — и синхронизует обработку этих двух буферов. Это значит, что, когда программа читает файл или пишет в него, она передвигает и входной указатель в буфере ввода, и выходной указатель в буфере вывода одновременно.

В примере выполняется следующее:

1. Выводится текущее состояние файла `planets.dat`.
2. Запрашивается, какую запись необходимо модифицировать.
3. Модифицируется эта запись.
4. Выводится содержимое измененного файла.

"Более усердная" программа использовала бы меню и цикл, который бы позволил выбирать неограниченно долго в списке действий, но данная версия выполняет каждое действие только один раз. Этот упрощенный подход позволяет ознакомиться с несколькими аспектами чтения-записи файлов, не углубляясь в вопросы разработки программы.



### ПРЕДОСТЕРЕЖЕНИЕ

Данная программа подразумевает, что файл `planets.dat` уже существует и создан программой `binary.cpp`.

Один из первых вопросов — это какой режим использовать. Для того чтобы читать файл, необходим режим `ios_base::in`. Для осуществления двоичного ввода/вывода необходим режим `ios_base::binary`. (И снова, на некоторых нестандартных системах можно или даже нужно опустить этот режим.) Чтобы записать в файл, не-

обходим режим `ios_base::out` или `ios_base::app`. Однако режим добавления позволяет программе добавлять данные только к концу файла. Вся остальная часть файла доступна только для чтения; т.е. исходные данные можно прочесть, но нельзя модифицировать, поэтому нужно воспользоваться режимом `ios_base::out`. Как показывает табл. 16.9, использование режимов `in` и `out` одновременно представляет собой режим чтения/записи, поэтому к ним достаточно добавить лишь двоичный \* режим. Как упоминалось ранее, для комбинации режимов используется оператор `|`. Таким образом, для произвольного доступа необходим следующий оператор:

```
finout.open(file,ios_base::in |
 ios_base::out | ios_base::binary);
```

Далее нужно задать способ передвижения по файлу. Класс `fstream` наследует два метода для этого: `seekg()` передвигает указатель ввода, `seekp()` — указатель вывода в указанную точку файла. (Фактически, поскольку класс `fstream` использует буферы для промежуточного хранения данных, указатели указывают на буфер, а не на реальный файл.) Кроме того, метод `seekg()` можно использовать с объектом класса `ifstream`, а метод `seekp()` — с объектом класса `ofstream`. Вот прототипы метода `seekg()`:

```
basic_istream<charT,traits>& seekg(off_type,
 ios_base::seekdir);
basic_istream<charT,traits>&
 seekg(pos_type);
```

Как вы видите, они являются шаблонами. В данной главе используется символьная специализация шаблона, т.е. тип `char`. Для `char`-специализации эти два прототипа эквивалентны следующим:

```
istream & seekg(streamoff,
 ios_base::seekdir);
istream & seekg(streampos);
```

Первый прототип осуществляет поиск позиции в файле, отстоящей от позиции, заданной вторым аргументом, на указанное количество байтов. Второй прототип осуществляет поиск позиции в файле, отстоящей от начала файла на указанное количество байтов.

### РАСШИРЕНИЕ ТИПОВ

Когда C++ только появился, методы `seekg()` применялись проще. Типы `streamoff` и `streampos` были определены оператором `typedef` для некоторых стандартных целых типов, например, для типа `long`. Однако задача создания переносимого стандарта привела к выводу, что аргумент целого типа не может передать достаточно информации для некоторых файловых систем, поэтому `streamoff` и `streampos` стали структурами или классами, хотя они позволяют производить некоторые базовые операции, например, использовать значение целого типа в качестве инициализационного. Затем старый класс `istream` был заменен на шаблон `basic_istream`, а `streampos` и `streamoff` были

заменены на базирующиеся на шаблоне типы `pos_type` и `off_type`. Однако `streampos` и `streamoff` продолжают существовать как символьные специализации `pos_type` и `off_type`. Аналогично можно использовать типы `wstreampos` и `wstreamoff`, если метод `seekg()` применяется для объекта класса `wistream`.

Давайте рассмотрим аргументы первого прототипа функции `seekg()`. Значения типа `streamoff` применяются для измерения в байтах расстояния смещения от определенного места в файле. Аргумент `streamoff` представляет собой позицию, отстоящую от одной из трех точек на указанное количество байтов. (Тип может быть определен как интегральный тип или как класс.) Аргумент `seek_dir` — это другой целый тип, определенный вместе с тремя возможными значениями в классе `ios_base`. Константа `ios_base::beg` означает, что смещение отсчитывается от начала файла. Константа `ios_base::cur` означает, что смещение отсчитывается от текущей позиции. Константа `ios_base::end` показывает, что смещение отсчитывается от конца файла.

Вот примеры вызова функции, в которых подразумевается, что `fin` является объектом класса `ifstream`:

```
// 30 байтов от начала файла
fin.seekg(30, ios_base::beg);

// один байт назад от текущей позиции
fin.seekg(-1, ios_base::cur);

// переход к концу файла
fin.seekg(0, ios_base::end);
```

Теперь рассмотрим второй прототип. Значения типа `streampos` определяют позицию в файле. Этот тип может быть классом, в этом случае класс включает в себя конструктор с аргументом `streamoff` и конструктор с аргументом целого типа и обеспечивает способ преобразования обоих типов в значение типа `streampos`. Значение типа `streampos` представляет собой абсолютную величину позиции в файле, отстоящую от начала файла на указанное количество байтов. Значение позиции `streampos` можно трактовать, как смещение от начала файла, где первый байт имеет индекс 0. Поэтому оператор

```
fin.seekg(112);
```

передает файловый указатель на 112-й байт, который является реальным 113-м байтом файла. Если необходимо проверить текущую позицию файлового указателя, можно воспользоваться методом `tellg()` для потока ввода и `tellp()` — для потока вывода. Каждый из них возвращает значение типа `streampos`, представляющее собой текущую позицию, отстоящую от начала файла на указанное количество байтов. Когда создается объект класса `fstream`, входной и выходной указатели передвигаются одновременно, поэтому в таком случае функции `tellg()` и `tellp()` возвращают одинаковое значение. Но

если используется объект класса **istream** для управления потоком ввода и объект класса **ostream** для управления потоком вывода, входной и выходной указатели передвигаются независимо друг от друга и функции **tellg()** и **tellp()** могут возвращать различные значения.

Функции **seekg()** можно использовать для перехода к началу файла. Вот блок кода, в котором открывается файл, файловый указатель перемещается в начало файла и выводится содержимое файла:

```
fstream finout; // поток для чтения и записи
finout.open(file,ios::in | ios::out |
 ios::binary);
// ПРИМЕЧАНИЕ. Некоторые разновидности системы
// UNIX требуют отбросить | ios::binary
int ct = 0;
if (finout.is_open())
{
 finout.seekg(0); //перейти к началу файла
 cout << "Here are the current contents of the "
 << file << " file:\n";
 while (finout.read((char *) &p1, sizeof p1))
 {
 cout << ct++ << ": " << setw(20)
 << p1.name << ":" "
 << setprecision(0) << setw(12)
 << p1.population
 << setprecision(2) << setw(6)
 << p1.g << "\n";
 }
 if (finout.eof())
 finout.clear(); // сбросить флаг eof
 else
 {
 cerr << "Error in reading "
 << file << ".\n";
 exit(1);
 }
}
else
{
 cerr << file << " could not be opened-bye.\n";
 exit(2);
}
```

Это аналогично началу листинга 16.18, но в данном коде есть некоторые изменения и дополнения. Прежде всего, как было только что описано, программа использует объект класса **fstream** с режимом чтения/записи, а также функцию **seekg()** для позиционирования в начало файла. (В действительности это не нужно для данной программы, но это пример того, как работает функция **seekg()**.) Затем программа производит небольшие изменения в нумерации записей при выводе. В программе также было сделано следующее важное дополнение:

```
if (finout.eof())
 finout.clear(); // сбросить флаг eof
else
{
 cerr << "Error in reading " << file << ".\n";
 exit(1);
}
```

Проблема заключается в том, что, поскольку программа считывает и выводит файл целиком, она устанавливает бит **eofbit**. При этом программе сообщается о том, что она закончила обработку файла и запрещает какой-либо последующий доступ к файлу. Используя метод **clear()**, можно изменить состояние потока, отключая бит **eofbit**. После этого программа может вновь получить доступ к файлу. Часть программного кода с оператором **else** реализует такую возможность, что программа прекращает читать файл не потому, что достигла его конца, а по какой-то другой причине, например, из-за аппаратной проблемы.

Следующий шаг — идентифицировать запись, которую нужно изменить, а затем модифицировать ее. Для этого программа запрашивает пользователя ввести номер записи. Умножая это число на количество байтов в записи, можно определить величину смещения относительно начала записи в файле. Если **record** — это номер записи, то необходимый номер байта определяется выражением **record \* sizeof pl**:

```
cout << "Enter the record number you
 wish to change: ";
long rec;
cin >> rec;
eatline(); //избавиться от символа новой строки
if (rec < 0 || rec >= ct)
{
 cerr << "Invalid record number-bye\n";
 exit(3);
}
// преобразовать в тип streampos
streampos place = rec * sizeof pl;
finout.seekg(place); //произвольный доступ
```

Переменная **ct** представляет собой количество записей; программа завершает выполнение при попытке выйти за пределы файла.

Затем программа выводит текущую запись:

```
finout.read((char *) &p1, sizeof p1);
cout << "Your selection:\n";
cout << rec << ": " << setw(20)
 << p1.name << ":" "
 << setprecision(0) << setw(12)
 << p1.population
 << setprecision(2) << setw(6)
 << p1.g << "\n";
if (finout.eof())
 finout.clear(); //сбросить флаг eof
```

После вывода записи на экран программа позволяет изменить запись:

```
cout << "Enter planet name: ";
cin.get(p1.name, 20);
eatline();
cout << "Enter planetary population: ";
cin >> p1.population;
cout << "Enter planet's acceleration of gravity: ";
```

```

cin >> pl.g;
finout.seekp(place); // возврат
finout.write((char *) &pl, sizeof pl) << flush;
if (finout.fail())
{
 cerr << "Error on attempted write\n";
 exit(5);
}

```

Программа очищает буфер вывода, чтобы гарантировать, что файл обновлен до того, как будет осуществлен переход к следующей стадии.

И наконец, чтобы вывести на экран содержимое обновленного файла, программа использует функцию `seekg()`, что позволяет переместить файловый указатель на начало файла. В листинге 16.19 показана программа полностью. Не забудьте, что в ней предполагается, что файл `planets.dat`, созданный программой `binary.cpp`, доступен.

### ЗАМЕЧАНИЕ ПО ПОВОДУ СОВМЕСТИМОСТИ

Более ранние реализации, скорее всего, не поддерживают текущий стандарт. Некоторые системы не поддерживают двоичный флаг. Symantec C++ добавляет новый ввод вместо замещения указанной записи. Кроме того, Symantec C++ требует заменить оператор (дважды)

```

while (fin.read((char *) &pl, sizeof pl))
следующим:
while (fin.read((char *) &pl, sizeof pl) &&
!fin.eof())

```

Вот результаты выполнения программы (см. листинг 16.19), которая базируется на файле `planets.dat`, имеющем несколько больше записей, чем при его последнем просмотре:

```

Here are the current contents of the
planets.dat File:
0: Earth: 5333000000 9.81
1: Bill's Planet: 23020020 8.82
2: Trantor: 580000000000 15.03
3: Trellan: 4256000 9.62
4: Freestone: 3845120000 8.68
5: Taanagoot: 350000002 10.23
6: Marin: 232000 9.79
Enter the record number you wish to change: 2
Your selection:
2: Trantor: 580000000000 15.03
Enter planet name: Trantor
Enter planetary population: 59500000000
Enter planet's acceleration of gravity: 10.53
Here are the new contents of the planets.dat file:
0: Earth: 5333000000 9.81
1: Bill's Planet: 23020020 8.82
2: Trantor: 595000000000 10.53
3: Trellan: 4256000 9.62
4: Freestone: 3845120000 8.68
5: Taanagoot: 350000002 10.23
6: Marin: 232000 9.79

```

Используя методы из данной программы, ее можно расширить, чтобы она позволяла добавлять новые дан-

ные и удалять записи. Если вы захотите расширить программу, то можно использовать классы и функции. Например, можно преобразовать структуру `planet` в класс; затем перегрузить операцию вставки `<<`, чтобы выражение `cout << pl` выводило на экран элементы класса, отформатированные как в этом примере.

## Внутреннее форматирование

Семейство классов `iostream` обеспечивает операции ввода/вывода между программой и терминалом. Семейство `fstream` использует такой же интерфейс для обеспечения операции ввода/вывода между программой и файлом. Кроме того, библиотека C++ имеет семейство классов `sstream`, которое использует такой же интерфейс для обеспечения ввода/вывода между программой и объектом класса `string`. Это значит, что те же самые методы класса `ostream`, которые использовались при работе с объектом `cout`, можно использовать и для передачи информации в объект класса `string`, а методы класса `istream`, например `getline()`, — для чтения информации из объекта класса `string`. Процесс чтения отформатированной информации из объекта `string` или записи отформатированной информации в объект `string` называется *внутренним форматированием*. Далее мы кратко рассмотрим его свойства. (Семейство `sstream` для поддержки класса `string` заменяет семейство классов из заголовочного файла `strstream.h` для поддержки массива типа `char`.)

Заголовочный файл `sstream` определяет класс `ostringstream`, производный от класса `ostream`. (Существует также и класс `wostream`, базирующийся на классе `wostream`, созданном для работы с набором 16-разрядных символов.) Если создать объект класса `ostringstream`, то в него можно записать информацию. Для объекта `ostringstream` можно пользоваться теми же самыми методами, что и для объекта `cout`, т.е. можно сделать следующее:

```

ostringstream outstr;
double price = 55.00;
char * ps = " for a copy of the draft C++"
 "standard!";
outstr.precision(2);
outstr << fixed;
outstr << "Pay only $" << price << ps << endl;

```

Отформатированный текст будет направлен в буфер, причем объект выполняет динамическое распределение памяти, чтобы увеличить буфер при необходимости. Класс `ostringstream` имеет функцию-элемент, названную `str()`, которая возвращает объект класса `string`, инициализированный содержимым буфера:

```

// возвращает строку с отформатированной
// информацией
string mesg = outstr.str();

```

## Листинг 16.19 Программа random.cpp.

```

// random.cpp - произвольный доступ к двоичному файлу
#include <iostream> // не требуется для большинства систем
using namespace std;
#include <fstream>
#include <iomanip>
#include <cstdlib> // (или stdlib.h) для функции exit()

struct planet
{
 char name[20]; // название планеты
 double population; // ее население
 double g; // ускорение свободного падения на ней
};

const char * file = "planets.dat"; //ПРЕДПОЛАГАЕТСЯ, ЧТО ФАЙЛ СУЩЕСТВУЕТ (пример с binary.cpp)
inline void eatline() { while (cin.get() != '\n') continue; }

int main()
{
 planet pl;
 cout << fixed;

 // вывести исходное содержимое файла
 fstream finout; // потоки чтения и записи
 finout.open(file,ios::in | ios::out | ios::binary);
 //ПРИМЕЧАНИЕ. Некоторые разновидности системы UNIX требуют опустить | ios::binary
 int ct = 0;
 if (finout.is_open())
 {
 finout.seekg(0); // перейти к началу файла
 cout << "Here are the current contents of the "
 << file << "\n";
 while (finout.read((char *) &pl, sizeof pl))
 {
 cout << ct++ << ": " << setw(20) << pl.name << ":"
 << setprecision(0) << setw(12) << pl.population
 << setprecision(2) << setw(6) << pl.g << "\n";
 }
 if (finout.eof())
 finout.clear(); // сбросить флаг eof
 else
 {
 cerr << "Error in reading " << file << ".\n";
 exit(1);
 }
 }
 else
 {
 cerr << file << " could not be opened - bye \n";
 exit(2);
 }

 // изменить запись
 cout << "Enter the record number you wish to change: ";
 long rec;
 cin >> rec;
 eatline(); // избавиться от символа новой строки
 if (rec < 0 || rec >= ct)
 {
 cerr << "Invalid record number - bye\n";
 exit(3);
 }
 streampos place = rec * sizeof pl; // преобразовать в тип streampos
 finout.seekg(place); // произвольный доступ к файлу
 if (finout.fail())
 {
 cerr << "Error on attempted seek\n";
 exit(4);
 }
 finout.read((char *) &pl, sizeof pl);
}

```

```

cout << "Your selection:\n";
cout << rec << ":" << setw(20) << pl.name << ":" "
 << setprecision(0) << setw(12) << pl.population
 << setprecision(2) << setw(6) << pl.g << "\n";
if (finout.eof())
 finout.clear(); // сбросить флаг eof

cout << "Enter planet name: ";
cin.get(pl.name, 20);
eatline();
cout << "Enter planetary population: ";
cin >> pl.population;
cout << "Enter planet's acceleration of gravity: ";
cin >> pl.g;
finout.seekp(place); // возврат
finout.write((char *) &pl, sizeof pl) << flush;
if (finout.fail())
{
 cerr << "Error on attempted write\n";
 exit(5);
}

// вывести содержимое измененного файла
ct = 0;
finout.seekg(0); // перейти к началу файла
cout << "Here are the new contents of the " << file
 << " file:\n";
while (finout.read((char *) &pl, sizeof pl))
{
 cout << ct++ << ":" << setw(20) << pl.name << ":" "
 << setprecision(0) << setw(12) << pl.population
 << setprecision(2) << setw(6) << pl.g << "\n";
}
finout.close();
return 0;
}

```

При использовании метода `str()` объект "замораживается" — после этого в него уже нельзя ничего записать.

Листинг 16.20 содержит короткий пример.

Результаты выполнения программы из листинга 16.20:

```

What's the name of your hard disk? Rocky
What's its capacity in MB? 2425
The hard disk Rocky has a capacity of 2425
megabytes.

```

Класс `istringstream` позволяет использовать методы семейства классов `istream` для чтения данных из объекта класса `istringstream`, который можно инициализировать строковым объектом. Пусть `facts` — это объект класса `string`. Для создания объекта класса `istringstream`, ассоциированного с этой строкой, необходимо сделать следующее:

```

// использовать строку facts для
// инициализации потока
istringstream instr(facts);

```

Затем используются методы класса `istream` для чтения данных из `instr`. Например, если `instr` содержит набор целых чисел в символьном формате, то их можно прочесть с помощью следующего программного кода:

```

int n;
int sum = 0;
while (instr << n)
 sum += num;

```

В листинге 16.21 использована перегруженная операция `>>` для чтения содержимого строки по одному слову.

Результаты выполнения программы из листинга 16.21:

```

It
was
a
dark
and
stormy
day,
and
the
full
moon
glowed
brilliantly.

```

Итак, классы `istringstream` и `ostringstream` дают возможность использовать методы классов `istream` и `ostream` для управления символьными данными, сохраненными в строках.

**Листинг 16.20 Программа shout.cpp.**

```
// shout.cpp - внутренне форматирование (вывод)

#include <iostream>
using namespace std;
#include <sstream>
#include <string>
int main()
{
 ostringstream outstr; // управляет строковым потоком

 string hdisk;
 cout << "What's the name of your hard disk? ";
 getline(cin, hdisk);
 int cap;
 cout << "What's its capacity in MB? ";
 cin >> cap;

 // записать отформатированную информацию в строковый поток
 outstr << "The hard disk " << hdisk << " has a capacity of "
 << cap << " megabytes.\n";
 string result = outstr.str(); // сохранить результат
 cout << result; // вывести содержимое

 return 0;
}
```

**Листинг 16.21 Программа strin.cpp.**

```
// strin.cpp - форматированное чтение из массива типа char

#include <iostream>
using namespace std;
#include <sstream>
#include <string>
int main()
{
 string lit = "It was a dark and stormy day, and "
 " the full moon glowed brilliantly. ";
 istringstream instr(lit); // использовать buf для ввода
 string word;;
 while (instr >> word) // считывание по одному слову
 cout << word << endl;
 return 0;
}
```

## Что дальше?

Если вы хорошо проработали эту книгу, значит, достигли хорошего понимания основных правил C++. Однако это только начало изучения языка. Вторая стадия изучения состоит в эффективном его использовании, и это долгое путешествие. Лучше всего оказаться в среде разработки или изучения языка, что позволит вам находиться в постоянном контакте с хорошим кодом C++ и программистами. Кроме того, теперь, когда вы знакомы с C++, вы можете читать книги, посвященные более сложным темам, в том числе объектно-ориентированному программированию. В приложении Н перечислены некоторые из подобных ресурсов.

Одна из перспектив ООП — это упрощение разработки и повышение надежности больших проектов. Одно из существенных действий в объектно-ориентированном подходе — это разработка классов, которые отражают реальную ситуацию (называемую *областью проблем*), которая подлежит моделированию. Поскольку реальные задачи зачастую сложны, поиск удобного набора классов может оказаться непростым делом. Создание сложной системы наспех обычно не дает рабочего результата; вместо этого лучше воспользоваться итеративным эволюционным подходом. В этом направлении практики в данной области разработано несколько методов и стратегий. В частности, важно производить столько итераций и эволюционных изменений в фазах анализа и дизайна, сколько возможно, а не писать и переписывать реальный код.

Два распространенных метода — это *анализ способов применения* и *CRC-карты*. При анализе способов применения разработчики рассматривают общие пути, или сценарии, по которым будет использоваться конечная система. Для этого производится идентификация элементов, действий и обязанностей, из которых могут быть выведены классы и их свойства. Метод CRC-карты (аббревиатура от слов Class/Responsibilities/Collaborators — Классы/Обязанности/Сотрудники) — это простой способ для анализа подобных сценариев. Разработчики создают индексную карту для каждого класса. Карта содержит описание имени класса, функции класса, в частности, представление данных и производимые действия, а также описывает сотрудников класса либо другие классы, с которыми данный класс должен взаимодействовать. Затем разработчики изучают сценарии, используя интерфейс, обеспечиваемый CRC-картами. Это может вести к поиску новых классов, перераспределению обязанностей и т.д.

Более масштабными являются систематические методы для работы над проектом в целом. Наиболее современный из них — это Unified Modeling Language (Уни-

фицированный язык моделирования), или UML. Это не язык программирования; скорее это язык для представления анализа и разработки программируемого проекта. Его разработали Гради Буч (Grady Booch), Джим Рамба (Jim Rumbaugh) и Айвар Якобсон (Ivar Jacobson), которые были первоначальными разработчиками трех более ранних языков объектного моделирования: Booch Method, OMT (Object Modeling Technique — Метод объектного моделирования) и OOSE (Object-Oriented Software Engineering — Объектно-ориентированное проектирование программного обеспечения) соответственно. UML является продуктом эволюции всех этих языков.

Для повышения общего уровня понимания C++ необходимо ознакомиться с определенными библиотеками классов. Microsoft, Borland и Symantec, например, предлагают широкий набор библиотек классов, способствующих программированию в среде Windows, а Symantec и Metrowerks предлагают подобный набор для программирования под Macintosh.

## Резюме

Поток представляет собой процесс передачи байтов в программу или из нее. Буфер — это область памяти для временного хранения данных, которая действует как посредник между программой и файлом или другими устройствами ввода/вывода. Информацию можно пересыпать между буфером и файлом большими блоками, имеющими размер, который наиболее эффективно обрабатывается устройствами наподобие дисковых накопителей. Между буфером и программой информацией может передаваться побайтово, что удобнее для внутривыборки обработки информации. C++ обрабатывает поток ввода, соединяя буферизованный поток с программой и с источником ввода. Подобным же образом C++ соединяет буферизованный поток с программой и точкой вывода. Файлы `iostream` и `fstream` представляют собой библиотеку классов ввода/вывода с широким набором классов для управления потоками. Программы, написанные на C++ и включающие файл `iostream`, автоматически открывают восемь потоков, управляемых восьмью объектами. Объект `cin` управляет потоком стандартного ввода, который по умолчанию соединен со стандартным устройством ввода, обычно с клавиатурой. Объект `cout` управляет стандартным потоком вывода, который по умолчанию соединен со стандартным устройством вывода, обычно монитором. Объекты `cerr` и `clog` управляют небуферизованным и буферизованным потоками, соединенными со стандартным устройством вывода ошибок, обычно монитором. Эти четыре объекта имеют четыре эквивалента для 16-битовых символов,

которые называются соответственно `wcin`, `wcout`, `wcerr` и `wclog`.

Библиотека классов ввода/вывода имеет множество полезных методов. Класс `istream` определяет несколько версий операции извлечения (`>>`), который распознает все базовые типы C++ и конвертирует символьный ввод в какой-либо из этих типов. Семейство методов `get()` и метод `getline()` обеспечивают поддержку односимвольного и строкового ввода. Аналогично класс `ostream` определяет версии операции вставки (`<<`), которую распознают все базовые типы C++, и преобразует их в соответствующий символьный вывод. Метод `put()` обеспечивает поддержку односимвольного вывода. Классы `wistream` и `wostream` обеспечивают подобную поддержку для 16-разрядных символов.

Управлять форматированием программного вывода можно, используя методы класса `ios_base` и манипуляторы (функции, которые можно конкатенировать с помощью оператора вставки), определенные в файлах `iostream` и `iomanip`. Эти методы и манипуляторы позволяют контролировать систему счисления, ширину поля, количество выводимых цифр после десятичной точки, системы вывода чисел с плавающей точкой и подобные элементы.

Файл `fstream` содержит определения классов, которые расширяют методы класса `iostream` для файлового ввода/вывода. Класс `ifstream` является производным от класса `istream`. Поставив объект класса `ifstream` в соответствие файлу, можно использовать все методы класса `istream` для чтения файла. Аналогично ассоциирование объекта класса `ofstream` с файлом позволяет использовать методы класса `ostream` для записи в файл. И наконец, ассоциирование объекта класса `fstream` с файлом позволяет воспользоваться методами файлового ввода/вывода.

Чтобы поставить файл в соответствие потоку, можно инициализировать файловый потоковый объект именем файла. Другой вариант — можно сначала создать файловый потоковый объект, а затем воспользоваться методом `open()`, чтобы ассоциировать поток с файлом. Метод `close()` разрывает связь между потоком и файлом. Конструкторы класса и метод `open()` имеют дополнительный второй аргумент, задающий режим работы с файлом. Режим файла определяет, используется ли файл для чтения и/или записи, урезается ли файл до нуля при его открытии для записи, приводит ли к ошибке попытка открытия несуществующего файла и какой режим доступа к файлу будет использоваться — двоичный или текстовый.

В текстовом файле вся информация сохранена в символьной форме. Например, числовые значения преобразуются в их символьное представление. Обычные опе-

раторы вставки и извлечения, а также функции `get()` и `getline()` поддерживают этот режим. В двоичном файле информация сохранена в том же двоичном представлении, которое внутренне использует компьютер для хранения информации. В двоичных файлах информация, в частности значения типов с плавающей точкой, сохранена более точно и компактно по сравнению с текстовыми файлами, но они являются менее переносимыми. Методы `read()` и `write()` поддерживают двоичный ввод и вывод соответственно.

Функции `seekg()` и `seekp()` обеспечивают в C++ произвольный доступ к файлам. Эти методы класса позволяют позиционировать файловый указатель относительно начала файла, конца файла или текущей позиции в файле. Методы `tellg()` и `tellp()` возвращают текущую позицию в файле.

В заголовочном файле `sstream` определены классы `istringstream` и `ostringstream`, позволяющие использовать методы классов `istream` и `ostream` для извлечения информации из строки или форматирования информации, помещаемой в строку.

## Вопросы для повторения

1. Какую роль файл `iostream` играет в вводе/выводе C++?
2. Почему ввод числа, например 121, требует, чтобы программа выполнила преобразование?
3. В чем разница между стандартным потоком ввода и стандартным потоком вывода ошибок?
4. Почему объект `cout` способен выводить различные типы данных C++, не требуя указания явных инструкций для каждого типа?
5. Какое свойство определений методов вывода позволяет конкатенировать вывод?
6. Напишите программу, которая запрашивает целое число и выводит его в десятичной, восьмеричной и шестнадцатиричной формах. Выведите каждую форму на экран в одной и той же строке в поле шириной 15 символов и используйте префиксы C++ для систем счисления.
7. Напишите программу, которая запрашивает информацию и форматирует ее, как показано ниже:

```
Enter your name: Billy Gruff
Enter your hourly wages: 12
Enter number of hours worked: 7.5
First format:
 Billy Gruff: $ 12.00: 7.5
Second format:
 Billy Gruff : $12.00 : 7.5
```

8. Рассмотрите следующую программу:

```
//rq16-8.cpp
#include <iostream>
using namespace std;

int main()
{
 char ch;
 int ct1 = 0;

 cin >> ch;
 while (ch != 'q')
 {
 ct1++;
 cin >> ch;
 }

 int ct2 = 0;
 cin.get(ch);
 while (ch != 'q')
 {
 ct2++;
 cin.get(ch);
 }
 cout << "ct1 = " << ct1
 << "; ct2 = " << ct2 << "\n";
 return 0;
}
```

Что она напечатает, если ввести следующее:

```
I see a q <Enter>
I see a q <Enter>
```

Здесь <Enter> обозначает результат нажатия клавиши Enter.

9. Оба из приведенных ниже операторов читают и отбрасывают символы до конца строки. Определите, чем все-таки они различаются.

```
while (cin.get() != '\n')
 continue;
cin.ignore(80, '\n');
```

## Упражнения по программированию

- Напишите программу, подсчитывающую количество символов в потоке ввода до первого символа \$, при чем данный символ должен остаться в потоке ввода.
- Напишите программу, копирующую ввод с клавиатуры (до эмулируемого символа конца файла) в файл с именем, задаваемым в командной строке.
- Напишите программу, копирующую один файл в другой. Программа должна получать имена файлов в командной строке. Если программа не может открыть файл, она должна сообщить об этом.
- Напишите программу, которая открывает два текстовых файла для ввода и один — для вывода. Программа конкatenирует соответствующие строки во входных файлах, используя в качестве символа-раз-

делителя пробел, и записывает результат в выходной файл. Если один файл короче, чем другой, то оставшиеся строки в более длинном файле необходимо просто скопировать в выходной файл. Например, пусть первый входной файл имеет следующее содержимое:

```
eggs kites donuts
balloons hammers
stones
```

А второй файл — следующее:

```
zero lassitude
finance drama
```

Тогда результирующий файл должен содержать:

```
eggs kites donuts zero lassitude
balloons hammers finance drama
stones
```

5. Мэт и Пэт хотят пригласить своих друзей на вечеринку, как это было в упражнении 5 главы 15, с той лишь разницей, что они хотят, чтобы программа использовала файлы. Они просят вас написать программу, способную выполнять следующее:

Читать список друзей Мэта из текстового файла под названием mat.dat, в котором в каждой строке по одному перечислены имена его друзей. Имена должны быть помещены в контейнер и затем выведены в отсортированном порядке.

Читать список друзей Пэт из текстового файла под названием pat.dat, в котором в каждой строке по одному перечислены имена ее друзья. Имена должны быть помещены в контейнер и затем выведены в отсортированном порядке.

Объединить оба списка, исключив повторы, и сохранить результат в файле matpat.dat, по одному имени в строке.

6. Представьте себе определения классов из упражнения 13.5. Если вы еще не выполняли это упражнение, сделайте это сейчас. Затем выполните следующее:

Напишите программу, которая использует стандартный и файловый ввод/вывод C++ в сочетании с типами данных employee, manager, fink и highfink, определенными так же, как в упражнении 13.5. Программа должна в основном повторять программу из листинга 16.17. Кроме того, она должна позволять добавлять новые данные в файл. При первом запуске программа должна запросить данные у пользователя, вывести на экран все записи и сохранить информацию в файле. При последующих случаях использования программы она должна сначала читать и отображать данные, а затем позволить пользователю добавить данные и

после этого отображать все данные. Одно из различий между двумя вариантами программ состоит в том, что данные должны обрабатываться как массив указателей на тип `employee`. Указатель может указывать на объект `employee` или на любой из объектов одного из трех производных типов. Массив должен оставаться небольшим, чтобы не усложнять проверку программы:

```
const int MAX = 10; //не более 10 объектов
...
employee * pc[MAX];
```

Для ввода с клавиатуры программа должна использовать меню, с помощью которого пользователь может выбрать, какой тип объекта создавать. Меню должно использовать оператор `new` для создания объекта требуемого типа и присваивать адрес объекта указателю из массива `pc`. После этого такой объект может использовать виртуальную функцию `setall()` для получения необходимой информации от пользователя:

```
pc[i]->setall(); //вызывает функцию,
//отвечающую типу объекта
```

Для сохранения данных в файле необходимо создать виртуальную функцию `writeall()`:

```
for (i = 0; i < index; i++)
 pc[i]->writeall(fout); //объект fout
 //класса ofstream соединен
 //с выходным файлом
```

### ПРИМЕЧАНИЕ

В упражнении необходимо использовать текстовый, а не двоичный ввод/вывод. (К сожалению, виртуальные объекты содержат указатели на таблицы указателей, и метод `write()` копирует и эту информацию в файл. Объект, наполненный информацией из файла с помощью метода `read()`, получает неверные значения указателей на функции, что искажает поведение виртуальных функций.) Для разделения полей данных используйте символ новой строки; таким образом проще определить поля для ввода. Другой вариант — можно выполнять двоичный ввод/вывод, но не записывать объекты как целое. Вместо этого можно создать методы класса, которые применяют функции `write()` и `read()` для каждого элемента класса отдельно, а не для объекта в целом. Таким способом программа может сохранить в файле только нужную информацию.

Более сложная часть — это чтение данных из файла. Проблема заключается в том, как программа может определить, что представляет собой следующий элемент данных, — объект `employee`, объект `manager`, тип `fink` или же тип `highfink`? Один из подходов состоит в том, чтобы перед записью данных в файл записывать целое число, определяющее, какой именно объект сохранен за ним в файле. Затем при чтении файла программа может прочесть число и, используя оператор `switch`, создать объект соответствующего типа и сохранить в нем данные:

```
enum classkind{ Employee, Manager, Fink,
 Highfink} ; // в заголовке класса
...
int classtype;
while((fin >> classtype).get(ch))
{ // символ новой строки разделяет
 // число типа int и данные
 switch(classtype)
 {
 case Employee : pc[i] = new employee;
 : break;
```

Затем можно воспользоваться указателем, чтобы вызвать виртуальную функцию `getall()` для чтения информации:

```
pc[i++]->getall();
```

# Системы счисления

**С**тандартный метод записи чисел основан на степенях числа 10. Например, рассмотрим число 2468. Цифра 2 обозначает 2 тысячи, 4 обозначает 4 сотни, 6 соответствует 6 десяткам, а 8 — соответственно 8 единицам:

$$2468 = 2 \cdot 1000 + 4 \cdot 100 + 6 \cdot 10 + 8 \cdot 1$$

Тысяча — это  $10 \cdot 10 \cdot 10$ , или  $10^3$ , или 10 в степени 3. С помощью подобной формы записи предыдущее соотношение можно записать следующим образом:

$$2468 = 2 \cdot 10^3 + 4 \cdot 10^2 + 6 \cdot 10^1 + 8 \cdot 10^0$$

Поскольку такой способ записи чисел основан на степенях числа 10, мы называем его десятичной записью. Но ведь для записи чисел можно выбрать любое другое число. В C++ для записи целых числе можно использовать степени числа 8 (восьмеричную) и степени числа 16 (шестнадцатиричную) систему. (Обратите внимание, что  $10^0$  равняется 1, как и любое ненулевое число в нулевой степени.)

## Восьмеричные числа

Восьмеричные числа основаны на степенях числа 8, в системе записи с основанием 8 используются цифры от 0 до 7. Признаком восьмеричной системы в C++ является использование 0 в качестве префикса. Поэтому 0177 — это восьмеричное значение. Чтобы найти десятичный эквивалент этому числу, необходимо воспользоваться следующим алгоритмом:

$$\begin{aligned} 0177 \text{ (восьмеричное число)} &= 1 \cdot 8^2 + 7 \cdot 8^1 + 7 \cdot 8^0 \\ &= 1 \cdot 64 + 7 \cdot 8 + 7 \cdot 1 \\ &= 127 \text{ (десятичное)} \end{aligned}$$

В операционных системах UNIX часто используется восьмеричное представление данных, что неразрывно связано с языками C++ и C.

## Шестнадцатиричные числа

Шестнадцатиричные числа основаны на степени числа 16. Поэтому 10 в шестнадцатиричной системе счисления соответствует значению  $16+0$ , или 16. Чтобы записать значения между 9 и шестнадцатиричным числом 16, необходимы дополнительные цифры. В стандартных системах для этой цели используются буквы латинского алфавита от a до f. В C++ допускается использование как прописных, так и строчных литер, что подтверждено в табл. А.1

Таблица А.1 Шестнадцатиричные числа.

| Шестнадцатиричные цифры | Десятичная величина |
|-------------------------|---------------------|
| a или A                 | 10                  |
| b или B                 | 11                  |
| c или C                 | 12                  |
| d или D                 | 13                  |
| e или E                 | 14                  |
| f или F                 | 15                  |

Чтобы отразить тот факт, что число записано в шестнадцатиричной системе, в C++ используют префикс 0x или 0X. Поэтому значение 0x2B3 является шестнадцатиричным числом. Чтобы отыскать его десятичный эквивалент, необходимо вычислить соответствующие степени 16:

$$\begin{aligned} 0x2B3 \text{ (шестнадцатиричное)} &= 2 \cdot 16^2 + 11 \cdot 16^1 + 3 \cdot 16^0 \\ &= 2 \cdot 256 + 11 \cdot 16 + 3 \cdot 1 \\ &= 691 \text{ (десятичное число)} \end{aligned}$$

В документации, сопровождающей аппаратные устройства, часто приходится использовать шестнадцатиричное представление данных для указания ячейки памяти или номера порта.

## Двоичные числа

Независимо от используемой системы представления данных, десятичной, восьмеричной или шестнадцатиричной, компьютер хранит информацию в виде двоичных кодов, используя в качестве основания число 2. В двоичной записи участвуют только две цифры, 0 и 1. В качестве примера рассмотрим двоичное число 10011011. Следует отметить, что в C++ не предусмотрена возможность двоичного представления величин. Двоичная запись основана на степенях цифры 2:

$$\begin{aligned}10011011 &= 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 \\&+ 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\&= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1 \\&= 155\end{aligned}$$

Двоичная форма удобна для хранения величин в памяти компьютера, где каждому отдельному блоку, называемому битом, можно назначить состояние "да" или "нет". А теперь просто отождествляйте параметр "нет" с 0, а параметр "да" — с 1. Ячейки памяти обычно организованы в блоки по 8 битов, называемые байтами. Биты в байте нумеруются согласно соответствующей степени 2. Поэтому самый крайний справа бит называется нулевым, следующий — битом номер 1 и т.д. Например, на рис. A.1 приведено 2-байтовое целое число.

## Двоичные и шестнадцатиричные числа

Шестнадцатиричную запись благодаря простому виду данных предпочитают двоичной записи в случаях, когда необходимо указать адреса ячеек памяти или целочисленные величины, содержащие битово-флаговые установочные параметры. В табл. A.2 можно проследить это соответствие.

Таблица A.2 Шестнадцатиричные цифры и их двоичные эквиваленты.

| Шестнадцатиричная цифра | Двоичный эквивалент |
|-------------------------|---------------------|
| 0                       | 0000                |
| 1                       | 0001                |
| 2                       | 0010                |
| 3                       | 0011                |
| 4                       | 0100                |
| 5                       | 0101                |
| 6                       | 0110                |
| 7                       | 0111                |
| 8                       | 1000                |
| 9                       | 1001                |
| A                       | 1010                |
| B                       | 1011                |
| C                       | 1100                |
| D                       | 1101                |
| E                       | 1110                |
| F                       | 1111                |

Чтобы преобразовать шестнадцатиричную величину в двоичную, необходимо просто заменить каждую шестнадцатиричную цифру ее двоичным эквивалентом. Например, 0xA4 соответствует 1010 0100. Аналогично, чтобы совершить обратное преобразование, следует каждый 4-разрядный блок заменить эквивалентной шестнадцатиричной цифрой. Возьмем двоичное число 100101, в шестнадцатиричной записи это 0x95.

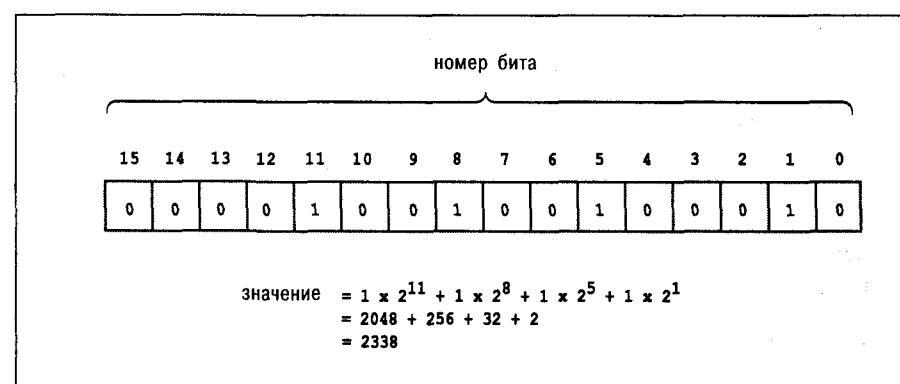


РИСУНОК A.1 Двухбайтовая целочисленная величина.

# Ключевые слова языка C++

**К**лючевыми словами называются идентификаторы, которые формируют словарный запас языка программирования. Их запрещено использовать в иных целях, например, в качестве имени переменной. В приве-

денном ниже списке перечислены ключевые слова языка C++, но некоторые из них не используются в современных реализациях языка. Слова, выделенные жирным шрифтом, являются также ключевыми и для С стандарта ANSI.

|                     |                 |                         |                    |                   |
|---------------------|-----------------|-------------------------|--------------------|-------------------|
| <b>asm</b>          | <b>auto</b>     | <b>bool</b>             | <b>break</b>       | <b>case</b>       |
| <b>catch</b>        | <b>char</b>     | <b>class</b>            | <b>const</b>       | <b>const_cast</b> |
| <b>continue</b>     | <b>default</b>  | <b>delete</b>           | <b>do</b>          | <b>double</b>     |
| <b>dynamic_cast</b> | <b>else</b>     | <b>enum</b>             | <b>explicit</b>    | <b>extern</b>     |
| <b>false</b>        | <b>float</b>    | <b>for</b>              | <b>friend</b>      | <b>goto</b>       |
| <b>if</b>           | <b>inline</b>   | <b>int</b>              | <b>long</b>        | <b>mutable</b>    |
| <b>namespace</b>    | <b>new</b>      | <b>operator</b>         | <b>private</b>     | <b>protected</b>  |
| <b>public</b>       | <b>register</b> | <b>reinterpret_cast</b> | <b>return</b>      | <b>short</b>      |
| <b>signed</b>       | <b>sizeof</b>   | <b>static</b>           | <b>static_cast</b> | <b>struct</b>     |
| <b>switch</b>       | <b>template</b> | <b>this</b>             | <b>throw</b>       | <b>true</b>       |
| <b>try</b>          | <b>typedef</b>  | <b>typeid</b>           | <b>typename</b>    | <b>union</b>      |
| <b>unsigned</b>     | <b>using</b>    | <b>virtual</b>          | <b>void</b>        | <b>volatile</b>   |
| <b>wchar_t</b>      | <b>while</b>    |                         |                    |                   |

# Таблица кодов ASCII

Символы в компьютере хранятся в виде числового кода. Коды ASCII (American Standard Code for Information Interchange — Американский стандартный код обмена информацией) наиболее часто используются в США. Язык C++ позволяет представлять многие одиночные символы, заключая их в одинарные кавычки, например, 'A' для символа A. Но можно представить одиночный символ, обращаясь к его восьмеричному или

шестнадцатиричному коду, указав перед ним обратный слэш; например, '\012' и '\0xa' обозначают символ перевода строки (LF). Подобные эскейп-последовательности могут являться частями строки, как, например, "Hello,\012my dear".

При использовании префикса в нижеприведенной таблице символ ^ обозначает код нажатой клавиши Ctrl.

| Десятичный | Восьмеричный | Шестнадцатиричный | Двоичный | Символ  | Название ASCII |
|------------|--------------|-------------------|----------|---------|----------------|
| 0          | 0            | 0                 | 00000000 | ^@      | NUL            |
| 1          | 01           | 0x1               | 00000001 | ^A      | SOH            |
| 2          | 02           | 0x2               | 00000010 | ^B      | STX            |
| 3          | 03           | 0x3               | 00000011 | ^C      | ETX            |
| 4          | 04           | 0x4               | 00000100 | ^D      | EOT            |
| 5          | 05           | 0x5               | 00000101 | ^E      | ENQ            |
| 6          | 06           | 0x6               | 00000110 | ^F      | ACK            |
| 7          | 07           | 0x7               | 00000111 | ^G      | BEL            |
| 8          | 010          | 0x8               | 00001000 | ^H      | BS             |
| 9          | 011          | 0x9               | 00001001 | ^I, tab | HT             |
| 10         | 012          | 0xa               | 00001010 | ^J      | LF             |
| 11         | 013          | 0xb               | 00001011 | ^K      | VT             |
| 12         | 014          | 0xc               | 00001100 | ^L      | FF             |
| 13         | 015          | 0xd               | 00001101 | ^M      | CR             |
| 14         | 016          | 0xe               | 00001110 | ^N      | SO             |
| 15         | 017          | 0xf               | 00001111 | ^O      | SI             |
| 16         | 020          | 0x10              | 00010000 | ^P      | DLE            |
| 17         | 021          | 0x11              | 00010001 | ^Q      | DC1            |
| 18         | 022          | 0x12              | 00010010 | ^R      | DC2            |
| 19         | 023          | 0x13              | 00010011 | ^S      | DC3            |
| 20         | 024          | 0x14              | 00010100 | ^T      | DC4            |

| <i>Десятичный</i> | <i>Восьмеричный</i> | <i>Шестнадцатиричный</i> | <i>Двоичный</i> | <i>Символ</i> | <i>Название ASCII</i> |
|-------------------|---------------------|--------------------------|-----------------|---------------|-----------------------|
| 21                | 025                 | 0x15                     | 00010101        | ^U            | NAK                   |
| 22                | 026                 | 0x16                     | 00010110        | ^V            | SYN                   |
| 23                | 027                 | 0x17                     | 00010111        | ^W            | ETB                   |
| 24                | 030                 | 0x18                     | 00011000        | ^X            | CAN                   |
| 25                | 031                 | 0x19                     | 00011001        | ^Y            | EM                    |
| 26                | 032                 | 0x1a                     | 00011010        | ^Z            | SUB                   |
| 27                | 033                 | 0x1b                     | 00011011        | ^[, esc       | ESC                   |
| 28                | 034                 | 0x1c                     | 00011100        | ^`            | FS                    |
| 29                | 035                 | 0x1d                     | 00011101        | ^]`           | GS                    |
| 30                | 036                 | 0x1e                     | 00011110        | ^``           | RS                    |
| 31                | 037                 | 0x1f                     | 00011111        | ^_            | US                    |
| 32                | 040                 | 0x20                     | 00100000        | space         | SP                    |
| 33                | 041                 | 0x21                     | 00100001        | !             |                       |
| 34                | 042                 | 0x22                     | 00100010        | "             |                       |
| 35                | 043                 | 0x23                     | 00100011        | #             |                       |
| 36                | 044                 | 0x24                     | 00100100        | \$            |                       |
| 37                | 045                 | 0x25                     | 00100101        | %             |                       |
| 38                | 046                 | 0x26                     | 00100110        | &             |                       |
| 39                | 047                 | 0x27                     | 00100111        |               |                       |
| 40                | 050                 | 0x28                     | 00101000        | (             |                       |
| 41                | 051                 | 0x29                     | 00101001        | )             |                       |
| 42                | 052                 | 0x2a                     | 00101010        | *             |                       |
| 43                | 053                 | 0x2b                     | 00101011        | +             |                       |
| 44                | 054                 | 0x2c                     | 00101100        | ,             |                       |
| 45                | 055                 | 0x2d                     | 00101101        | -             |                       |
| 46                | 056                 | 0x2e                     | 00101110        | .             |                       |
| 47                | 057                 | 0x2f                     | 00101111        | /             |                       |
| 48                | 060                 | 0x30                     | 00110000        | 0             |                       |
| 49                | 061                 | 0x31                     | 00110001        | 1             |                       |
| 50                | 062                 | 0x32                     | 00110010        | 2             |                       |
| 51                | 063                 | 0x33                     | 00110011        | 3             |                       |
| 52                | 064                 | 0x34                     | 00110100        | 4             |                       |
| 53                | 065                 | 0x35                     | 00110101        | 5             |                       |
| 54                | 066                 | 0x36                     | 00110110        | 6             |                       |
| 55                | 067                 | 0x37                     | 00110111        | 7             |                       |
| 56                | 070                 | 0x38                     | 00111000        | 8             |                       |
| 57                | 071                 | 0x39                     | 00111001        | 9             |                       |

| <i>Десятичный</i> | <i>Восьмеричный</i> | <i>Шестнадцатиричный</i> | <i>Двоичный</i> | <i>Символ</i> | <i>Название ASCII</i> |
|-------------------|---------------------|--------------------------|-----------------|---------------|-----------------------|
| 58                | 072                 | 0x3a                     | 00111010        | :             |                       |
| 59                | 073                 | 0x3b                     | 00111011        | ;             |                       |
| 60                | 074                 | 0x3c                     | 00111100        | <             |                       |
| 61                | 075                 | 0x3d                     | 00111101        | =             |                       |
| 62                | 076                 | 0x3e                     | 00111110        | >             |                       |
| 63                | 077                 | 0x3f                     | 00111111        | ?             |                       |
| 64                | 0100                | 0x40                     | 01000000        | @             |                       |
| 65                | 0101                | 0x41                     | 01000001        | A             |                       |
| 66                | 0102                | 0x42                     | 01000010        | B             |                       |
| 67                | 0103                | 0x43                     | 01000011        | C             |                       |
| 68                | 0104                | 0x44                     | 01000100        | D             |                       |
| 69                | 0105                | 0x45                     | 01000101        | E             |                       |
| 70                | 0106                | 0x46                     | 01000110        | F             |                       |
| 71                | 0107                | 0x47                     | 01000111        | G             |                       |
| 72                | 0110                | 0x48                     | 01001000        | H             |                       |
| 73                | 0111                | 0x49                     | 01001001        | I             |                       |
| 74                | 0112                | 0x4a                     | 01001010        | J             |                       |
| 75                | 0113                | 0x4b                     | 01001011        | K             |                       |
| 76                | 0114                | 0x4c                     | 01001100        | L             |                       |
| 77                | 0115                | 0x4d                     | 01001101        | M             |                       |
| 78                | 0116                | 0x4e                     | 01001110        | N             |                       |
| 79                | 0117                | 0x4f                     | 01001111        | O             |                       |
| 80                | 0120                | 0x50                     | 01010000        | P             |                       |
| 81                | 0121                | 0x51                     | 01010001        | Q             |                       |
| 82                | 0122                | 0x52                     | 01010010        | R             |                       |
| 83                | 0123                | 0x53                     | 01010011        | S             |                       |
| 84                | 0124                | 0x54                     | 01010100        | T             |                       |
| 85                | 0125                | 0x55                     | 01010101        | U             |                       |
| 86                | 0126                | 0x56                     | 01010110        | V             |                       |
| 87                | 0127                | 0x57                     | 01010111        | W             |                       |
| 88                | 0130                | 0x58                     | 01011000        | X             |                       |
| 89                | 0131                | 0x59                     | 01011001        | Y             |                       |
| 90                | 0132                | 0x5a                     | 01011010        | Z             |                       |
| 91                | 0133                | 0x5b                     | 01011011        | [             |                       |
| 92                | 0134                | 0x5c                     | 01011100        | \             |                       |
| 93                | 0135                | 0x5d                     | 01011101        | ]             |                       |
| 94                | 0136                | 0x5e                     | 01011110        | ^             |                       |

| <i>Десятичный</i> | <i>Восьмеричный</i> | <i>Шестнадцатиричный</i> | <i>Двоичный</i> | <i>Символ</i> | <i>Название ASCII</i> |
|-------------------|---------------------|--------------------------|-----------------|---------------|-----------------------|
| 95                | 0137                | 0x5f                     | 01011111        | -             |                       |
| 96                | 0140                | 0x60                     | 01100000        | '             |                       |
| 97                | 0141                | 0x61                     | 01100001        | a             |                       |
| 98                | 0142                | 0x62                     | 01100010        | b             |                       |
| 99                | 0143                | 0x63                     | 01100011        | c             |                       |
| 100               | 0144                | 0x64                     | 01100100        | d             |                       |
| 101               | 0145                | 0x65                     | 01100101        | e             |                       |
| 102               | 0146                | 0x66                     | 01100110        | f             |                       |
| 103               | 0147                | 0x67                     | 01100111        | g             |                       |
| 104               | 0150                | 0x68                     | 01101000        | h             |                       |
| 105               | 0151                | 0x69                     | 01101001        | i             |                       |
| 106               | 0152                | 0x6a                     | 01101010        | j             |                       |
| 107               | 0153                | 0x6b                     | 01101011        | k             |                       |
| 108               | 0154                | 0x6c                     | 01101100        | l             |                       |
| 109               | 0155                | 0x6d                     | 01101101        | m             |                       |
| 110               | 0156                | 0x6e                     | 01101110        | n             |                       |
| 111               | 0157                | 0x6f                     | 01101111        | o             |                       |
| 112               | 0160                | 0x70                     | 01110000        | p             |                       |
| 113               | 0161                | 0x71                     | 01110001        | q             |                       |
| 114               | 0162                | 0x72                     | 01110010        | r             |                       |
| 115               | 0163                | 0x73                     | 01110011        | s             |                       |
| 116               | 0164                | 0x74                     | 01110100        | t             |                       |
| 117               | 0165                | 0x75                     | 01110101        | u             |                       |
| 118               | 0166                | 0x76                     | 01110110        | v             |                       |
| 119               | 0167                | 0x77                     | 01110111        | w             |                       |
| 120               | 0170                | 0x78                     | 01111000        | x             |                       |
| 121               | 0171                | 0x79                     | 01111001        | y             |                       |
| 122               | 0172                | 0x7a                     | 01111010        | z             |                       |
| 123               | 0173                | 0x7b                     | 01111011        | {             |                       |
| 124               | 0174                | 0x7c                     | 01111100        |               |                       |
| 125               | 0175                | 0x7d                     | 01111101        | }             |                       |
| 126               | 0176                | 0x7e                     | 01111110        | ~             |                       |
| 127               | 0177                | 0x7f                     | 01111111        | del, rubout   |                       |

# Приоритет операций

Приоритетом операций определяется порядок, в котором операции применяются к какому-либо значению. Операции языка C++ распределены по 18 группам приоритетности, представленным в табл. D.1. Операции из группы 1 обладают наивысшим приоритетом. Если задано выполнение сразу двух операций, то операция с высшим приоритетом выполняется первой. Если две операции обладают равным приоритетом, то C++, чтобы определить ранее выполняемую операцию, использует правила ассоциативности. Операции, принадлежащие к одной и той же группе, обладают равным приоритетом и свойством ассоциативности, таким как “справа налево” (в таблице Л-П) или “слева направо” (в таблице П-Л). Свойство ассоциативности “слева направо” обозначает, что сначала выполняется операция, записанная слева, тогда как при ассоциативности “справа налево” первой выполняется операция, записанная справа.

Некоторые символы, например \* или &, используются для обозначения более чем одной операции. В таком случае одна форма записи является *унарной* (требующей наличия одного операнда), другая — *бинарной* (требующей наличия двух операндов), и компилятор в соответствии с употребляемой формой распознает, что имеет в виду программист. В таблице для случаев двойкого использования символов возле каждой группы операций указано, являются ли они унарными или бинарными.

Вот несколько примеров того, как влияют на программу приоритет и ассоциативность:

3 + 5 \* 6

Операция \* (умножение) обладает высшим приоритетом по сравнению с операцией + (сложение), поэтому она первой применяется к числу 5, благодаря чему выражение принимает значение 30+3, или 33.

120/6\*5

Обе операции — / (деление) и \* (умножение) — обладают одинаковым приоритетом, но они связаны свойством ассоциативности “слева направо”. Поэтому операция, записанная слева от общего операнда, выполняется первой, следовательно, выражение принимает значение 20\*5, или 100.

```
char * str = "Whoa";
char ch = *str++;
```

Обе унарные операции, \* и ++, обладают одинаковым приоритетом, но они связаны ассоциативностью “справа налево”. Именно поэтому операция инкремента действует на str, а не на \*str. Следовательно, при выполнении этой операции увеличивается значение указателя, после чего указатель уже указывает на новую переменную, но не изменяет значение этой переменной. Однако операция инкремента записана в постфиксной форме, что приводит к тому, что значение указателя наращивается после того, как исходное значение \*str присвоено переменной ch. Поэтому приведенное выражение присваивает символ W переменной ch, а после этого указатель перемещается на символ h.

Обратите внимание, что в таблице унарные и бинарные характеристики операций указаны в графе приоритета, это позволяет различать операции, обозначаемые одинаковыми символами, например, унарная операция адресации и бинарная операция поразрядного И.

Таблица D.1 Приоритеты и ассоциативность операций языка C++.

| Приоритет                       | Операция                                                                                                                                                                                                                                                                                 | Ассоциативность | Значение                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1                               | <code>::</code><br><i>(выражение)</i>                                                                                                                                                                                                                                                    |                 | Операция определения диапазона доступа<br>группировка                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 2                               | <code>()</code><br><code>()</code><br><code>[]</code><br><code>-&gt;</code><br><code>.</code><br><code>const_cast</code><br><code>dynamic_cast</code><br><code>reinterpret_cast</code><br><code>static_cast</code><br><code>typeid</code><br><code>++</code><br><code>--</code>          | Л-П             | Вызов функции<br>Конструирование значения, а именно <code>type (expr)</code><br>Список индексов массива<br>Операция косвенной принадлежности<br>Операция прямой принадлежности<br>Специализированное приведение типов<br>Специализированное приведение типов<br>Специализированное приведение типов<br>Специализированное приведение типов<br>Идентификация типа<br>Операция инкремента, постфиксный оператор<br>Операция декремента, постфиксный оператор                                                                      |
| 3 (все<br>операции<br>унарные)  | <code>!</code><br><code>-</code><br><code>+</code><br><code>-</code><br><code>++</code><br><code>--</code><br><code>&amp;</code><br><code>*</code><br><code>()</code><br><code>sizeof</code><br><code>new</code><br><code>new []</code><br><code>delete</code><br><code>delete []</code> | П-Л             | Логическое отрицание<br>Поразрядное отрицание<br>Унарный плюс (знак положительного числа)<br>Унарный минус (знак отрицательного числа)<br>Операция инкремента, префиксный оператор<br>Операция декремента, префиксный оператор<br>Адрес<br>Разыменование (косвенное значение)<br>Приведение типов, а именно <code>(type) expr</code><br>Размер, определяемый в байтах<br>Динамически распределяет память<br>Динамически размещает массив в памяти<br>Очистка динамической памяти<br>Очистка динамически распределенного массива |
| 4                               | <code>.*</code><br><code>-&gt;*</code>                                                                                                                                                                                                                                                   | Л-П             | Разыменовывает элемент<br>Косвенное разыменование элемента                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 5 (все<br>операции<br>бинарные) | <code>*</code><br><code>/</code><br><code>^</code>                                                                                                                                                                                                                                       | Л-П             | Умножение<br>Деление<br>Возведение в степень                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 6 (все<br>бинарные)             | <code>+</code><br><code>-</code>                                                                                                                                                                                                                                                         | Л-П             | Сложение<br>Вычитание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 7                               | <code>&lt;&lt;</code><br><code>&gt;&gt;</code>                                                                                                                                                                                                                                           | Л-П             | Левое смещение<br>Правое смещение                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| 8                               | <code>&lt;</code><br><code>&lt;=</code><br><code>&gt;=</code><br><code>&gt;</code>                                                                                                                                                                                                       | Л_П             | Меньше чем<br>Меньше или равно<br>Больше или равно<br>Больше чем                                                                                                                                                                                                                                                                                                                                                                                                                                                                |

| <i>Приоритет</i> | <i>Операция</i>         | <i>Ассоциативность</i> | <i>Значение</i>                                                |
|------------------|-------------------------|------------------------|----------------------------------------------------------------|
| 9                | <code>==</code>         | <b>Л-П</b>             | Равно                                                          |
|                  | <code>!=</code>         |                        | Не равно                                                       |
| 10 (бинарные)    | <code>&amp;</code>      | <b>Л-П</b>             | Поразрядное И                                                  |
| 11               | <code>^</code>          | <b>Л-П</b>             | Поразрядное исключающее ИЛИ                                    |
| 12               | <code> </code>          | <b>Л-П</b>             | Поразрядное ИЛИ                                                |
| 13               | <code>&amp;&amp;</code> | <b>Л-П</b>             | Логическое И                                                   |
| 14               | <code>  </code>         | <b>Л-П</b>             | Логическое ИЛИ                                                 |
| 15               | <code>=</code>          | <b>П-Л</b>             | Простое присваивание                                           |
|                  | <code>*=</code>         |                        | Умножение с последующим присваиванием результата               |
|                  | <code>/=</code>         |                        | Деление с последующим присваиванием результата                 |
|                  | <code>%=</code>         |                        | Деление с остатком с последующим присваиванием результата      |
|                  | <code>+=</code>         |                        | Сложение с последующим присваиванием результата                |
|                  | <code>-=</code>         |                        | Вычитание с последующим присваиванием результата               |
|                  | <code>&amp;=</code>     |                        | Выполняет поразрядное И, присваивается результат               |
|                  | <code>^=</code>         |                        | Выполняет поразрядное исключающее ИЛИ, присваивается результат |
|                  | <code> =</code>         |                        | Выполняет поразрядное ИЛИ, присваивается результат             |
|                  | <code>&lt;&lt;=</code>  |                        | Смещение влево и присваивание                                  |
|                  | <code>&gt;&gt;=</code>  |                        | Смещение вправо и присваивание                                 |
| 16               | <code>:?</code>         | <b>П-Л</b>             | Условная операция                                              |
| 17               | <code>throw</code>      | <b>Л-П</b>             | Генерация исключения                                           |
| 18               | ,                       | <b>Л-П</b>             | Объединяет два выражения в одно                                |

# Другие операции

Чтобы не перегружать книгу излишней информацией, в основном тексте не упоминаются два класса операций. Первая группа включает поразрядные операции, с помощью которых можно работать с отдельными битами величин, эти операции язык C++ унаследовал от С. Вторую группу составляют двухэлементные операции разыменования, они являются последним дополнением C++. Именно эти операции рассматриваются в данном приложении.

## Поразрядные операции

Поразрядные операции обрабатывают биты целочисленных величин. Например, операция левого смещения сдвигает биты влево, а поразрядное отрицание превращает каждую единицу в нуль, а каждый нуль — в единицу. Всего в C++ имеется шесть таких операций: `<<`, `>>`, `~`, `&`, `|` и `^`.

## Операции смещения

Операция смещения влево имеет следующий синтаксис:

```
value << shift
```

Здесь *value* — это целая величина, биты которой будут смещены, а *shift* — это количество битов, на которые нужно выполнить смещение. Например:

```
13 << 3
```

обозначает, что в числе 13 необходимо все биты сместить на три позиции влево. Освободившиеся позиции заполняются нулями, а биты, вышедшие за пределы числа, отбрасываются (рис. Е.1).

Поскольку каждая позиция бита представляет собой величину, вдвое превышающую величину, представленную битом справа (см. приложение А), то смещение на один бит эквивалентно умножению рассматриваемой величины на 2. Таким же образом, сдвиг на 2 позиции эквивалентен умножению на  $2^2$ , а сдвиг на  $n$  позиций соответствует умножению величины на  $2^n$ .

Операция смещения влево позволяет осуществлять возможности, присущие языкам машинных кодов (ассемблера). Различие состоит в том, что ассемблерный оператор смещения влево непосредственно изменяет содержимое регистра, тогда как в C++ эта операция создает новую величину, не изменяя существующую. В качестве примера рассмотрим следующую ситуацию:

```
int x = 20;
int y = x << 3;
```

Такая запись не изменяет значение *x*. Выражение *x*`<<3` с помощью величины *x* создает новую величину, точно так же как в результате операции *x+3* появляется новое значение, а значение *x* остается неизменным.

Чтобы с помощью операции смещения влево изменить значение переменной, необходимо кроме этой операции выполнить операцию присваивания. Можно выполнить обычную операцию присваивания или же операцию `<<=`, объединяющую действия по смещению и присваиванию.

```
x = x << 4; // обычное присваивание
y <<= 2; // смещение и присваивание
```

Операция смещения вправо (`>>`), оправдывает свое название, смещающая биты вправо. Ей свойственна следующая синтаксическая структура:

```
value >> shift
```



РИСУНОК Е.1 Операция смещения влево.

Здесь `value` должно быть целочисленной величиной, а `shift` — это количество битов, на которое осуществляется смещение. Например:

```
17 >> 2
```

обозначает, что все биты числа 17 необходимо сдвинуть на две позиции вправо. Для целых беззнаковых значений освобожденные позиции заменяются нулями, а разряды, вышедшие за пределы позиций числа, отбрасываются. Для целочисленных величин со знаком, вакантные позиции могут заполняться нулями или же значениями исходных крайних слева битов. Конкретный выбор зависит от реализации языка. На рис. Е.2 приведен вариант заполнения нулями.

Смещение на одну позицию вправо эквивалентно целочисленному делению пополам. Поэтому в общих чертах смещение на  $n$  позиций вправо соответствует целочисленному делению на  $2^n$ .

В C++ существует операция, позволяющая одновременно сдвигать биты числа вправо и заменить значение переменной полученной величиной:

```
int q = 43;
q >>= 2; //заменяет 43 на 43>>2 или,
 //что то же самое, на 10
```

В некоторых реализациях языка с помощью операций сдвига влево или вправо умножение или деление на соответствующую степень 2 выполняется быстрее, чем с помощью операций умножения или деления, но с повышением качества оптимизации кода компиляторами это различие исчезает.

## Логические поразрядные операции

Логические поразрядные операции аналогичны обычным логическим операциям, кроме того, что они действуют на значение бит за битом, а не сразу на все значение в целом. Рассмотрим обычную операцию отрицания (`!`) и поразрядную операцию отрицания (`~`). Операция `!` преобразует `true` (или ненулевую величину) в `false`, а `false` — в `true`. Операция `~` заменяет противоположным значением каждый бит числа (1 на 0 и 0 на 1). Например, представим значение 3 типа `unsigned char`

положным значением каждого отдельный бит числа (1 на 0 и 0 на 1). Например, представим значение 3 типа `unsigned char`

```
unsigned char x = 3;
```

Выражение `!x` равно 0. Чтобы получить значение `~x`, запишем его в двоичном виде: 00000011. Затем преобразуем каждое значение 0 в 1 и каждое значение 1 — в 0. В результате будет получено значение 11111100, что в десятичном виде соответствует 252. На рис. Е.3 приведен пример с 16-разрядным значением.

Поразрядная операция ИЛИ (`|`), комбинируя два исходных целочисленных значения, вычисляет новое значение. В этом новом значении каждый бит установлен равным 1, если один из соответствующих битов исходных значений или оба одновременно равны 1. Если же оба соответствующих разряда равны 0, то и результирующий разряд также устанавливается равным 0 (рис. Е.4).

В табл. Е.1 обобщаются правила комбинирования битов с помощью операции `|`.

Таблица Е.1 Результат операции  $b_1 | b_2$ .

| Значение бита | $b_1=0$ | $b_1=1$ |
|---------------|---------|---------|
| $b_2=0$       | 0       | 1       |
| $b_2=1$       | 1       | 1       |

Поразрядная операция исключающего ИЛИ (`^`) из двух целочисленных величин образует новое значение. Каждый бит нового значения устанавливается равным 1, если один из соответствующих битов исходных значений, но не одновременно оба равны 1. Если же оба бита

Десятичное число 13 хранится как двухразрядное значение типа `int`

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Результат операции `~13` — каждое значение 1 преобразовано в 0, каждое значение 0 заменено на 1.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

РИСУНОК Е.3. Операция поразрядного отрицания.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| b | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

|       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a   b | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

1, так как соответствующий бит значения `b` равен 1  
0, так как соответствующие биты значений `a` и `b` равны 0  
1, так как соответствующий бит значения `a` равен 1  
1, так как соответствующие биты значений `a` и `b` равны 1

РИСУНОК Е.4 Поразрядная операция ИЛИ.



РИСУНОК Е.2 Операция сдвига вправо.

равны 0 или 1, то результирующее значение бита полагается равным 0 (рис. Е.5).

В табл. Е.2 обобщаются правила комбинирования битов с помощью операции  $\wedge$ .

**Таблица Е.2 Результат операции  $b1 \wedge b2$ .**

| Значение бита | $b1=0$ | $b1=1$ |
|---------------|--------|--------|
| $b2=0$        | 0      | 1      |
| $b2=1$        | 1      | 0      |

Поразрядная операция И ( $\&$ ) из двух целочисленных величин образует новое значение. Каждый бит нового значения устанавливается равным 1 только в том случае, если оба соответствующих бита исходного значения равны 1. Если хотя бы один из них или оба одновременно равны 0, то результирующее значение бита полагается равным 0 (рис. Е.6).

В табл. Е.3 обобщаются правила комбинирования битов с помощью операции  $\&$ .

**Таблица Е.3 Результат операции  $b1 \& b2$ .**

| Значение бита | $b1=0$ | $b1=1$ |
|---------------|--------|--------|
| $b2=0$        | 0      | 0      |
| $b2=1$        | 0      | 1      |

## Несколько стандартных приемов по работе с битами

Часто при управлении процессами, протекающими в аппаратных устройствах, возникает необходимость включить или отключить некоторые конкретные биты или проверить их состояние. С помощью поразрядных операций можно осуществить подобные действия. Ниже приведен краткий обзор таких методов.

Например, пусть переменная **lottabits** хранит исходную величину, а переменная **bit** — значение конкретного бита. Биты пронумерованы в направлении справа налево, начиная с 0, поэтому значение, соответствующее

позиции бита  $n$ , равно  $2^n$ . Это значит, что имеется целочисленная величина, в которой только бит под номером 3 имеет значение 1, равное  $2^3$ , или 8. Другими словами, каждый отдельный **бит** соответствует степени числа 2 (более подробно о двухразрядных числах рассказано в приложении А). Поэтому ниже термин **бит** будет обозначать степень числа 2; это соответствует ситуации, когда конкретный бит равен 1, а все остальные — 0.

### Установка бита

Две приведенные ниже операции устанавливают бит переменной **lottabits**, соответствующий биту, который представлен переменной **bit**:

```
lottabits = lottabits | bit;
lottabits |= bit;
```

Каждая из операций устанавливает соответствующий бит равным 1, независимо от его исходного значения. Так происходит благодаря тому, что операция ИЛИ, сопоставляя 1 с 0 или с 1, в любом случае отображает 1. А другие биты переменной **lottabits** остаются неизменными, потому что сопоставление 0 с 0 с помощью операции ИЛИ порождает 0, а сопоставление 0 с 1 порождает 1.

### Переключение бита

Две приведенные ниже операции переключают бит переменной **lottabits**, соответствующий ненулевому биту переменной **bit**:

```
lottabits = lottabits ^ bit;
lottabits ^= bit;
```

Сопоставление 1 с 0 с помощью операции исключающего ИЛИ порождает 1, при этом устанавливается неустановленный бит, а сопоставление 0 с 1 с помощью исключающего ИЛИ порождает 0, при этом отменяется установленный бит. Оставшиеся биты переменной **lottabits** остаются неизменными, потому что сопоставление 0 с 0 с помощью операции ИЛИ порождает 0, а сопоставление 0 с 1 порождает 1.

|              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|--------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a            | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| b            | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| $a \wedge b$ | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |   |   |
|              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

1, так как соответствующий бит значения **b** равен 1  
0, так как соответствующие биты значений **a** и **b** равны 0  
1, так как соответствующий бит значения **a** равен 1  
0, так как соответствующие биты значений **a** и **b** равны 1

**РИСУНОК Е.5 Поразрядная операция исключающего ИЛИ.**

|              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|--------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a            | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| b            | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| $a \wedge b$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

0, так как только один из соответствующих битов равен 1  
0, так как соответствующие биты значений **a** и **b** равны 0  
1, так как соответствующие биты значений **a** и **b** равны 1

**РИСУНОК Е.6 Поразрядная операция И.**

## Отмена установленного бита

Приведенная ниже операция отменяет установку бита переменной **lottabits**, соответствующего биту, представленному переменной **bit**

```
lottabits = lottabits & ~bit;
```

Такой оператор отменяет установку бита независимо от его предыдущего состояния. Сначала операция **~bit** создает целочисленную величину, все биты которой равны 1, кроме того бита, который исходно имел значение 1, этот бит теперь равен 0. Сравнение 0 с любым другим битом с помощью операции И приводит к отмене установки этого бита. Все же оставшиеся разряды переменной **lottabits** остаются неизменными, потому что сопоставление 1 с любым другим битом с помощью операции ИЛИ не изменяет исходное состояние бита.

Сокращенная запись оператора, выполняющего аналогичные описанным выше действия:

```
lottabits &= ~bit;
```

## Проверка значения бита

Предположим, что необходимо узнать состояние бита переменной **lottabits**, соответствующего величине **bit**. Следующая проверка не обязательно приведет к нужным результатам:

```
if (lottabits == bits) //недостаточная проверка
```

Даже если соответствующий бит **lottabits** равен 1, это не исключает равенство 1 оставшихся битов переменной. Приведенное выше равенство справедливо *только в том случае, если* соответствующий бит равен 1. Исправить положение можно, выполнив операцию И с величинами **lottabits** и **bit**. После этого значения всех битов будут равны нулю, так как сопоставление 0 с помощью логического И с любым другим битом порождает только 0. А бит, соответствующий значению **bit**, останется неизмененным, так как 1 в комбинации с любым значением порождает 1. Таким образом, корректная проверка будет следующей:

```
if (lottabits&bit == bits) //проверка бита
```

## Операции разыменования элементов

Прежде чем обсуждать операции разыменования элементов, необходимо сделать небольшое вступление. Язык C++ позволяет создавать указатели на элементы класса, это довольно сложный процесс. Чтобы увидеть, что же происходит, обратимся к такому примеру класса, который порождает некоторые трудности:

```
class example
{
private:
 int feet;
 int inches;
public:
 example();
 example(int ft);
 ~example();
 void show_in(); //отображает
 //элемент inches
 example operator+(example &ex);
};
```

Теперь предположим, что нужно определить указатель на элемент **inches** этого класса. Простейший вариант не срабатывает:

```
int * pi = &inches; // в C++ некорректно
```

Такой вариант неудачен, потому что **inches** не принадлежит к типу **int**. Поскольку переменная **inches** объявлена в классе, ее действие распространяется только в пределах класса. Поэтому тип переменной **inches** должен указывать класс, к которому принадлежит этот элемент. Чтобы сделать объявление корректным, необходимо с помощью операции определения области действия определить класс для указателей и для элементов:

```
int example::* pi = &example::inches;
// корректно в C++
```

В этом объявлении фраза **int example::\*** определяет тип "указатель на элемент типа **int** класса **example**." Выражение **&example::inches** обозначает "адрес элемента **inches** класса **example**".

Такое объявление можно использовать в функциях над элементами или в дружественных функциях. Указатель **pi** действует как элемент класса, и поэтому его необходимо активизировать с помощью объекта класса. Вот здесь и появляются операции разыменования элементов. Для начала предположим, что **ex** является примером объекта, объявленного в функции-элементе. Чтобы обеспечить доступ к элементу **inches** из **ex**; можно воспользоваться стандартной записью **ex.inches**. Можно также выполнить операцию **.\*** в комбинации с указателем **pi**:

```
cout << ex.inches; //отображается элемент
 //inches
cout << ex.*pi; //аналогично первой строке
```

Подобно тому как операция **.** (точка) обеспечивает доступ к элементу с помощью имени этого элемента, операция **element.\*** осуществляет доступ к элементу с помощью указателя на этот элемент.

Точно так же предположим, что **px** — это указатель на объект **example**. Тогда с помощью операции **->** можно обратиться к элементу **inches** по имени или с помощью операции разыменования **->\*** обратиться к **inches** через указатель на элемент:

```

px = &ex;
// px - указатель на объект example
cout << px->inches;
// отображает элемент inches
cout << px->*pi; //аналогично первой строке

```

Обратите внимание, что `px` указывает на исходный объект, тогда как `pi` указывает на элемент класса.

Чтобы разобраться в том, как эти новые операции в действительности выполняются, применим их несколько необычным образом, введя функцию `operator+()`. Эта функция добавляет два объекта. Один из них — это аргумент функции. Поскольку он является объектом, а не указателем, то, чтобы обратиться к элементу `inches`, можно применить операцию `.*`. Оставшийся объект — это вызывающий объект, представленный указателем `this`. Поэтому его можно применять в комбинации с операцией `->`, как показано в следующем фрагменте программы:

```

example example::operator+(example &ex)
{
 example sum;
 int example::*pi = &example::inches;
 // указывает на элемент
 // inches класса example
 sum.inches = ex.*pi + this->*pi;
 sum.feet = 12 * sum.inches;
 return sum;
}

```

В этом примере `ex.*pi` является элементом `inches` класса `ex`, а `this->*pi` представляет элемент `inches` объекта, на который указывает `this`. Обратите внимание, что указатель `*pi` использован как имя элемента.

В листинге Е.1 приведены оставшиеся определения метода, а также функция `main()`, использующая класс.

#### Листинг Е.1 Программа memb\_pt.cpp.

```

// memb_pt.cpp - разыменованные указатели
// на элементы класса
#include <iostream>
using namespace std;

class example
{
private:
 int feet;
 int inches;
public:
 example();
 example(int ft);
 ~example();
 void show_in();
 example operator+(example &ex);
};

example::example()
{
 feet = 0;
 inches = 0;
}

```

```

example::example(int ft)
{
 feet = ft;
 inches = 12 * feet;
}

example::~example()
{
}

void example::show_in()
{
 cout << inches << " inches\n";
}

example example::operator+(example &ex)
{
 example sum;
 int example::*pi = &example::inches;
 // указывает на элемент inches класса example
 sum.inches = ex.*pi + this->*pi;
 sum.feet = 12 * sum.inches;
 return sum;
}

int main()
{
 example car(15);
 example van(20);
 example garage;

 garage = car + van;
 car.show_in();
 van.show_in();
 garage.show_in();

 return 0;
}

```

Результаты выполнения программы:

```

180 inches
240 inches
420 inches

```

# Класс шаблона STRING

Класс `string` основан на определении шаблона:

```
template<class charT, class traits =
 char_traits<charT>, class Allocator =
 allocator<charT> >
class basic_string {...} ;
```

Здесь `charT` представляет тип, хранящийся в строке. Параметр `traits` является классом, определяющим свойства, которыми обязательно должен обладать тип, представляемый как строка. Например, он должен включать метод `length()`, возвращающий длину строки, заданной массивом типа `charT`. Конец такого массива обозначается величиной `charT(0)`, являющейся обобщением нулевого символа. (Выражение `charT(0)` приводит тип значения `0` к типу `charT`). Поскольку в этом примере используется тип `char`, можно было бы записать просто `0`, или, вообще, на месте нуля мог бы стоять любой объект, созданный конструктором `charT()`. Рассматриваемый класс содержит также методы для сравнения величин и многое другое. Параметр `Allocator` является классом, оперирующим с распределением памяти для строковых величин. Стандартный шаблон `allocator<charT>` использует методы `new` и `delete` обычным образом.

Существует две зарезервированные специализации шаблона:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Эти специализации, в свою очередь, используют следующие специализации:

```
char_traits<char>
allocator<char>
char_traits<wchar_t>
allocator<wchar_t>
```

Можно создать класс `string` для некоторого типа, отличного от `char` или `wchar_t`, определяя класс `traits` и используя шаблон `basic_string`.

## Тринадцать типов и констант

Шаблон `basic_string` определяет несколько типов, которые затем используются при задании методов:

```
typedef traits
 traits_type;
typedef typename traits::char_type
 value_type;
typedef Allocator
 allocator_type;
typedef typename Allocator::size_type
 size_type;
typedef typename Allocator::difference_type
 difference_type;
typedef typename Allocator::reference
 reference;
typedef typename Allocator::const_reference
 const_reference;
typedef typename Allocator::pointer
 pointer;
typedef typename Allocator::const_pointer
 const_pointer;
```

Обратите внимание, что `traits` — это параметр шаблона, который позже будет соответствовать некоторому специальному типу, например, типу `char_traits<char>`; тогда `traits_type` станет типом `typedef` конкретно для этого типа. Запись

```
typedef typename traits::char_type
 value_type;
```

сообщает, что `char_type` является именем типа, определенного в классе, представленном `traits`. Ключевое слово `typename` сообщает компилятору о том, что выражение `traits::char_type` — это имя типа. Например, строковой специализации `value_type` будет соответствовать `char`.

Метод `size_type` используется таким же образом, как `size_of`, за исключением того, что он возвращает размер строки, используя термины сохраненного типа. При строковой специализации типом возвращаемого ответа будет `char`, и поэтому область действия `size_type` полно-

стью совпадает с областью действия `size_of`. Этот тип является типом без знака.

Метод `difference_type` используют для оценки расстояний между двумя элементами строки, опять же в единицах, соответствующих величине одного элемента. Обычно им может быть вариант исходного для `size_type` типа со знаком.

Для конкретизации `char` тип `pointer` будет соответствовать типу `char *`, а тип `reference` будет принадлежать типу `char &`. Однако, если необходимо выполнить специализацию для самостоятельно разработанного типа, эти типы (`pointer` и `reference`) могут соотноситься с классами, обладающими свойствами, присущими исходным классам указателей и ссылок.

Чтобы алгоритм STL можно было использовать для строковых объектов, в шаблоне определено несколько типов для итераторов:

```
typedef (models random access iterator)
iterator;
typedef (models random access iterator)
const_iterator;
typedef std::reverse_iterator<iterator>
reverse_iterator;
typedef std::reverse_iterator<const_iterator>
const_reverse_iterator;
```

Шаблон определяет статические константы:

```
static const size_type pros = -1;
```

Поскольку `size_type` — это тип без знака, присваивание величины `-1` в действительности соответствует присваиванию переменной `pros` — наибольшей из возможных величин без знака. Это значение соответствует величине, на `1` превышающей наибольший индекс массива.

## Информация о данных, конструкторах и другие сведения

Рассказать о конструкторах можно, объяснив, какое влияние они оказывают. Поскольку индивидуальные особенности классов зависят от реализации, эти эффекты следует распознавать по информации, отображаемой в общедоступной части интерфейса. Возвращаемые величины для методов, перечисленных в табл. F.1, можно использовать для описания эффектов, порождаемых конструкторами и другими методами. Обратите внимание, что большая часть терминологии позаимствована из библиотеки STL.

Важно четко осознавать различие между методами `begin()`, `rend()`, `data()` и `c_str()`. Все они ссылаются на первый символ строки, но с помощью различных функций. Методы `begin()` и `rend()` возвращают итераторы, являющиеся обобщениями указателей, как уже извест-

но из главы 15, в которой описывались библиотеки STL. Точнее, метод `begin()` возвращает прямой итератор, а метод `rend()` — копию обратного итератора. Оба они ссылаются на саму строку, управляемую строковым объектом. (Поскольку строковый класс динамически распределяет память и самому объекту не нужно хранить реальное содержимое строки, термин *управлять* используют, чтобы выразить взаимоотношение между объектом и строкой.) Можно воспользоваться методами, которые возвращают итераторы с помощью алгоритмов библиотеки STL, использующих механизмы итераторов. Например, чтобы преобразовать элементы строки в символы верхнего регистра, можно применить функцию `transform()`:

```
string word;
cin >> word;
transform(word.begin(), word.end(),
toupper);
```

С другой стороны, методы `data()` и `c_str()`, действительно возвращают обычные указатели. Более того, возвращаемые указатели указывают на первый элемент *массива*, хранящего строковый символ. Этот массив может иногда являться копией исходной строки, управляемой строковым объектом. (В качестве внутреннего представления строковый объект может использовать массив, хотя это не обязательно.) Поскольку возвращаемые указатели, вполне возможно, могут указывать на исходные данные, они имеют тип `const` и с их помощью невозможно изменить данные. Ко всему сказанному необходимо добавить, что после изменения объекта указатели не обязательно сохраняются; это отражает тот факт, что они указывают на исходные данные. Различие между `data()` и `c_str()` состоит в том, что указанный массив `c_str()` завершается нулевым символом (или его эквивалентом), тогда как указатель `data()` только дает гарантию существования строковых символов. Поэтому метод `c_str()` можно использовать, например, как аргумент функции, являющейся С-строкой:

```
string file("tofu.man");
ofstream outFile(file.c_str());
```

Таким же образом `data()` и `size()` можно использовать в функции, получающей в качестве аргумента указатель на элемент массива и величину, представляющую число элементов, которые необходимо обработать:

```
string vampire("Do not stake me, oh my
darling!");
int vlad = byte_check(vampire.data(),
vampire.size());
```

В конкретной реализации могла бы существовать договоренность использовать для представления строки объекта `string` динамически размещенную С-строку и

**Таблица F.1 Некоторые методы данных типа string.**

| <i>Метод</i>                 | <i>Возвращает</i>                                                                                                                                                                                                                                                                                                                                                                                           |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>begin()</code>         | Итератор для первого символа строки (существует также вариант <code>const</code> , который возвращает итератор <code>const</code> ).                                                                                                                                                                                                                                                                        |
| <code>end()</code>           | Итератор, являющийся меткой "вне файла" (существует вариант <code>const</code> ).                                                                                                                                                                                                                                                                                                                           |
| <code>rbegin()</code>        | Обратный итератор, являющийся меткой "вне файла" (доступен вариант <code>const</code> ).                                                                                                                                                                                                                                                                                                                    |
| <code>rend()</code>          | Обратный итератор, ссылающийся на первый символ (существует вариант <code>const</code> ).                                                                                                                                                                                                                                                                                                                   |
| <code>size()</code>          | Число элементов в строке, эквивалентное расстоянию от <code>begin()</code> до <code>end()</code> .                                                                                                                                                                                                                                                                                                          |
| <code>length()</code>        | То же самое, что и <code>size()</code> .                                                                                                                                                                                                                                                                                                                                                                    |
| <code>capacity()</code>      | Число элементов, расположенных в строке.                                                                                                                                                                                                                                                                                                                                                                    |
| <code>max_size()</code>      | Максимально допустимая длина строки.                                                                                                                                                                                                                                                                                                                                                                        |
| <code>data()</code>          | Указатель типа <code>const charT*</code> на первый элемент массива, первые элементы <code>size()</code> которого равны соответствующим элементам в строке, управляемой указателем <code>*this</code> . Предполагается, что после изменения строкового объекта указатель недействителен.                                                                                                                     |
| <code>c_str()</code>         | Указатель типа <code>const charT*</code> на первый элемент массива, первые элементы <code>size()</code> которого равны соответствующим элементам в строке, управляемой указателем <code>*this</code> , а следующий элемент является символом <code>charT(0)</code> (меткой окончания строки) для типа <code>charT</code> . Предполагается, что после изменения строкового объекта указатель недействителен. |
| <code>get_allocator()</code> | Копия объекта распределителя, используемого для распределения памяти под строковые объекты.                                                                                                                                                                                                                                                                                                                 |

прямой итератор как указатель `char *`. В этом случае в данной реализации методы `begin()`, `data()` и `c_str()` возвращали бы один и тот же указатель. Но, согласно установленным правилам (а не следуя соображениям простоты), они возвращают ссылки на три различных объекта данных.

В табл. F.2 перечислены шесть конструкторов и один деструктор для класса шаблонов `basic_string`. Обратите внимание, что все шесть конструкторов содержат аргумент в следующей форме:

```
const Allocator& a = Allocator()
```

Вспомним, что термин `Allocator` является именем параметра шаблона для класса генераторов, управляющих памятью. Термин `Allocator()` обозначает стандартный конструктор для текущего класса. Поэтому конструкторы, заданные по умолчанию, используют стандартный вариант объекта, распределяющего память, но у программиста есть возможность использовать другие варианты этих объектов. Теперь исследуем индивидуальные особенности конструкторов.

**Таблица F.2 Строковые конструкторы.**

#### Прототипы конструкторов и деструкторов

```
explicit basic_string(const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(const basic_string& str, size_type pos = 0, size_type n =npos, const
 Allocator& a = Allocator());
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());
~basic_string();
```

## Стандартные конструкторы

Стандартным конструктором является:

```
explicit basic_string(const Allocator&
 a = Allocator());
```

часто можно договориться о стандартном аргументе и для создания пустых строк использовать конструктор:

```
string bean;
wstring theory;
```

Следующие соотношения устанавливаются после вызова конструкторов:

- Метод `data()` возвращает не пустой указатель, к которому можно прибавить `0`.
- Метод `size()` возвращает 0.
- Величина, возвращаемая методом `capacity()`, не указана.

Предположим, что величина, возвращаемая методом `data()`, присвоена указателю `str`. Тогда первое условие обозначает, что `str + 0` существует, это, в свою очередь, свидетельствует о существовании `str[0]`.

## Конструкторы, использующие массив

Следующий конструктор позволяет инициализировать строковый объект значениями С-строки; говоря более общим языком, с его помощью можно инициализировать специализацию `charT` из массива величин типа `charT`:

```
basic_string(const charT* s, const
 Allocator& a = Allocator());
```

Чтобы определить, сколько символов необходимо копировать, конструктор применяет метод `traits::length()` к массиву, указанному указателем `s`. (Указатель `s` будет пустым указателем.) Например, оператор

```
string toast("Here's looking at you, kid.");
```

инициализирует объект `toast`, используя указанную символьную строку. Метод `traits::length()` для типа `char` применит нулевой символ, чтобы определить количество символов, подлежащих копированию.

После вызова конструкторов устанавливаются следующие соотношения:

- Метод `data()` возвращает указатель на первый элемент копии массива `s`.
- Метод `size()` возвращает величину, равную `traits::length()`.
- Метод `capacity()` возвращает величину, не меньшую, чем `size()`.

## Конструкторы, использующие часть массива

Следующий конструктор позволяет инициализировать строковый объект с помощью значений части С-строки; говоря более общим языком, он позволяет инициализировать специализацию `charT` из части массива величин типа `charT`:

```
basic_string(const charT* s, size_type n,
 const Allocator& a = Allocator());
```

Этот конструктор копирует символы общим количеством `n` из массива, указанного указателем `s`, в созданный объект. Обратите внимание, что процесс копирования не останавливается, если в `s` содержится меньше `n` символов. Если `n` превышает `s`, то конструктор интерпретирует содержимое ячеек памяти, следующих за строкой, как величины типа `charT`.

Конструктор требует, чтобы `s` был непустым указателем и чтобы `n < npos`. (Не забывайте, что `npos` — это

константа из статического класса, равная максимально возможному числу элементов в строке.) Если `n` равно `npos`, то конструктор генерирует исключительную ситуацию `out_of_range`. (Поскольку `n` относится к типу `size_type`, величина `n` не может превышать величину `npos`.) Другими словами, после вызова конструкторов действуют следующие отношения:

- Метод `data()` возвращает указатель на первый элемент копии массива `s`.
- Метод `size()` возвращает `n`.
- Метод `capacity()` возвращает величину, по меньшей мере равную `size()`.

## Конструктор копирования

Конструктор копирования обладает несколькими аргументами, принимающими стандартные значения:

```
basic_string(const basic_string& str,
 size_type pos = 0, size_type n = npos,
 const Allocator& a = Allocator());
```

При вызове только с аргументом `basic_string` инициализируется новый объект, имеющий строковый аргумент:

```
string mel("I'm ok!");
string ida(mel);
```

Здесь `ida` извлекает копию строки, управляемой объектом `mel`.

Второй необязательный аргумент `pos` указывает на место в исходной строке, с которого необходимо начать копирование:

```
string att("Telephone home.");
string et(att, 4);
```

Нумерация позиций начинается с 0, поэтому в позиции 4 находится символ `r`. Таким образом, строка `et` инициализирована значением "phone home".

Третий необязательный аргумент `n` обозначает максимальное число символов, подлежащих копированию. Поэтому фрагмент

```
string att("Telephone home.");
string pt(att, 4, 5);
```

инициализирует `pt` значением строки "phone". Тем не менее, этот конструктор не выходит за пределы исходной строки; например, фрагмент

```
string pt(att, 4, 200)
```

завершит работу после копирования точки. Поэтому конструктор в действительности копирует количество символов, не превышающее меньшую из величин `n` и `str.size() - pos`.

Для этого конструктора необходимо, чтобы `pos <= str.size()`, что является признаком ситуации, когда позиция, в которую осуществляется копирование, находится внутри исходной строки; если же это не так, то конструктор генерирует исключительную ситуацию `out_of_range`. Другими словами, если `copy_len` соответствует меньшей из величин `n` и `str.size() - pos`, то после вызова конструктора устанавливаются следующие отношения:

- Метод `data()` возвращает указатель на копию элементов `copy_len`, скопированных из строки `str`, начиная с позиции `pos` в `str`.
- Метод `size()` возвращает `copy_len`.
- Метод `capacity()` возвращает величину, не превышающую `size()`.

## Конструктор, использующий `n` копий символа

Конструктор, о котором идет речь, создает строковый объект, состоящий из последовательности `n` символов, каждый из которых равен `c`:

```
basic_string(size_type n, charT c,
 const Allocator& a = Allocator());
```

Для этого конструктора необходимо, чтобы выполнялось условие `n < npos`. Если `n` равно `npos`, конструктор генерирует исключительную ситуацию `out_of_range`. Другими словами, после вызова конструктора устанавливаются следующие отношения:

- Метод `data()` возвращает указатель на первый элемент `n`-элементной строки, состоящей из значений `c`.
- Метод `size()` возвращает `n`.
- Метод `capacity()` возвращает величину, которая не меньше, чем `size()`.

## Конструкторы, использующие диапазон значений

Последний конструктор использует диапазон значений, задаваемый итератором, как в языке STL:

Таблица F.3 Некоторые методы, работающие с ячейками памяти.

| Метод                                            | Эффект                                                                                                                                                                                                                                                                                          |
|--------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void resize(size_type n)</code>            | Генерирует исключительную ситуацию, если <code>n &gt; npos</code> . В противоположном случае изменяет размер строки на <code>n</code> позиций, "обрезая" конец строки, если <code>n &lt; size()</code> , и продлевая строку символами <code>charT(0)</code> , если <code>n &gt; size()</code> . |
| <code>void resize(size_type n, charT c)</code>   | Генерирует исключительную ситуацию, если <code>n &gt; npos</code> . В противоположном случае, изменяет размер строки на <code>n</code> , "обрезая" конец строки, если <code>n &lt; size()</code> , и продлевая строку символами <code>c</code> , если <code>n &gt; size()</code> .              |
| <code>void reserve(size_type res_arg = 0)</code> | Приравнивает <code>capacity()</code> к величине, большей или равной <code>res_arg</code> . Поскольку при этом приходится переместить строку, метод удаляет предыдущие ссылки, итераторы и указатели в строке.                                                                                   |
| <code>void clear()</code>                        | Удаляет все символы из строки.                                                                                                                                                                                                                                                                  |
| <code>bool empty() const</code>                  | Возвращает <code>true</code> , если <code>size() == 0</code> .                                                                                                                                                                                                                                  |

```
template<class InputIterator>
basic_string(InputIterator begin,
 InputIterator end,
 const Allocator& a = Allocator());
```

Итератор `begin` указывает на элемент в исходном файле, с которого начинается копирование, а `end` указывает на элемент, следующий за последним элементом, подлежащим копированию.

Эту форму можно использовать совместно с массивом, строкой или контейнером STL:

```
char cole[40] = "Old King Cole was a merry
 old soul.";
string title(cole + 4, cole + 8);
vector<char> input;
char ch;
while (cin.get(ch) && ch != '\n')
 input.push_back(ch);
string str_input(input.begin(), input.end());
```

При первом использовании `InputIterator` причисляется к типу `const char *`. При вторичном использовании `InputIterator` причисляется к типу `vector<char>::iterator`.

После вызова конструктора устанавливаются следующие соотношения:

- Метод `data()` возвращает указатель на первый элемент строки, сформированной копированием элементов из диапазона `[begin, end)`.
- Метод `size()` возвращает расстояние между `begin` и `end`. (Расстояние измеряется в единицах, соответствующих размеру данных типа, полученных при разыменовании итератора.)
- Метод `capacity()` возвращает величину, которая не меньше, чем `size()`.

## Ячейки памяти

Ячейками памяти распоряжаются несколько методов. Эти методы могут реализовать очистку содержимого памяти, задание нового размера строки, настройку размера строки. В табл. F.3 перечислены некоторые методы, работающие с ячейками памяти.

## Доступ к строке

Существует четыре метода для доступа к отдельным символам, два из которых используют операцию `[]`, а два используют метод `at()`:

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
reference at(size_type n);
const_reference at(size_type n) const;
```

Первый метод `operator[]()` позволяет осуществить доступ к отдельному элементу строки, используя обозначения массивов; его используют для поиска или изменения величин. Второй метод `operator[]()` можно использовать совместно с объектами `const` и только для поиска значений:

```
string word("tack");
cout << word[0]; // отображает t
word[3] = 't'; // записывает k вместо t
const ward("garlic");
cout << ward[2]; // отображает r
```

Метод `at()` обеспечивает такие же возможности доступа, но номер используется как обозначение аргумента функции:

```
string word("tack");
cout << word.at(0); // отображает t
```

Различие (не учитывая особенностей синтаксической формы) состоит в том, что метод `at()` обеспечивает проверку пределов и отбрасывает исключительные ситуации `out_of_range`, если `pos >= size()`. Обратите внимание, что `pos` относится к типу `size_type`, типу без знака, поэтому величине `pos` запрещено присваивать отрицательные значения. Метод `operator[]()` не проверяет пределы, и поэтому его поведение не определено, если `pos >= size()`, за исключением тех случаев, когда вариант `const` возвращает эквивалент нулевого символа, если `pos == size()`.

Поэтому у программиста есть выбор между безопасностью (использование `at()` и проверка на исключительные ситуации) или быстротой выполнения (использование обозначений, характерных для массивов).

Существует также функция, возвращающая новую строку, которая является подчиненной строкой для исходного объекта:

```
basic_string substr(size_type pos = 0,
size_type n = npos) const;
```

Она возвращает строку, являющуюся копией исходной строки, начиная с позиции `pos` и продолжая ее на `n` символов или до конца строки, в зависимости от того, как далеко находится конец строки. Например, в следующем фрагменте `pet` инициализируется значением "donkey":

```
string message("Maybe the donkey will learn
to sing.");
string pet(message.substr(10, 6));
```

## Базовые методы присваивания

Существует три совмещенных метода присваивания:

```
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
```

Первый метод присваивает одну строку другой, второй — присваивает С-строку строковому объекту, и третий присваивает отдельный символ строковому объекту. Благодаря этому возможны следующие операции:

```
string name("George Wash");
string pres, veep, source;
pres = name;
veep = "Road Runner";
source = 'X';
```

## Поиск строковых величин

Класс `string` предусматривает шесть функций поиска, каждая из которых обладает четырьмя прототипами. Приведем их краткое описание.

### Семейство методов `find()`

Приводим прототипы методов `find()`:

```
size_type find (const basic_string& str,
size_type pos = 0) const;
size_type find (const charT* s, size_type
pos = 0) const;
size_type find (const charT* s, size_type
pos, size_type n) const;
size_type find (charT c, size_type pos = 0)
const;
```

Первый элемент возвращает позицию, соответствующую началу подчиненной строки `str`, когда последние впервые появилась в вызванном объекте; тогда как поиск начинается с позиции `pos`. Если подчиненная строка не найдена, метод возвращает `npos`.

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find(shorter);
// присваивает loc1 значение 1
size_type loc2 = longer.find(shorter,
loc1 + 1); // присваивает loc2 значение 16
```

Поскольку вторичный поиск запускается с позиции 2 (символ `a` в `That`), первый раз метод находит `hat` только в конце строки. При проверке значения на корректность используется величина `string::npos`:

```
if (loc1 == string::npos)
cout << "Not found\n";
```

Второй метод выполняет то же самое, но использует в качестве подчиненной строки массив символов вместо строкового объекта:

```
size_type loc3 = longer.find("is");
// присваивает loc3 значение 5
```

Третий метод выполняет то же самое, что и второй, но использует только *n* первых символов строки *s*. Результат такой же, как при использовании конструктора **basic\_string(const charT\* s, size\_type n)** и результирующего объекта в качестве аргумента строки первой формы метода **find()**. Например, в следующем фрагменте осуществляется поиск подчиненной строки "fun":

```
size_type loc4 = longer.find("funds", 3);
//присваивает loc4 значение 10
```

Четвертый метод выполняет то же самое, что и первый, но использует отдельный символ вместо строкового объекта в качестве подчиненной строки:

```
size_type loc5 = longer.find('a');
//присваивает loc5 значение 2
```

## Семейство методов **rfind()**

Методы **rfind()** обладают следующими прототипами:

```
size_type rfind(const basic_string& str,
 size_type pos = npos) const;
size_type rfind(const charT* s,
 size_type pos = npos) const;
size_type rfind(const charT* s,
 size_type pos, size_type n) const;
size_type rfind(charT c,
 size_type pos = npos) const;
```

Эти методы работают подобно аналогичным методам **find()**, однако они отыскивают последний случай появления строки или символа, начинающихся с позиции *pos* или до нее. Если подчиненная строка не найдена, метод возвращает *npos*.

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.rfind(shorter);
// присваивает loc1 значение 16
size_type loc2 = longer.rfind(shorter,
 loc1 - 1); //присваивает loc2 значение 1
```

## Семейство методов **find\_first\_of()**

Методы **find\_first\_of()** обладают следующими прототипами:

```
size_type find_first_of(const basic_string&
 str, size_type pos = 0) const;
size_type find_first_of(const charT* s,
 size_type pos, size_type n) const;
size_type find_first_of(const charT* s,
 size_type pos = 0) const;
size_type find_first_of(charT c,
 size_type pos = 0) const;
```

Эти методы работают аналогично соответствующим методам **find()**, за исключением того, что вместо поиска строки, полностью совпадающей с подчиненной строкой, они обнаруживают первый случай появления любого отдельного символа из подчиненной строки.

```
string longer("That is a funny hat.");
string shorter("fluke");
size_type loc1 =
 longer.find_first_of(shorter);
// присваивает loc1 значение 10
size_type loc2 =
 longer.find_first_of("fat"); //присваивает
 //loc2 значение 2
```

Первым из четырех символов слова *fluke* в строке *longer* появился символ *c* в слове *funny*. Первое появление одной из букв слова *fat* в строке *longer* зафиксировано в слове *That*.

## Семейство методов **find\_last\_of()**

Методы **find\_last\_of()** обладают следующими прототипами:

```
size_type find_last_of (const basic_string&
 str, size_type pos = npos) const;
size_type find_last_of (const charT* s,
 size_type pos, size_type n) const;
size_type find_last_of (const charT* s,
 size_type pos = npos) const;
size_type find_last_of (charT c,
 size_type pos = npos) const;
```

Эти методы работают так же, как соответствующие методы **rfind()**, но вместо поиска на полное соответствие подчиненной строке они осуществляют поиск последнего появления одного из символов подчиненной строки.

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find_last_of(shorter);
// присваивает loc1 значение 18
size_type loc2 = longer.find_last_of("any");
// присваивает loc2 значение 17
```

Последнее появление одной из трех букв подчиненной строки *hat* в строке *longer* зафиксировано, когда символ *t* появляется в слове *hat*. Последнее появление одного из трех символов, составляющих *any*, в строке *longer* отмечено в слове *hat*.

## Семейство методов **find\_first\_not\_of()**

Методы **find\_first\_not\_of()** обладают следующими прототипами:

```
size_type find_first_not_of(const
 basic_string& str, size_type pos = 0) const;
size_type find_first_not_of(const charT* s,
 size_type pos, size_type n) const;
size_type find_first_not_of(const charT* s,
 size_type pos = 0) const;
size_type find_first_not_of(charT c,
 size_type pos = 0) const;
```

Рассматриваемые методы функционируют так же, как соответствующие им методы **find\_first\_of()**, за исключением того, что они ищут в строке первый случай появления

ления символа, не присутствующего в подчиненной строке.

```
string longer("That is a funny hat.");
string shorter("This");
size_type loc1 = longer.find_first_not_of(shorter);
// присваивает loc1 значение 2
size_type loc2 = longer.find_first_not_of("Thatch");
// присваивает loc2 значение 4
```

Символ **a** в слове **That** является первым символом в строке **longer**, которого нет в слове **This**. Первый пробел в строке **longer** — это первый символ, которого нет в слове **Thatch**.

## Семейство методов `find_last_not_of()`

Методы `find_last_not_of()` обладают следующими прототипами:

```
size_type find_last_not_of (const
 basic_string& str, size_type pos = npos)
 const;
size_type find_last_not_of (const charT* s,
 size_type pos, size_type n) const;
size_type find_last_not_of (const charT* s,
 size_type pos = npos) const;
size_type find_last_not_of (charT c,
 size_type pos = npos) const;
```

Рассматриваемые методы работают подобно методам `find_last_of()`, но они отыскивают последний случай появления символа, не присутствующего в подчиненной строке.

```
string longer("That is a funny hat.");
string shorter("That.");
size_type loc1 = longer.find_last_not_of(shorter);
// устанавливает loc1 в 15
size_type loc2 = longer.find_last_not_of(shorter, 10);
// устанавливает loc2 в 10
```

Последний пробел в строке **longer** является последним символом, который отсутствует в строке **shorter**. Символ **f** в строке **longer** — это последний символ, которого нет в строке **shorter** (если начинать поиск с позиции 10).

## Функции и методы сравнения

Класс строковых значений снабжен методами и функциями, позволяющими сравнивать две строки. Сначала приведем прототипы рассматриваемых методов:

```
int compare(const basic_string& str) const;
int compare(size_type pos1, size_type n1,
 const basic_string& str) const;
int compare(size_type pos1, size_type n1,
 const basic_string& str,
 size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare(size_type pos1, size_type n1,
 const charT* s, size_type n2 = npos) const;
```

Эти методы используют метод `traits::compare()`, установленный для определенного типа символов, применяемых в строковых величинах. Первый метод возвращает величину, которая меньше нуля, если первая строка предшествует второй в соответствии с порядком, предусмотренным методом `traits::compare()`. Он возвращает 0, если две строки одинаковые, или же возвращает величину, которая больше нуля, если вторая строка следует за первой. Если же обе строки идентичны до конца более короткой из двух строк, то считается, что первая более короткая строка предшествует более длинной строке.

```
string s1("bellflower");
string s2("bell");
string s3("cat");
int a13 = s1.compare(s3); // a13 меньше 0
int a12 = s1.compare(s2); // a12 больше 0
```

Второй метод похож на первый, но в нем для сравнения используется только **n1** символов, начиная с позиции **pos1** в первой строке.

```
string s1("bellflower");
string s2("bell");
int a2 = s1.compare(0, 4, s2); // a2 равно 0
```

Третий метод напоминает первый, за исключением того, что в нем для сравнения используются только **n1** символов в первой строке, начиная с позиции **pos1**, и **n2** символов, начиная с позиции **pos2** во второй строке. Например, в следующем фрагменте кода осуществляется сравнение набора символов **out** в слове **stout** со словом **out** в строке **about**:

```
string st1("stout boar");
string st2("mad about ewe");
int a3 = st1.compare(2, 3, st2, 6, 3);
// a3 равно 0
```

Четвертый метод подобен первому, но в нем вместо строкового объекта на месте второй строки используется строковый массив.

Пятый метод похож на третий, за исключением того, что в нем вместо строкового объекта на месте второй строки используется строковый массив.

Функции, осуществляющие сравнение не поэлементно, являются перегруженными операциями сравнения:

```
operator==()
operator<()
operator<=()
operator>()
operator>=()
operator!=()
```

Каждая операция является перегруженной, поэтому с ее помощью можно сравнить два строковых объекта, строковый объект и строковый массив. Они определены на основании метода `compare()`, поэтому представляют более простые пути сравнения, реализуемые благодаря удобству используемых обозначений.

## Модификаторы строковых значений

Для изменения строковых значений класс `string` содержит несколько методов. Некоторые из них поставляются вместе с большим количеством перегруженных версий, поэтому их можно применять к строковым объектам, строковым массивам, отдельным символам и диапазонам итераторов.

### Конкатенирование и дописывание

Одну строку можно прибавить к другой с помощью перегруженной операции `+=` или же с помощью метода `append()`. Оба варианта приводят к генерированию исключительной ситуации `length_error`, если длина результирующей строки превосходит максимально допустимую длину строки. Операция `+=` позволяет прибавлять строковый объект, строковый массив или отдельный символ к другой строковой величине:

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

Методы `append()` также позволяют добавлять строковый объект, массив или отдельный символ к другой строке. Но дополнительно они дают возможность присоединить часть строкового объекта, заданную с помощью начальной позиции и количества символов или посредством указания диапазона. Можно также добавить часть строки, указав количество требуемых символов строки. В варианте, позволяющем дописывать один символ, можно уточнить, сколько раз необходимо скопировать указанный символ.

```
basic_string& append(const basic_string& str);
basic_string& append(const basic_string&
 str, size_type pos, size_type n);
template<class InputIterator>
basic_string& append(InputIterator first,
 InputIterator last);
basic_string& append(const charT* s);
basic_string& append(const charT* s,
 size_type n);
basic_string& append(size_type n, chart c);
// дописывает n копий символа с
```

Рассмотрим несколько примеров:

```
string test("The");
test.append("ory"); //значение строки test
 //равно "Theory"
test.append(3,'!'); //значение строки test
 //равно "Theory!!!"
```

Функция `operator+()` является перегруженной, что позволяет осуществлять конкатенирование строковых величин. Перегруженные функции не изменяют строковые значения; наоборот, они создают новые значения, состоящие из двух совмещенных строк. Функции сложения не являются элементарными, и поэтому они позво-

ляют добавлять строковый объект к строковому объекту или массиву, строковый массив к строковому объекту, отдельный символ к строковому объекту и объект к символу. Вот несколько примеров:

```
string st1("red");
string st2("rain");
string st3 = st1 + "uce"; //строка st3
 //равна "reduce"
string st4 = 't' + st2; //строка st4
 //равна "train"
string st5 = st1 + st2; //строка st5
 //равна "redrain"
```

### Дополнительные средства присваивания

Класс строковых величин поддерживает методы `assign()`, позволяющие присвоить строковому объекту всю строку, ее часть или последовательность одинаковых символов.

```
basic_string& assign(const basic_string&);
basic_string& assign(const basic_string&
 str, size_type pos, size_type n);
basic_string& assign(const charT* s,
 size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
// присваивает n копий символа с
template<class InputIterator>
basic_string& assign(InputIterator first,
 InputIterator last);
```

Ниже приведены несколько примеров:

```
string test;
string stuff("set tubs clones ducks");
test.assign(stuff, 1, 5); //строка test
 //равна "et tu"
test.assign(6, '#'); //строка test
 //равна "#####"
```

### Методы вставки

Метод `insert()` позволяет вставлять строковый объект, массив, символ или несколько символов в строковый объект. Эти методы подобны методам `append()`, но они содержат дополнительный аргумент, показывающий, куда необходимо вставлять новые данные. В роли аргумента может выступать номер позиции или итератор. Новые данные вставляются в позицию, находящуюся перед точкой вставки. Некоторые из методов возвращают ссылку на результирующую строку. Если `pos1` находится за пределами указанной строки или если `pos2` выходит за пределы вставляемой строки, метод генерирует исключительную ситуацию `out_of_range`. Если размер результирующей строки превышает максимально допустимый размер, метод вызывает исключительную ситуацию `length_error`.

```
basic_string& insert(size_type pos1,
 const basic_string& str);
```

```

basic_string& insert(size_type pos1, const
 basic_string& str, size_type pos2,
 size_type n);
basic_string& insert(size_type pos,
 const charT* s, size_type n);
basic_string& insert(size_type pos,
 const charT* s);
basic_string& insert(size_type pos,
 size_type n, charT c);
iterator insert(iterator p, charT c = charT());
void insert(iterator p, size_type n, charT c);
template<class InputIterator>
 void insert(iterator p, InputIterator
 first, InputIterator last);

```

Например, с помощью следующего фрагмента программы строка "former" вставляется перед символом b в строку banker:

```

string st3("The banker.");
st3.insert(4, "former ");

```

Затем следующий фрагмент программы вставляет строку "waltzed" (не считая символа !, который был бы уже девятым) как раз в конец строки "The former banker.":

```

st3.insert(st3.size() - 1, " waltzed!", 8);

```

## Методы удаления

Методы `erase()` удаляют символы из строки. Вот их прототипы:

```

basic_string& erase(size_type pos = 0,
 size_type n =npos);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);

```

Первый метод удаляет с позиций `pos` `n` следующих символов или все оставшиеся до конца строки символы, в зависимости от того, какой из вариантов осуществляется первым. Второй метод удаляет один-единственный символ, указанный позицией итератора, и возвращает итератор на следующий элемент или, если элементов больше нет, возвращает значение `end()`. Третий метод удаляет все символы в диапазоне (символы от первого включительно и до последнего, не включая сам последний символ). Метод возвращает итератор на элемент, следующий за последним удаленным символом.

## Методы замены

Многочисленные методы `replace()` определяют часть строки, которую необходимо заменить, и саму замену. Часть, подлежащую замене, можно указать, задав начальную позицию и количества символов или же диапазон итератора. В качестве замены может выступать строковый объект, строковый массив или какой-нибудь символ, продублированный несколько раз. Строковые объекты или массивы, используемые в качестве замены, позже можно изменить, указав необходимую часть с

помощью номера позиции и количества символов, задав только количество символов или диапазон итератора.

```

basic_string& replace(size_type pos1,
 size_type n1, const basic_string& str);
basic_string& replace(size_type pos1,
 size_type n1, const basic_string& str,
 size_type pos2, size_type n2);
basic_string& replace(size_type pos,
 size_type n1, const charT* s,
 size_type n2);
basic_string& replace(size_type pos,
 size_type n1, const charT* s);
basic_string& replace(size_type pos,
 size_type n1, size_type n2, charT c);
basic_string& replace(iterator il,
 iterator i2, const basic_string& str);
basic_string& replace(iterator il, iterator
 i2, const charT* s, size_type n);
basic_string& replace(iterator il,
 iterator i2, const charT* s);
basic_string& replace(iterator il,
 iterator i2, size_type n, charT c);
template<class InputIterator>
 basic_string& replace(iterator il,
 iterator i2, InputIterator jl,
 InputIterator j2);

```

Приведем несколько примеров:

```

string test("Take a right turn at Main Street.");
test.replace(7,5,"left"); //заменить right
//на left

```

## Другие методы изменения: `copy()` и `swap()`

Метод `copy()` копирует строковый объект или его часть в указанный строковый массив:

```

size_type copy(charT* s, size_type n,
 size_type pos = 0) const;

```

Здесь `s` указывает на массив назначения, `n` указывает число символов для копирования, а `pos` показывает позицию в строковом объекте, с которой нужно начинать копирование. Копирование выполняется для `n` символов или до последнего символа в строке, в зависимости от того, что произойдет раньше. Функция возвращает число скопированных символов. Метод не добавляет пробелы, и поэтому программист должен самостоятельно следить за тем, чтобы размер массива был достаточным для размещения копии.



### ПРЕДОСТЕРЕЖЕНИЕ

Метод `copy()` не добавляет пробелы и не проверяет размер массива назначения.

Метод `swap()` переставляет содержимое двух строк с помощью алгоритма, не зависящего от времени выполнения:

```

void swap(basic_string<charT,traits,Allocator>&);

```

## ВЫВОД И ВВОД ДАННЫХ

Класс `string` перегружает операцию `<<`, предназначенную для вывода на дисплей строковых объектов. Эта операция возвращает ссылку на объект `istream`, что позволяет конкatenировать результаты вывода:

```
string claim("The string class has many
 ↪features.");
cout << claim << endl;
```

Класс `string` перегружает также операцию `>>`, поэтому вводимую информацию можно считывать в строку:

```
string who;
cin >> who;
```

Ввод прекращается после достижения маркера конца файла, после считывания максимально допустимого в строке числа символов или по достижении служебного символа. (Определение служебного символа будет зависеть от набора символов и от реализации типа `charT`.)

Существует две функции `getline()`. Первая имеет следующий прототип:

```
template<class charT, class traits,
 class Allocator>
basic_istream<charT,traits>&
getline(basic_istream<charT,traits,Allocator>& is,
 basic_string<charT,traits,Allocator>&
 str, charT delim);
```

Она считывает символы из потока ввода данных `is` в строку `str` до тех пор, пока не будет достигнут символ разделителя `delim`, конец строки или пока не будет найден признак конца файла. Символ `delim` считывается (функция удаляет его из потока ввода данных), но не сохраняется. Второй вариант функции пропускает третий аргумент и использует символ начала новой строки (или его обобщение) вместо символа `delim`:

```
string str1, str2;
getline(cin, str1); //читывает до
 //символа конца строки
getline(cin, str2, '.'); //читывает до
 //появления точки
```

# Методы и функции библиотеки STL

**Н**азначение библиотеки STL заключается в поиске эффективных методов применения обычных алгоритмов. В этой библиотеке алгоритмы выражены с помощью обобщенных функций, которые затем можно наполнять любыми аргументами, удовлетворяющими требованиям конкретных алгоритмов, и использовать эти функции при образовании классов контейнеров. При составлении этого приложения авторы предполагали, что читатель имеет общее представление о библиотеке STL, хотя бы в пределах изложенного в главе 15. Например, предполагается, что читатель знаком с итераторами и конструкторами.

## Элементы, общие для всех контейнеров

Все контейнеры определяют типы, приведенные в табл. G.1. В этой таблице X обозначает тип контейнера, например, `vector<int>`, а T — тип, хранящийся в контейнере, например, `int`.

Для задания этих элементов определение класса будет использовать `typedef`. Перечисленные типы можно использовать при объявлении подходящих переменных. Например, следующий фрагмент кода иллюстрирует возможный метод, используемый для замены первого появления слова "bonus" в векторе, включающем объекты `string`, на слово "bogus". Из примера видно, как мож-

но воспользоваться типами элементов для объявления переменных:

```
vector<string> input;
string temp;
while (cin >> temp && temp != "quit")
 input.push_back(temp);
vector<string>::iterator want =
 find(input.begin(), input.end(),
 string("bonus"));
if (want != input.end())
{
 vector<string>::reference r = *want;
 r = "bogus";
}
```

В этом фрагменте кода переменная `r` введена как ссылка на элемент в `input`, на который указывает `want`.

Все эти типы можно применять и в более общих программах, в которых типы контейнера и элемента являются обобщенными. Например, предположим, что требуется функция `min()`, возвращающая минимальный элемент и использующая в качестве аргумента ссылку на этот контейнер. Этим подразумевается, что операция < определена для типа величины и что алгоритм `min_element()` библиотеки STL, использующий интерфейс итератора, применяться не будет. В этом случае аргументом может быть `vector<int>`, `list<string>` или `deque<double>`, тогда для представления контейнера сле-

Таблица G.1 Типы, определенные для любых контейнеров.

| Тип                             | Значение                                                                                                                                         |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>X::value_type</code>      | T, тип элемента                                                                                                                                  |
| <code>X::reference</code>       | Работает как <code>T &amp;</code>                                                                                                                |
| <code>X::const_reference</code> | Работает как константа <code>T &amp;</code>                                                                                                      |
| <code>X::iterator</code>        | Тип итератора, указывающего на <code>T *</code> , работает как <code>T *</code>                                                                  |
| <code>X::const_iterator</code>  | Итератор, указывающий на константу T, работает как <code>T *</code>                                                                              |
| <code>X::difference_type</code> | Целочисленный тип со знаком, используемый для представления расстояний от одного итератора до другого; например, разницы между двумя указателями |
| <code>X::size_type</code>       | Целочисленный тип без знака <code>size_type</code> может представлять размер объекта с данными, число элементов и списки индексов                |

дует использовать шаблон и его параметр для **Bag**. Поэтому типом аргумента для функции будет являться **const Bag & b**. А что происходит с типом возвращаемой величины? Этот тип должен соответствовать типу величины контейнера — **Bag::value\_type**. Тем не менее, на этом этапе **Bag** является всего лишь параметром шаблона, и компилятор никаким образом не может узнать, что элемент **value\_type** действительно является типом. Но, чтобы показать, что элемент **class** является **typedef**, можно воспользоваться ключевым словом **typename**:

```
// vector<string> как определяемый класс
vector<string>::value_type st;

// Bag как еще не определенный тип
typename Bag::value_type m;
```

При обработке первого определения компилятор имеет доступ к определению шаблона **vector**, который свидетельствует, что **value\_type** является величиной **typedef**. При считывании второго определения ключевое слово **typename** гарантирует, что конструкция **Bag::value\_type** является величиной **typedef**. Из сказанного следует определение:

```
template<typename Bag>
typename Bag::value_type min(const Bag & b)
{
 typename Bag::const_iterator it;
 typename Bag::value_type m = *b.begin();
 for (it = b.begin(); it != b.end(); ++it)
 if (*it < m)
 m = *it;
 return m;
}
```

Все контейнеры также содержат функции-элементы или операции, перечисленные в табл. G.2. В ней, как и

раньше, **X** обозначает тип контейнера, например, **vector<int>**, а **T** — тип, сохраняемый в контейнере, например, **int**. Как и раньше, величины **a** и **b** относятся к типу **X**.

Для контейнера существование операции **>** предполагает, что эта операция определена для типа величин контейнера. Лексикографическое сравнение является обобщением сортировки по алфавиту. В этом процессе сравниваются элемент за элементом для двух контейнеров до тех пор, пока в одном из контейнеров не найдется элемент, не равный соответствующему элементу второго контейнера. Тогда контейнеры располагаются в порядке, идентичном порядку несуществующей пары элементов. Приведем пример. Пусть два контейнера идентичны для первых десяти элементов, но одиннадцатый элемент первого контейнера меньше, чем одиннадцатый элемент второго контейнера, тогда первый контейнер предшествует второму. Если же контейнеры совпадают вплоть до последнего элемента, то более короткий контейнер предшествует более длинному.

## Дополнительные элементы для векторов, списков и очередей с двухсторонним доступом

Векторы, списки и очереди с двухсторонним доступом являются последовательностями и обладают методами, перечисленными в табл. G.3. В ней, как и раньше, **X** соответствует типу контейнера, например, **vector<int>**, а **T** — типу, сохраняемому в контейнере, например, **int**, а является величиной типа **X**, **t** является величиной типа **X::value\_type**, **i** и **j** являются итераторами ввода, **q2** и **p**

**Таблица G.2 Методы, определенные для всех контейнеров.**

| Метод/операция                 | Описание                                                                                  |
|--------------------------------|-------------------------------------------------------------------------------------------|
| <b>begin()</b>                 | Возвращает итератор к первому элементу                                                    |
| <b>end()</b>                   | Возвращает итератор за позицию конечного элемента                                         |
| <b>rbegin()</b>                | Возвращает обратный итератор за позицию конечного элемента                                |
| <b>rend()</b>                  | Возвращает обратный итератор к первому элементу                                           |
| <b>size()</b>                  | Возвращает количество элементов                                                           |
| <b>maxsize()</b>               | Возвращает наибольший из возможных размеров контейнера                                    |
| <b>empty()</b>                 | Возвращает <b>true</b> , если контейнер пуст                                              |
| <b>swap()</b>                  | Переставляет содержимое двух контейнеров                                                  |
| <b>==</b>                      | Возвращает <b>true</b> , если у двух контейнеров совпадают размеры, элементы и их порядок |
| <b>!=</b>                      | <b>a != b</b> обозначает то же самое, что и <b>!(a == b)</b>                              |
| <b>&lt;</b>                    | Возвращает <b>true</b> , если <b>a</b> в лексикографическом порядке предшествует <b>b</b> |
| <b>&gt;</b>                    | <b>a &gt; b</b> обозначает то же самое, что <b>b &lt; a</b>                               |
| <b>&lt;=</b>                   | <b>a &lt;= b</b> обозначает то же самое, что <b>!(a &gt; b)</b>                           |
| <b>&gt;= operator&gt;=&gt;</b> | <b>a &gt;= b</b> обозначает то же самое, что <b>!(a &lt; b)</b>                           |

Таблица G.3 Методы, определенные для векторов, списков и очередей с двухсторонним доступом.

| Метод                        | Описание                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>a.insert(p,t)</code>   | Вставляется копия <code>t</code> перед <code>p</code> ; возвращает итератор, указывающий на вставленную копию <code>t</code> . Стандартное значение <code>t</code> равно <code>T()</code> , а точнее, соответствует величине, используемой для типа <code>T</code> при отсутствии явной инициализации.                                                                                    |
| <code>a.insert(p,n,t)</code> | Вставляются <code>n</code> копий <code>t</code> перед <code>p</code> ; отсутствует возвращаемая величина.                                                                                                                                                                                                                                                                                 |
| <code>a.insert(p,i,j)</code> | Вставляются копии элементов из диапазона <code>[i,j)</code> перед <code>p</code> , отсутствует возвращаемая величина.                                                                                                                                                                                                                                                                     |
| <code>a.resize(n,t)</code>   | Если <code>n &gt; a.size()</code> , вставляется <code>n - a.size()</code> копий <code>t</code> перед <code>a.end()</code> ; <code>t</code> имеет стандартное значение <code>T()</code> , а именно значение, используемое для типа <code>T</code> при отсутствии явной инициализации.<br>Если <code>n &lt; a.size()</code> , элементы, следующие за <code>n</code> -м элементом удаляются. |
| <code>a.assign(i,j)</code>   | Заменяется текущее содержимое <code>a</code> на копии элементов из диапазона <code>[i,j)</code> .                                                                                                                                                                                                                                                                                         |
| <code>a.assign(n,t)</code>   | Заменяется текущее содержимое на <code>n</code> копий <code>t</code> . Стандартное значение <code>t</code> равно <code>T()</code> , величине, используемой для типа <code>T</code> при отсутствии явной инициализации.                                                                                                                                                                    |
| <code>a.erase(q)</code>      | Удаляется элемент, указанный указателем <code>q</code> ; возвращается итератор к элементу, который следовал за элементом, на который указывал <code>q</code>                                                                                                                                                                                                                              |
| <code>a.erase(q1,q2)</code>  | Удаляет элементы в диапазоне <code>[q1,q2)</code> ; возвращается итератор, указывающий на элемент, на который исходно указывал <code>q2</code> .                                                                                                                                                                                                                                          |
| <code>a.clear()</code>       | То же самое, что <code>erase(a.begin(), a.end())</code> .                                                                                                                                                                                                                                                                                                                                 |
| <code>a.front()</code>       | Возвращается <code>*a.begin()</code> (первый элемент).                                                                                                                                                                                                                                                                                                                                    |
| <code>a.back()</code>        | Возвращается <code>*--a.end()</code> (последний элемент).                                                                                                                                                                                                                                                                                                                                 |
| <code>a.push_back(t)</code>  | Вставляется <code>t</code> перед <code>a.end()</code> .                                                                                                                                                                                                                                                                                                                                   |
| <code>a.pop_back()</code>    | Удаляется последний элемент.                                                                                                                                                                                                                                                                                                                                                              |

являются просто итераторами, `q` и `q1` являются итераторами разыменования (к ним можно применять операцию `*`), а `n` является целочисленной величиной типа `X::size_type`.

В табл. G.4 перечислены методы, общие для двух из трех существующих классов последовательностей.

Дополнительно к перечисленным методам шаблон `list` обладает методами из табл. G.5. Здесь `a` и `b` являются контейнерами `list`, а `T` обозначает тип, сохраняемый в списке, например, `int`, `t` — это величина типа `T`, `i` и `j` — итераторы ввода, `q2` и `p` — итераторы, `q` и `q1` разыменованные итераторы, а `n` — целочисленная величина типа `X::size_type`. В таблице использованы стандартные для библиотеки STL, где `[i, j)` обозначает диапазон от элемента `i` (включительно) до элемента `j`, не включая последний элемент.

## Дополнительные элементы для наборов и карт

Ассоциативные контейнеры, из которых происходят модели для наборов и карт, обладают параметрами шаблонов `Key` и `Compare`. Эти параметры отображают соответственно тип параметра `key`, используемого для упорядочивания содержимого, и тип функционального объекта, называемого *объектом сравнения* и используемого для сравнения ключевых величин. Для контейнеров `set` и `multiset` сохраняемые ключи являются хранящимися величинами, поэтому тип ключей совпадают с типом величин. Для контейнеров `map` и `multimap` сохраняемые величины одного типа (шаблонный параметр `T`) связанны с ключевым типом (шаблонным параметром `Key`), а тип величины является типом `pair<const Key, T>`. Для

Таблица G.4 Методы, определенные для некоторых последовательностей.

| Метод                        | Описание                                                                                                                                          | Контейнер                  |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------|
| <code>a.push_front(t)</code> | Вставляется копия <code>t</code> перед первым элементом.                                                                                          | <code>list, deque</code>   |
| <code>a.pop_front()</code>   | Удаляется первый элемент.                                                                                                                         | <code>list, deque</code>   |
| <code>a[n]</code>            | Возвращается <code>*(a.begin() + n)</code> .                                                                                                      | <code>vector, deque</code> |
| <code>a.at(n)</code>         | Возвращается <code>*(a.begin() + n)</code> , генерируется исключительная ситуация <code>out_of_range</code> , если <code>n &gt; a.size()</code> . | <code>vector, deque</code> |

Таблица G.5 Дополнительные методы для списков.

| Метод                                          | Описание                                                                                                                                                                                                                                                                                                                           |
|------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>a.splice(p,b)</code>                     | Перемещает содержимое списка <b>b</b> в список <b>a</b> , вставляя его перед <b>p</b> .                                                                                                                                                                                                                                            |
| <code>a.splice(p,b,i)</code>                   | Перемещает элемент из списка <b>b</b> , указанный указателем <b>i</b> в позицию, предшествующую <b>p</b> в списке <b>a</b> .                                                                                                                                                                                                       |
| <code>a.splice(p,b,i,j)</code>                 | Перемещает элементы из диапазона <code>[i,j]</code> списка <b>b</b> в позицию, предшествующую <b>p</b> в списке <b>a</b> .                                                                                                                                                                                                         |
| <code>a.remove(const T&amp; t)</code>          | Удаляет из списка все элементы, имеющие значение <b>t</b> .                                                                                                                                                                                                                                                                        |
| <code>a.remove_if(Predicate pred)</code>       | Удаляет все значения, для которых <code>pred(*i)</code> верно, если <b>i</b> является итератором в список <b>a</b> . ( <b>Predicate</b> является булевой функцией или функциональным объектом, как уже обсуждалось в главе 15.)                                                                                                    |
| <code>a.unique()</code>                        | Удаляет все, кроме первого, элементы из каждой группы последовательно повторяющихся равных элементов.                                                                                                                                                                                                                              |
| <code>a.unique BinaryPredicate bin_pred</code> | Удаляет все, кроме первого элемента в группе последовательных элементов, для которых <code>bin_pred(*i, *(i - 1))</code> верно. ( <b>BinaryPredicate</b> является булевой функцией или функциональным объектом, как уже обсуждалось в главе 15.)                                                                                   |
| <code>a.merge(b)</code>                        | Объединяет содержимое списка <b>b</b> с содержимым списка <b>a</b> , используя операцию <code>&lt;</code> , определенную для типа величин. Если же элемент из списка <b>a</b> равен элементу из списка <b>b</b> , элемент списка <b>a</b> располагается первым. Список <b>b</b> после слияния остается пустым.                     |
| <code>a.merge(b, Compare comp)</code>          | Объединяет содержимое списка <b>b</b> с содержимым списка <b>a</b> , используя функцию сравнения <code>comp</code> или функциональный объект. Если же элемент из списка <b>a</b> равен элементу из списка <b>b</b> , то элемент из списка <b>a</b> располагается первым. Список <b>b</b> остается пустым после выполнения слияния. |
| <code>a.sort()</code>                          | Сортируются элементы списка <b>a</b> с помощью операции <code>&lt;</code> .                                                                                                                                                                                                                                                        |
| <code>a.sort(Compare comp)</code>              | Сортируются элементы списка <b>a</b> с помощью функции сравнения <code>comp</code> или функционального объекта.                                                                                                                                                                                                                    |
| <code>a.reverse()</code>                       | Обращается порядок элементов в списке <b>a</b> .                                                                                                                                                                                                                                                                                   |

описания этих характеристик ассоциативные контейнеры включают дополнительные элементы, перечисленные в табл. G.6.

Ассоциативные контейнеры обеспечивают методы, перечисленные в табл. G.7. Вообще, для объекта сравнения совсем не обязательно, чтобы величины с одинаковым ключом совпадали; термин "эквивалентные ключи" обозначает, что две величины, которые могут совпадать или не совпадать, обладают одним и тем же ключом. В таблице X является классом контейнеров, а — объектом типа X. Если X использует уникальные ключи (а именно `set` или `map`), то `a_uniq` является объектом типа X. Если X использует многочисленные ключи (а именно `multiset` или `multimap`), то `a_eq` является объектом

типа X. Как и раньше, i и j являются итераторами ввода, ссылающимися на элементы `value_type`, [i, j] составляют действительный диапазон, p и q2 — это итераторы для a, q и q1 — разыменнованные итераторы для a, [q1, q2] — действительный диапазон, t — величина из X::value\_type (может являться парой), k — величина из X::key\_type.

## Функции STL

Библиотека алгоритмов STL, поддерживаемая заголовочными файлами `algorithm` и `numeric`, обеспечивает большое количество функций, не являющихся элементами и основанных на итераторах. Как рассматривается в главе 15, имена шаблонных параметров выбирают

Таблица G.6 Типы, определенные для ассоциативных контейнеров.

| Тип              | Величина                                                                                                                                                                                                                                                        |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| X::key_type      | Key, ключевой тип                                                                                                                                                                                                                                               |
| X::key_compare   | Compare, величина, имеющая стандартное значение <code>less&lt;key_type&gt;</code>                                                                                                                                                                               |
| X::value_compare | Бинарный тип предиката, совпадающий с <code>key_compare</code> для контейнеров <code>set</code> и <code>multiset</code> и выполняющий упорядочивание для величин <code>pair&lt;const Key, T&gt;</code> в контейнерах <code>map</code> и <code>multimap</code> . |
| X::mapped_type   | T, тип связанных данных (только <code>map</code> и <code>multimap</code> )                                                                                                                                                                                      |

Таблица G.7 Методы, определенные для наборов, множественных наборов, карт и мультикарт.

| Метод                           | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>a.key_comp()</code>       | Возвращает объект сравнения, используемый для конструирования <code>a</code> .                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>a.value_comp()</code>     | Возвращает объект типа <code>value_compare</code> .                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>a.unique.insert(t)</code> | Вставляет величину <code>t</code> в контейнер <code>a</code> тогда и только тогда, когда <code>a</code> не содержит величину с эквивалентным ключом. Метод возвращает величину типа <code>pair&lt;iterator,bool&gt;</code> . Компонент <code>bool</code> принимает истинное значение, если вставка выполняется успешно, и ложное значение в противоположном случае. Компонент итератора указывает на элемент, ключ которого эквивалентен ключу <code>t</code> . |
| <code>a.equal.insert(t)</code>  | Вставляет <code>t</code> и возвращает итератор, указывающий на его местонахождение.                                                                                                                                                                                                                                                                                                                                                                             |
| <code>a.insert(p,t)</code>      | Вставляет <code>t</code> , используя <code>p</code> как указание, с какой позиции <code>insert()</code> должен начать поиск. Если <code>a</code> — контейнер с уникальным ключом, вставка происходит тогда и только тогда, когда <code>a</code> не содержит элемент с эквивалентным ключом; иначе вставка не происходит. Независимо от того, успешно ли происходит вставка, метод возвращает итератор, указывающий на место с эквивалентным ключом.             |
| <code>a.insert(i,j)</code>      | Вставляет элементы из диапазона <code>[i,j)</code> в <code>a</code> .                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>a.erase(k)</code>         | Удаляет все элементы в <code>a</code> , ключи которых эквивалентны <code>k</code> , и возвращает число удаленных элементов.                                                                                                                                                                                                                                                                                                                                     |
| <code>a.erase(q)</code>         | Удаляет элемент, указанный указателем <code>q</code> .                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>a.erase(q1,q2)</code>     | Удаляет элементы в диапазоне <code>[q1,q2)</code> .                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>a.clear()</code>          | То же самое, что <code>erase(a.begin(), a.end())</code> .                                                                                                                                                                                                                                                                                                                                                                                                       |
| <code>a.find(k)</code>          | Возвращает итератор, указывающий на элемент, ключ которого эквивалентен <code>k</code> ; возвращает <code>a.end()</code> , если таких элементов не обнаружено.                                                                                                                                                                                                                                                                                                  |
| <code>a.count(k)</code>         | Возвращает число элементов, имеющих ключи, эквивалентные <code>k</code> .                                                                                                                                                                                                                                                                                                                                                                                       |
| <code>a.lower_bound(k)</code>   | Возвращает итератор к первому элементу с ключом, который не меньше чем <code>k</code> .                                                                                                                                                                                                                                                                                                                                                                         |
| <code>a.upper_bound(k)</code>   | Возвращает итератор к первому элементу, ключ которого больше чем <code>k</code> .                                                                                                                                                                                                                                                                                                                                                                               |
| <code>a.equal_range(k)</code>   | Возвращает пару, первый элемент которой <code>a.lower_bound(k)</code> , а второй элемент — <code>a.upper_bound(k)</code> .                                                                                                                                                                                                                                                                                                                                      |
| <code>a.operator[](k)</code>    | Возвращает ссылку на величину, связанную с ключом <code>k</code> (только для контейнеров <code>map</code> ).                                                                                                                                                                                                                                                                                                                                                    |

ся так, чтобы отразить, какую концепцию моделирует каждый конкретный параметр. Например, `ForwardIterator` используется, чтобы показать, что параметр как минимум моделирует требования прямого итератора, а `Predicate` используется, чтобы отразить тот факт, что параметр должен соответствовать функции с одним аргументом и возвращаемой величиной булевого типа. В стандартной реализации различают четыре группы алгоритмов: неизменяющиеся последовательности операций, изменяющиеся последовательности операций, сортирующие и связанные операции и числовые операции. Термин *последовательность операций* отражает тот факт, что функция использует пару итераторов в качестве аргументов и с их помощью определяет диапазон или последовательность, над которой проводятся операции. Термин *изменяющаяся* обозначает, что функция может изменять контейнер.

## Неизменяющаяся последовательность операций

В табл. G.8 подытожены сведения, относящиеся к неизменяющейся последовательности операций. Аргументы не указаны, а перегруженные функции перечислены

только один раз. Более полное описание, включающее прототипы, приведены после таблицы. Таким образом, достаточно просмотреть таблицу, чтобы получить представление о том, что выполняет функция, и затем, если эта функция требуется, изучить подробности ее реализации.

Теперь обратите внимание на прототипы. Пары итераторов ограничивают диапазон, в то время как выбранный параметр шаблона обозначает тип используемых итераторов. Как обычно, диапазон, записанный в форме `[first, last)`, распространяется на все элементы, начиная с элемента `first` включительно до элемента `last`, не включая его. Некоторые функции используют два диапазона, причем они могут относиться к разным видам контейнеров. Например, чтобы сравнить список сектором, можно воспользоваться функцией `equal()`. Функции, передаваемые в качестве аргументов, являются функциональными объектами, которые могут быть указателями (для которых имя функции становится примером) или объектами, для которых определена операция `()`. Как и в главе 15, предикат является булевой функцией с одним аргументом, а бинарный предикат является булевой функцией с двумя аргументами. (Функции

Таблица G.8 Неизменяющаяся последовательность операций.

| Функция                      | Описание                                                                                                                                                                                                             |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>for_each()</code>      | Применяет неизменяющийся функциональный объект к каждому элементу в диапазоне.                                                                                                                                       |
| <code>find()</code>          | Находит первый случай появления величины в диапазоне.                                                                                                                                                                |
| <code>find_if()</code>       | Находит первое значение, удовлетворяющее критерию проверки предиката в диапазоне.                                                                                                                                    |
| <code>find_end()</code>      | Находит последний случай появления подчиненной последовательности, значения которых совпадают со значениями второй последовательности. Совпадение может устанавливаться с помощью равенства или бинарного предиката. |
| <code>find_first_of()</code> | Находит первый случай появления любого элемента второй последовательности, который совпадает со значением в первой последовательности. Совпадение может устанавливаться с помощью равенства или бинарного предиката. |
| <code>adjacent_find()</code> | Находит первый элемент, который совпадает с элементом, следующим сразу за ним. Совпадение может устанавливаться с помощью равенства или бинарного предиката.                                                         |
| <code>count()</code>         | Возвращает величину, соответствующую количеству появлений данной величины в диапазоне.                                                                                                                               |
| <code>count_if()</code>      | Возвращает количество совпадений данной величины с величинами из диапазона, совпадение определяется заданным бинарным предикатом.                                                                                    |
| <code>mismatch()</code>      | Находит в диапазоне первый элемент, не совпадающий с соответствующим элементом во втором диапазоне, и возвращает итератор к обоим элементам. Совпадение может определяться равенством или бинарным предикатом.       |
| <code>equal()</code>         | Возвращает истинное значение, если каждый элемент в одном диапазоне совпадает с соответствующим элементом в другом диапазоне. Совпадение может устанавливаться с помощью равенства или бинарного предиката.          |
| <code>search()</code>        | Находит первый случай появления подчиненной последовательности, элементы которой совпадают с величинами второй последовательности. Совпадение может устанавливаться с помощью равенства или бинарного предиката.     |
| <code>search_n()</code>      | Находит первую подчиненную последовательность из n элементов, каждый из которых совпадает с данной величиной. Совпадение может устанавливаться с помощью равенства или бинарного предиката.                          |

не обязательно должны принадлежать к типу `bool`, так как они возвращают значение 0 для ложного значения и ненулевое значение — для истинного.)

```
template<class InputIterator, class Function>
Function for_each(InputIterator first,
 InputIterator last, Function f);
```

Функция `for_each()` применяет функциональный объект `f` к каждому элементу диапазона `[first, last)`. Она возвращает `f`.

```
template<class InputIterator, class T>
InputIterator find(InputIterator first,
 InputIterator last, const T & value);
```

Функция `find()` возвращает итератор `it` к первому элементу, имеющему значение `value`, из диапазона `[first, last)`.

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first,
 InputIterator last, Predicate pred);
```

Функция `find_if()` возвращает итератор `it` к первому элементу из диапазона `[first, last)`, для которого вызов функционального объекта `pred(*i)` выдает истинное значение.

```
template<class ForwardIterator1,
 class ForwardIterator2>
```

```
ForwardIterator1 find_end(
 ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2);

template<class ForwardIterator1,
 class ForwardIterator2,
 class BinaryPredicate>
ForwardIterator1 find_end(
 ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2,
 BinaryPredicate pred);
```

Функция `find_end()` возвращает итератор `it` к последнему элементу из диапазона `[first1, last1)`, обозначающему начало подчиненной последовательности, которая соответствует содержимому диапазона `[first2, last2)`. В первой версии для сравнения элементов к типу величины применяется операция `==`. Во второй версии применяется бинарный предикативный функциональный объект `pred` для сравнения элементов. Поэтому элементы, указанные указателями `it1` и `it2`, совпадают, если `pred(*it1, *it2)` принимает истинное значение.

```

template<class ForwardIterator1,
 class ForwardIterator2>
ForwardIterator1 find_first_of(
 ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2);

template<class ForwardIterator1,
 class ForwardIterator2,
 class BinaryPredicate>
ForwardIterator1 find_first_of(
 ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2,
 BinaryPredicate pred);

```

Функция `find_first_of()` возвращает итератор `it` к первому элементу из диапазона `[first1, last1]`, совпадающему с любым элементом из диапазона `[first2, last2]`. В первой версии для сравнения элементов к типу величины применяется операция `==`. Во второй версии для сравнения используется бинарный предикативный функциональный объект `pred`. Поэтому элементы, указанные указателями `it1` и `it2`, совпадают, если `pred(*it1, *it2)` принимает истинное значение.

```

template<class ForwardIterator>
ForwardIterator adjacent_find(
 ForwardIterator first,
 ForwardIterator last);

template<class ForwardIterator,
 class BinaryPredicate>
ForwardIterator adjacent_find(
 ForwardIterator first,
 ForwardIterator last,
 BinaryPredicate pred);

```

Функция `adjacent_find()` возвращает итератор `it` к первому элементу из диапазона `[first1, last1]`, совпадающему со следующим элементом. Функция возвращает `last`, если такая пара не найдена. В первой версии для сравнения элементов к типу величины применяется операция `==`. Во второй версии для сравнения используется бинарный предикативный функциональный объект `pred`. Поэтому элементы, указанные указателями `it1` и `it2`, совпадают, если `pred(*it1, *it2)` принимает истинное значение.

```

template<class InputIterator, class T>
iterator_traits<InputIterator>::difference_type
count(InputIterator first,
 InputIterator last, const T& value);

```

Функция `count()` возвращает число элементов в диапазоне `[first, last]`, совпадающих с величиной `value`. Для сравнения величин к типу величины применяется операция `==`. Возвращаемая величина причисляется к целочисленному типу, достаточному, чтобы вместить максимальное число объектов, которые может содержать контейнер.

```

template<class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type
count_if(InputIterator first,
 InputIterator last, Predicate pred);

```

Функция `count_if()` возвращает число элементов в диапазоне `[first, last]`, для которых функциональный объект `pred` возвращает истинное значение при передаче элемента в качестве аргумента.

```

template<class InputIterator1,
 class InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2);

template<class InputIterator1,
 class InputIterator2,
 class BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 BinaryPredicate pred);

```

В первом варианте функция `mismatch()` находит первый элемент в диапазоне `[first1, last1]`, который не совпадает с соответствующим элементом в диапазоне, начинающимся с элемента `first2`, и возвращает пару, содержащую итераторы для двух несовпадающих элементов. Если же несовпадающие элементы не найдены, возвращается величина `pair<last1, first2 + (last1 - first1)>`. В первой версии для проверки совпадения используется операция `==`. Во второй версии для сравнения элементов используется бинарный предикативный функциональный объект `pred`. Точнее, элементы, указанные указателями `it1` и `it2`, не совпадают, если `pred(*it1, *it2)` принимает ложное значение.

```

template<class InputIterator1,
 class InputIterator2>
bool equal(InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2);

template<class InputIterator1,
 class InputIterator2,
 class BinaryPredicate>
bool equal(InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 BinaryPredicate pred);

```

Функция `equal()` возвращает `true`, если каждый элемент в диапазоне `[first1, last1]` совпадает с соответствующим элементом в последовательности, начинающейся с элемента `first2`, и `false` — в противоположном случае. В первом варианте для сравнения элементов к типу величин применяется операция `==`. Во второй версии для сравнения используется бинарный предикативный функциональный объект `pred`. Поэтому элементы, указанные

указателями `it1` и `it2` совпадают, если `pred(*it1, *it2)` принимает истинное значение.

```
template<class ForwardIterator1,
 class ForwardIterator2>
ForwardIterator1 search(
 ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2);

template<class ForwardIterator1,
 class ForwardIterator2,
 class BinaryPredicate>
ForwardIterator1 search(
 ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2,
 BinaryPredicate pred);
```

Функция `search()` находит в диапазоне `[first1, last1)` первый случай появления последовательности из диапазона `[first2, last2)`. Она возвращает `last1`, если такая последовательность не найдена. В первом варианте для сравнения элементов к типу величин применяется операция `==`. Во второй версии для сравнения используется бинарный предикативный функциональный объект `pred`. Поэтому элементы, указанные указателями `it1` и `it2` совпадают, если `pred(*it1, *it2)` принимает истинное значение.

```
template<class ForwardIterator,
 class Size, class T>
ForwardIterator search_n(
 ForwardIterator first,
 ForwardIterator last,
 Size count,
 const T& value);

template<class ForwardIterator,
 class Size, class T,
 class BinaryPredicate>
ForwardIterator1 search_n(
 ForwardIterator first,
 ForwardIterator last,
 Size count,
 const T& value,
 BinaryPredicate pred);
```

Функция `search_n()` находит первый случай появления последовательности элементов в диапазоне `[first1, last1)`, соответствующей последовательности, состоящей из повторяющихся `count` раз значений `value`. Она возвращает `last1`, если такая последовательность не обнаружена. В первом варианте для сравнения элементов к типу величин применяется операция `==`. Во второй версии для сравнения используется бинарный предикативный функциональный объект `pred`. Поэтому элементы, указанные указателями `it1` и `it2` совпадают, если `pred(*it1, *it2)` принимает истинное значение.

## Изменяющаяся последовательность операций

В табл. G.9 приведены операции, изменяющие последовательность действий. Аргументы в таблице пропущены, а перегруженные функции указаны только один раз. Более подробное описание, включающее описание прототипов, представлено после таблицы. Таким образом, чтобы понять, что выполняет функция, достаточно просмотреть таблицу и узнать о подробностях, если функция признана необходимой.

Теперь перейдем к прототипам. Как было сказано ранее, пары итераторов отмечают диапазоны, тогда как выбранный параметр шаблона указывает на тип итератора. Как обычно, диапазон, заданный в форме `[first, last)`, начинается включительно с элемента `first` до элемента `last`, не включая его. Функции, передаваемые в качестве аргумента, являются функциональными объектами, они могут быть реализованы указателями или объектами, для которых определены операция `()`. Как и в главе 15, предикат является булевой функцией с одним аргументом, а бинарный предикат — булевой функцией с двумя аргументами. (Функции не обязательно принадлежат к типу `bool`, так как они возвращают значение 0 в качестве ложного значения и ненулевую величину — в качестве истинного значения.) Как и в главе 15, унарный функциональный объект является объектом, требующим наличия одного аргумента, а бинарный функциональный объект — объектом, требующим наличия двух аргументов.

```
template<class InputIterator,
 class OutputIterator>
OutputIterator copy(InputIterator first,
 InputIterator last,
 OutputIterator result);
```

Функция `copy()` копирует элементы из диапазона `[first, last)` в диапазон `[result, result + (last - first))`. Она возвращает значение `result + (last - first)`, представляющее собой итератор, который указывает на элемент, следующий за последним скопированным в ячейку памяти элементом. Функция требует, чтобы результат не включался в диапазон `[first, last)` или, другими словами, чтобы результирующий объект не перекрывал исходный объект.

```
template<class BidirectionalIterator1,
 class BidirectionalIterator2>
BidirectionalIterator2
 copy_backward(BidirectionalIterator1 first,
 BidirectionalIterator1 last,
 BidirectionalIterator2 result);
```

Функция `copy_backward()` копирует элементы из диапазона `[first, last)` в диапазон `[result - (last - first), result)`.

Таблица G.9 Операции, изменяющие последовательность действий.

| Функция                         | Описание                                                                                                                                                                                  |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>copy()</code>             | Копирует элементы из диапазона в ячейки, указанные итератором.                                                                                                                            |
| <code>copy_backward()</code>    | Копирует элементы из диапазона в ячейки, указанные итератором. Копирование начинается с конца диапазона и выполняется в обратном направлении.                                             |
| <code>swap()</code>             | Изменяет местоположение двух величин, хранимых по адресам, указанным в ссылках.                                                                                                           |
| <code>swap_ranges()</code>      | Обменивает значения соответствующих величин в двух диапазонах.                                                                                                                            |
| <code>iter_swap()</code>        | Обменивает значения двух величин, хранимых по адресам, указанным итераторами.                                                                                                             |
| <code>transform()</code>        | Применяет функциональный объект к каждому элементу в диапазоне (или к паре элементов в двух диапазонах), копируя возвращаемую величину в соответствующее место другого диапазона.         |
| <code>replace()</code>          | Заменяет каждую появляющуюся указанную величину в диапазоне другой величиной.                                                                                                             |
| <code>replace_if()</code>       | Заменяет каждую появляющуюся указанную величину в диапазоне на другую величину, если предикативный функциональный объект, примененный к исходной величине, возвращает истинное значение.  |
| <code>replace_copy()</code>     | Копирует один диапазон в другой, заменяя каждую появляющуюся указанную величину на другую величину.                                                                                       |
| <code>replace_copy_if()</code>  | Копирует один диапазон в другой, заменяя каждую величину, для которой предикативный функциональный объект принимает истинное значение, другой указанной величиной.                        |
| <code>fill()</code>             | Присваивает каждой величине диапазона указанное значение.                                                                                                                                 |
| <code>fill_n()</code>           | Присваивает значению <i>n</i> последовательных элементов.                                                                                                                                 |
| <code>generate()</code>         | Присваивает каждое значение из диапазона возвращаемой величине генератора, который является функциональным объектом, не требующим аргумента.                                              |
| <code>generate_n()</code>       | Присваивает первые <i>n</i> значений из диапазона возвращаемой величине генератора, который является функциональным объектом, не требующим аргумента.                                     |
| <code>remove()</code>           | Удаляет все появляющиеся указанные значения из диапазона и возвращает итератор к метке конечного элемента для результирующего диапазона.                                                  |
| <code>remove_if()</code>        | Удаляет из диапазона все появляющиеся значения, для которых объект предиката образует истинное значение, и возвращает итератор к метке конечного элемента для результирующего диапазона.  |
| <code>remove_copy()</code>      | Копирует элементы из одного диапазона в другой, пропуская значения, равные указанной величине.                                                                                            |
| <code>remove_copy_if()</code>   | Копирует элементы из одного диапазона в другой, пропуская элементы, для которых предикативный функциональный объект возвращает истинное значение.                                         |
| <code>unique()</code>           | Уменьшает каждую последовательность двух или более равных элементов до одного уникального элемента.                                                                                       |
| <code>unique_copy()</code>      | Копирует элементы из одного диапазона в другой, уменьшая каждую последовательность двух или более элементов до одного уникального элемента.                                               |
| <code>reverse()</code>          | Обращает последовательность элементов в диапазоне.                                                                                                                                        |
| <code>reverse_copy()</code>     | Копирует диапазон в обратном порядке в другой диапазон.                                                                                                                                   |
| <code>rotate()</code>           | Трактует диапазон как окружность и сдвигает элементы влево.                                                                                                                               |
| <code>rotate_copy()</code>      | Копирует один диапазон в другой в обратном порядке.                                                                                                                                       |
| <code>random_shuffle()</code>   | Случайным образом переставляет элементы в диапазоне.                                                                                                                                      |
| <code>partition()</code>        | Помещает все элементы, соответствующие предикативному функциональному объекту, перед всеми элементами, которые не соответствуют ему.                                                      |
| <code>stable_partition()</code> | Помещает все элементы, соответствующие предикативному функциональному объекту, перед всеми элементами, не соответствующими ему, сохраняя относительный порядок элементов в каждой группе. |

Копирует элементы из диапазона в ячейки, указанные итератором.  
 Копирует элементы из диапазона в ячейки, указанные итератором. Копирование начинается с конца диапазона и выполняется в обратном направлении.  
 Изменяет местоположение двух величин, хранимых по адресам, указанным в ссылках.  
 Обменивает значения соответствующих величин в двух диапазонах.  
 Обменивает значения двух величин, хранимых по адресам, указанным итераторами.  
 Применяет функциональный объект к каждому элементу в диапазоне (или к паре элементов в двух диапазонах), копируя возвращаемую величину в соответствующее место другого диапазона.  
 Заменяет каждую появляющуюся указанную величину в диапазоне другой величиной.  
 Заменяет каждую появляющуюся указанную величину в диапазоне на другую величину, если предикативный функциональный объект, примененный к исходной величине, возвращает истинное значение.  
 Копирует один диапазон в другой, заменяя каждую появляющуюся указанную величину на другую величину.  
 Копирует один диапазон в другой, заменяя каждую величину, для которой предикативный функциональный объект принимает истинное значение, другой указанной величиной.  
 Присваивает каждой величине диапазона указанное значение.  
 Присваивает значению *n* последовательных элементов.  
 Присваивает каждое значение из диапазона возвращаемой величине генератора, который является функциональным объектом, не требующим аргумента.  
 Присваивает первые *n* значений из диапазона возвращаемой величине генератора, который является функциональным объектом, не требующим аргумента.  
 Удаляет все появляющиеся указанные значения из диапазона и возвращает итератор к метке конечного элемента для результирующего диапазона.  
 Удаляет из диапазона все появляющиеся значения, для которых объект предиката образует истинное значение, и возвращает итератор к метке конечного элемента для результирующего диапазона.  
 Копирует элементы из одного диапазона в другой, пропуская значения, равные указанной величине.  
 Копирует элементы из одного диапазона в другой, пропуская элементы, для которых предикативный функциональный объект возвращает истинное значение.  
 Уменьшает каждую последовательность двух или более равных элементов до одного уникального элемента.  
 Копирует элементы из одного диапазона в другой, уменьшая каждую последовательность двух или более элементов до одного уникального элемента.  
 Обращает последовательность элементов в диапазоне.  
 Копирует диапазон в обратном порядке в другой диапазон.  
 Трактует диапазон как окружность и сдвигает элементы влево.  
 Копирует один диапазон в другой в обратном порядке.  
 Случайным образом переставляет элементы в диапазоне.  
 Помещает все элементы, соответствующие предикативному функциональному объекту, перед всеми элементами, которые не соответствуют ему.  
 Помещает все элементы, соответствующие предикативному функциональному объекту, перед всеми элементами, не соответствующими ему, сохраняя относительный порядок элементов в каждой группе.

Копирование начинается с элемента в позиции `last - 1`, копируемого в позицию `result - 1`, и продолжается в обратном направлении с последней позиции в позицию `first`. Функция возвращает `result - (last - first)`, другими словами, итератор, указывающий на позицию, следующую сразу за последним скопированным элементом. Функция требует, чтобы результат не попадал в диапазон `[first, last)`. Тем не менее, поскольку копирование происходит в обратном направлении, результирующий и исходный объект могут перекрываться.

```
template<class T> void swap(T& a, T& b);
```

Функция `swap()` выполняет обмен значений, хранящихся по двум адресам, указанным ссылками.

```
template<class ForwardIterator1,
 class ForwardIterator2>
ForwardIterator2 swap_ranges(
 ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2);
```

Функция `swap_ranges()` заменяет значения в диапазоне `[first1, last1)` соответствующими величинами из диапазона, начинающегося с позиции `first2`. Используемые значения не могут перекрываться.

```
template<class ForwardIterator1,
 class ForwardIterator2>
void iter_swap(ForwardIterator1 a,
 ForwardIterator2 b);
```

Функция `iter_swap()` выполняет обмен между значениями, хранящимися по адресам, указанным итераторами.

```
template<class InputIterator, class
 OutputIterator, class UnaryOperation>
OutputIterator transform(
 InputIterator first,
 InputIterator last,
 OutputIterator result,
 UnaryOperation op);
```

Этот вариант функции `transform()` применяет унарный функциональный объект `op` к каждому элементу из диапазона `[first, last)` и присваивает возвращаемую величину соответствующему элементу из диапазона, начинающегося в позиции `result`. Поэтому указатель `*result` устанавливается в позиции `op(*first)`. Он возвращает значение `result + (last - first)`, другими словами, величину, выходящую за пределы результирующего диапазона.

```
template<class InputIterator1,
 class InputIterator2,
 class OutputIterator,
 class BinaryOperation>
OutputIterator transform(InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 OutputIterator result,
 BinaryOperation binary_op);
```

Этот вариант функции `transform()` применяет бинарный функциональный объект `op` к каждому элементу диапазона `[first1, last1)` и к каждому элементу диапазона `[first2, last2)`, а затем присваивает возвращаемое значение соответствующему элементу из диапазона, начинающемуся с позиции `result`. Поэтому указатель `*result` установлен в позиции `op(*first1, *first2)`. Функция возвращает значение `result + (last - first)`, являющееся величиной `past-the-end` для результирующего диапазона.

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first,
 ForwardIterator last,
 const T& old_value,
 const T& new_value);
```

Функция `replace()` заменяет каждое появляющееся значение `old_value` в диапазоне `[first, last)` величиной `new_value`.

```
template<class ForwardIterator,
 class Predicate, class T>
void replace_if(ForwardIterator first,
 ForwardIterator last,
 Predicate pred,
 const T& new_value);
```

Функция `replace_if()` заменяет каждое значение `old` из диапазона `[first, last)`, для которого предикат `pred(old)` возвращает истинное значение `new_value`.

```
template<class InputIterator,
 class OutputIterator, class T>
OutputIterator replace_copy(
 InputIterator first,
 InputIterator last,
 OutputIterator result,
 const T& old_value,
 const T& new_value);
```

Функция `replace_copy()` копирует элементы из диапазона `[first, last)` в диапазон, начинающийся с позиции `result`, но каждый раз, когда встречается значение `old_value`, оно заменяется на значение `new_value`. Функция возвращает `result + (last - first)`, генератор конечного значения для результирующего диапазона.

```
template<class Iterator, class OutputIterator,
 class Predicate, class T>
OutputIterator replace_copy_if(
 Iterator first,
 Iterator last,
 OutputIterator result,
 Predicate pred,
 const T& new_value);
```

Функция `replace_copy_if()` копирует элементы из диапазона `[first, last)` в диапазон, начинающийся с позиции `result`, подставляя значение `new_value` вместо каждой величины `old`, для которой предикат `pred(old)` возвращает истинное значение.

Функция возвращает значение `result + (last - first)`, генератор конечного значения для результирующего диапазона.

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first,
 ForwardIterator last,
 const T& value);
```

Функция `fill()` устанавливает каждый элемент из диапазона `[first, last)` равным `value`.

```
template<class OutputIterator,
 class Size, class T>
void fill_n(OutputIterator first,
 Size n,
 const T& value);
```

Функция `fill_n()` устанавливает каждый из первых `n` элементов, начиная с позиции `first`, равным `value`.

```
template<class ForwardIterator,
 class Generator>
void generate(ForwardIterator first,
 ForwardIterator last,
 Generator gen);
```

Функция `generator()` устанавливает каждый элемент из диапазона `[first, last)` равным `gen()`, где `gen` является генераторным функциональным объектом, т.е. при этом не требуется указывать аргументы. Например, `gen` может быть указателем на `rand()`.

```
template<class OutputIterator,
 class Size,
 class Generator>
void generate_n(OutputIterator first,
 Size n,
 Generator gen);
```

Функция `generator_n()` устанавливает каждый из первых `n` элементов в диапазоне, начинающемся с позиции `first`, равным `gen()`, где `gen` является генераторным функциональным объектом. При этом не требуется указывать аргументы. Например, `gen` может быть указателем на `rand()`.

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first,
 ForwardIterator last,
 const T& value);
```

Функция `remove()` удаляет все значения из диапазона `[first, last)` и возвращает итератор, генерирующий конечное значение для результирующего диапазона. Функция стабильна в том смысле, что порядок оставшихся элементов не изменяется.

#### ПРИМЕЧАНИЕ

Поскольку многочисленные функции `remove()` и `unique()` не являются функциями-элементами и поскольку они не ограничены контейнерами библиотеки STL, они не могут изменить размер контейнера. Наоборот, они возвращают ите-

ратор, показывающий новый адрес конечного значения. Обычно удаленные объекты сдвигаются в конец контейнера. Тем не менее, для контейнеров библиотеки STL можно использовать обратный итератор или итератор метода `erase()`, чтобы восстановить `end()`.

```
template<class ForwardIterator,
 class Predicate>
ForwardIterator remove_if(
 ForwardIterator first,
 ForwardIterator last,
 Predicate pred);
```

Функция `remove_if()` удаляет все значения `val`, для которых предикат `pred(val)` принимает истинные значения из диапазона `[first, last)`, и возвращает итератор в конец последовательности для полученного диапазона. Функция стабильна том в смысле, что порядок неудаленных элементов не изменяется.

```
template<class InputIterator,
 class OutputIterator, class T>
OutputIterator remove_copy(
 InputIterator first,
 InputIterator last,
 OutputIterator result,
 const T& value);
```

Функция `remove_copy()` копирует величины из диапазона `[first, last)` в диапазон, начинающийся с позиции `result`, пропуская при копировании величину `value`. Функция возвращает итератор к концу последовательности для полученного диапазона. Функция стабильна том в смысле, что порядок неудаленных элементов не изменен.

```
template<class InputIterator,
 class OutputIterator,
 class Predicate>
OutputIterator remove_copy_if(
 InputIterator first,
 InputIterator last,
 OutputIterator result,
 Predicate pred);
```

Функция `remove_copy_if()` копирует величины из диапазона `[first, last)` в диапазон, начинающийся с позиции `result`, пропуская при копировании те величины `val`, для которых предикат `pred(val)` принимает истинные значения. Функция возвращает итератор к концу последовательности для полученного диапазона. Функция стабильна том в смысле, что порядок неудаленных элементов не изменен.

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first,
 ForwardIterator last);
```

```
template<class ForwardIterator,
 class BinaryPredicate>
ForwardIterator unique(ForwardIterator first,
 ForwardIterator last,
 BinaryPredicate pred);
```

Функция `unique()` укорачивает каждую последовательность двух или более одинаковых элементов в диапазоне `[first, last)` до одного уникального элемента и возвращает итератор к концу последовательности нового диапазона. В первом варианте для сравнения элементов к типу величины применяется операция `==`. Во втором случае для сравнения используется бинарный предикативный функциональный объект `pred`. Это обозначает, что элементы, указанные указателями `it1` и `it2`, совпадают, если `pred(*it1, *it2)` возвращает истинное значение.

```
template<class InputIterator,
 class OutputIterator>
OutputIterator unique_copy(
 InputIterator first,
 InputIterator last,
 OutputIterator result);

template<class InputIterator,
 class OutputIterator,
 class BinaryPredicate>
OutputIterator unique_copy(
 InputIterator first,
 InputIterator last,
 OutputIterator result,
 BinaryPredicate pred);
```

Этот вариант функции `unique_copy()` копирует элементы из диапазона `[first, last)` в диапазон, начинающийся с позиции `result`, укорачивая каждую последовательность из двух или более одинаковых элементов до одного элемента. Функция возвращает итератор к концу последовательности нового диапазона. В первом варианте для сравнения элементов к типу величины применяется операция `==`. Во втором случае для сравнения используется бинарный предикативный функциональный объект `pred`. При этом элементы, указанные указателями `it1` и `it2`, совпадают, если `pred(*it1, *it2)` возвращает истинное значение.

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first,
 BidirectionalIterator last);
```

Функция `reverse()` обращает порядок в диапазоне `[first, last)`, вызывая функцию `swap(first, last - 1)` и т.д.

```
template<class BidirectionalIterator,
 class OutputIterator>
OutputIterator reverse_copy(
 BidirectionalIterator first,
 BidirectionalIterator last,
 OutputIterator result);
```

Функция `reverse copy()` копирует в обратном порядке элементы из диапазона `[first, last)` в диапазон, начинающийся с позиции `result`. Эти два диапазона не могут перекрываться.

```
template<class ForwardIterator>
void rotate(ForwardIterator first,
 ForwardIterator middle,
 ForwardIterator last);
```

Функция `rotate()` осуществляет левый поворот элементов из диапазона `[first, last)`. Элемент с позиции `middle` сдвигается на позицию `first`, элемент с позиции `middle + 1` сдвигается на позицию `first + 1` и т.д. Элементы, предшествующие `middle` циклически перемещаются к концу контейнера, поэтому элемент, находившийся в позиции `first`, будет следовать за элементом, находящимся в позиции `last - 1`.

```
template<class ForwardIterator,
 class OutputIterator>
OutputIterator rotate_copy(
 ForwardIterator first,
 ForwardIterator middle,
 ForwardIterator last,
 OutputIterator result);
```

Функция `rotate_copy()` копирует элементы из диапазона `[first, last)` в диапазон, начинающийся с позиции `result`, используя вращение последовательности из функции `rotate()`.

```
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first,
 RandomAccessIterator last);
```

Вариант функции `random_shuffle()` перемещает элементы в диапазоне `[first, last)`. Используется равномерное распределение; это обозначает, что все возможные перестановки из исходного состояния равновероятны.

```
template<class RandomAccessIterator,
 class RandomNumberGenerator>
void random_shuffle(
 RandomAccessIterator first,
 RandomAccessIterator last,
 RandomNumberGenerator& random);
```

Еще один вариант функции `random_shuffle()` также перемешивает элементы в диапазоне `[first, last)`. Объект функции `random` определяет распределение. Получая `n` элементов, выражение `random(n)` должно возвратить величину из диапазона `[0, n)`.

```
template<class BidirectionalIterator,
 class Predicate>
BidirectionalIterator partition(
 BidirectionalIterator first,
 BidirectionalIterator last,
 Predicate pred);
```

Функция `partition()` размещает все элементы, значение которых `val` обращает предикат `pred(val)` в истинное значение, перед всеми элементами, для которых эта проверка дает ложное значение. Функция возвращает итератор к позиции, следующей за последней позицией, в кото-

рой находилась величина, обращавшая предикативную объектную функцию в истинное значение.

```
template<class BidirectionalIterator,
 class Predicate>
BidirectionalIterator stable_partition(
 BidirectionalIterator first,
 BidirectionalIterator last,
 Predicate pred);
```

Функция `stable_partition()` размещает все элементы, значения которых `val` обращают в истинное значение предикат `pred(val)`, перед теми элементами, для которых проверка дает ложное значение. Функция сохраняет относительный порядок в каждой из двух групп элементов. Она возвращает итератор к позиции, следующей за последней позицией, в которой находилась величина, обращавшая предикативную объектную функцию в истинное значение.

## Операции сортировки и связывания

В табл. G.10 собраны операции сортировки и связывания. Аргументы не указаны, а перегруженные функции указаны только один раз. Для каждой функции существует вариант, в котором для сравнения элементов используется операция `<`, и вариант, в котором используется функциональный объект, реализующий сравнение. Более подробные сведения, включающие описание прототипов, представлены после таблицы. Таким образом, чтобы понять, что выполняет функция, достаточно воспользоваться таблицей.

Функции, приведенные в таблице, определяют порядок двух элементов с помощью операции `<` или с помощью объекта сравнения, разработанного с помощью шаблонного типа `Compare`. Если `comp` является объектом типа `Compare`, то `comp(a,b)` становится обобщением `a < b` и возвращает `true`, если `a` предшествует `b` в схеме упорядочения. Если выражения `a < b` и `b < a` ложны, то `a` и `b` эквивалентны. Объект сравнения должен обеспечивать, по крайней мере, ограниченное слабое упорядочение. Это обозначает следующее:

Выражение `comp(a,a)` должно быть ложным и являться обобщением того факта, что любое значение не может быть меньше самого себя. (Это ограничивающая часть.)

Если же выражения `comp(a,b)` и `comp(b,c)` принимают истинные значения, тогда `comp(a,c)` также является истинным (что позволяет установить отношение транзитивности для функции сравнения).

Если `a` эквивалентно `b`, а `b` эквивалентно `c`, то `a` эквивалентно `c` (поэтому эквивалентность является отношением транзитивности).

Если операцию `<` применять к целым числам, то эквивалентность предполагает равенство, но это не относится к более общим классам. Например, можно определить структуру из нескольких элементов, содержащих почтовые адреса, и определить объект сравнения `comp`, который сортировал бы структуры по почтовому индексу. Тогда любые два адреса с одинаковыми почтовыми индексами будут эквивалентны, но не совпадающими в точности.

Теперь обратим внимание на прототипы. Разделим этот раздел на несколько подразделов. Как уже упоминалось раньше, пары итераторов отмечают диапазоны, тода как выбранный параметр шаблона указывает на тип итератора. Как всегда, диапазон, заданный в форме `[first, last)`, начинается с позиции `first` включительно и завершается перед позицией `last`. Функции, передаваемые как аргументы, являются функциональными объектами, они могут выражаться через указатели или объекты, для которых определена операция `()`. Как объяснялось в главе 15, предикат является булевой функцией с одним аргументом, а бинарный предикат — булевой функцией с двумя аргументами. (Функции не обязательно принадлежат типу `bool`, так как они возвращают нулевую величину вместо ложного значения и ненулевую величину в качестве истинного значения.) Как и в главе 15, унарный объект функции — это объект, получающий только один аргумент, а бинарный объект функции — объект, получающий два аргумента.

### Сортировка

Сначала рассмотрим алгоритмы сортировки.

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first,
 RandomAccessIterator last);

template<class RandomAccessIterator,
 class Compare>
void sort(RandomAccessIterator first,
 RandomAccessIterator last,
 Compare comp);
```

Функция `sort()` сортирует диапазон `[first, last)` в порядке возрастания, используя для сравнения операцию `<`, определенную для сравниваемых типов. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first,
 RandomAccessIterator last);

template<class RandomAccessIterator,
 class Compare>
void stable_sort(RandomAccessIterator first,
 RandomAccessIterator last,
 Compare comp);
```

Таблица G.10 Операции сортировки и связывания.

| Функция                                 | Описание                                                                                                                                                                                  |
|-----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sort()</code>                     | Сортирует диапазон.                                                                                                                                                                       |
| <code>stable_sort()</code>              | Сортирует диапазон, сохраняя относительный порядок эквивалентных элементов.                                                                                                               |
| <code>partial_sort()</code>             | Частично сортирует диапазон, обеспечивая полную сортировку первых <i>l</i> элементов.                                                                                                     |
| <code>partial_sort_copy()</code>        | Копирует частично упорядоченный диапазон в другой диапазон.                                                                                                                               |
| <code>nth_element()</code>              | Получая итератор в состав диапазона, находит позицию, в которой мог бы находиться элемент, если бы диапазон был упорядочен, и помещает туда этот элемент.                                 |
| <code>lower_bound()</code>              | Получая величину, находит первую позицию в упорядоченном диапазоне, перед которой можно было бы вставить данную величину, не нарушая упорядоченность.                                     |
| <code>upper_bound()</code>              | Получив величину, находит последнюю позицию в упорядоченном диапазоне, перед которой можно было бы вставить данную величину, не нарушая упорядоченность.                                  |
| <code>equal_range()</code>              | Получая величину, находит такой наибольший поддиапазон в упорядоченном диапазоне, в котором данную величину можно вставить перед любым из элементов, не нарушая при этом упорядоченность. |
| <code>binary_search()</code>            | Возвращает истинное значение, если отсортированный диапазон содержит величину, эквивалентную данную величине, или ложное значение в противоположном случае.                               |
| <code>merge()</code>                    | Объединяет два упорядоченных диапазона в третий.                                                                                                                                          |
| <code>inplace_merge()</code>            | Соединяет два последовательных упорядоченных диапазона на месте.                                                                                                                          |
| <code>includes()</code>                 | Возвращает истинное значение, если каждый элемент из одного набора можно отыскать в другом наборе.                                                                                        |
| <code>set_union()</code>                | Конструирует объединение из двух наборов, содержащее все элементы из обоих исходных наборов.                                                                                              |
| <code>set_intersection()</code>         | Конструирует пересечение двух наборов, содержащее только те элементы, которые присутствуют одновременно в обоих исходных наборах.                                                         |
| <code>set_difference()</code>           | Конструирует разницу двух наборов, содержащую только те элементы, которые присутствуют в первом наборе, но не присутствуют во втором.                                                     |
| <code>set_symmetric_difference()</code> | Создает набор, содержащий элементы, присутствующие либо в одном наборе, либо в другом, но не в обоих одновременно.                                                                        |
| <code>make_heap</code>                  | Преобразовывает диапазон в кучу.                                                                                                                                                          |
| <code>push_heap()</code>                | Добавляет элемент в кучу.                                                                                                                                                                 |
| <code>pop_heap()</code>                 | Удаляет наибольший элемент из кучи.                                                                                                                                                       |
| <code>sort_heap()</code>                | Сортирует кучу.                                                                                                                                                                           |
| <code>min()</code>                      | Возвращает минимальное значение из двух заданных.                                                                                                                                         |
| <code>max()</code>                      | Возвращает большее значение из двух заданных.                                                                                                                                             |
| <code>min_element()</code>              | Находит первую появившуюся величину в диапазоне.                                                                                                                                          |
| <code>max_element()</code>              | Находит первую наибольшую величину в диапазоне.                                                                                                                                           |
| <code>lexicographic_compare()</code>    | Сравнивает две последовательности в лексиграфическом порядке, возвращая истинное значение, если первая последовательность меньше, чем вторая, и ложное значение в противоположном случае. |
| <code>next_permutation()</code>         | Производит следующую перестановку в последовательности.                                                                                                                                   |
| <code>previous_permutation()</code>     | Производит предыдущую перестановку в последовательности.                                                                                                                                  |

Функция `stable_sort()` сортирует диапазон `[first, last)`, сохраняя относительный порядок эквивалентных элементов. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class RandomAccessIterator>
void partial_sort(
 RandomAccessIterator first,
 RandomAccessIterator middle,
 RandomAccessIterator last);

template<class RandomAccessIterator,
 class Compare>
void partial_sort(
 RandomAccessIterator first,
 RandomAccessIterator middle,
 RandomAccessIterator last,
 Compare comp);
```

Функция `partial_sort()` частично сортирует диапазон `[first, last)`. Первые элементы, определяемые выражением `middle - first` сортируемого диапазона, помещаются в диапазон `[first, middle)`, а оставшиеся элементы не сортируются. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class InputIterator,
 class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
 InputIterator first,
 InputIterator last,
 RandomAccessIterator result_first,
 RandomAccessIterator result_last);

template<class InputIterator,
 class RandomAccessIterator,
 class Compare>
RandomAccessIterator partial_sort_copy(
 InputIterator first,
 InputIterator last,
 RandomAccessIterator result_first,
 RandomAccessIterator result_last,
 Compare comp);
```

Функция `partial_sort_copy()` копирует первые *n* элементов сортируемого диапазона `[first, last)` в диапазон `[result_first, result_first + n)`. Величина *n* принимает меньшее значение из величин `last - first` и `result_last - result_first`. Функция возвращает `result_first + n`. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first,
 RandomAccessIterator nth,
 RandomAccessIterator last);

template<class RandomAccessIterator,
 class Compare>
void nth_element(RandomAccessIterator first,
 RandomAccessIterator nth,
 RandomAccessIterator last,
 Compare comp);
```

Функция `nth_element()` находит такой элемент в диапазоне `[first, last)`, который оказался бы в *n*-й позиции упорядоченного диапазона, и помещает этот элемент в *n*-ю позицию. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

### Двоичный поиск

При использовании группы алгоритмов двоичного поиска предполагается, что диапазон упорядочен. В этом случае требуются только прямые итераторы, но эти алгоритмы более эффективны при использовании случайных итераторов.

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(
 ForwardIterator first,
 ForwardIterator last,
 const T& value);

template<class ForwardIterator,
 class T, class Compare>
ForwardIterator lower_bound(
 ForwardIterator first,
 ForwardIterator last,
 const T& value,
 Compare comp);
```

Функция `lower_bound()` находит первую позицию в упорядоченном диапазоне `[first, last)`, перед которой можно вставить задаваемое значение, не нарушая при этом порядок. Функция возвращает итератор, указывающий на эту позицию. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(
 ForwardIterator first,
 ForwardIterator last,
 const T& value);

template<class ForwardIterator,
 class T, class Compare>
ForwardIterator upper_bound(
 ForwardIterator first,
 ForwardIterator last,
 const T& value,
 Compare comp);
```

Функция `upper_bound()` находит последнюю позицию в упорядоченном диапазоне `[first, last)`, перед которой можно вставить задаваемую величину, не нарушая при этом порядок. Функция возвращает итератор, указывающий на эту позицию. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first,
 ForwardIterator last,
 const T& value);
```

```
template<class ForwardIterator,
 class T, class Compare>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first,
 ForwardIterator last,
 const T& value,
 Compare comp);
```

Функция `equal_range()` находит наибольший поддиапазон `[it1, it2)` в упорядоченном диапазоне `[first, last)`, такой, что данную величину можно вставить перед любым итератором из этого диапазона, не нарушая при этом порядок. Функция возвращает пару итераторов — `it1` и `it2`. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first,
 ForwardIterator last,
 const T& value);

template<class ForwardIterator,
 class T, class Compare>
bool binary_search(ForwardIterator first,
 ForwardIterator last,
 const T& value,
 Compare comp);
```

Функция `binary_search()` возвращает истинное значение, если в упорядоченном диапазоне `[first, last)` присутствует величина, эквивалентная величине `value`, и ложное значение в ином случае. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

### ПРИМЕЧАНИЕ

Не следует забывать, что, если для упорядочения используется отношение `<`, величины `a` и `b` эквивалентны только тогда, когда несправедливы оба выражения: `a < b` и `b < a`. Для обычных чисел эквивалентность подразумевает равенство, но это не так для структур, сортируемых на основе только одного элемента. Поэтому может найтись не единственная позиция, в которую можно вставить новую величину, не нарушая упорядоченности данных. Подобным образом, если для упорядочивания используется объект сравнения `comp`, эквивалентность обозначает, что оба предиката, `comp[a,b]` и `comp[b,a]`, ложны. (Последнее является обобщением утверждения о том, что `a` и `b` могут быть эквивалентными только в том случае, если `a` не меньше, чем `b`, а `b` не меньше, чем `a`.)

## Слияние

Функции слияния подразумевают, что диапазоны упорядочены.

```
template<class InputIterator1,
 class InputIterator2,
 class OutputIterator>
OutputIterator merge(
 InputIterator1 first1,
 InputIterator1 last1,
```

```
InputIterator2 first2,
InputIterator2 last2,
OutputIterator result);

template<class InputIterator1,
 class InputIterator2,
 class OutputIterator,
 class Compare>
OutputIterator merge(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 OutputIterator result,
 Compare comp);
```

Функция `merge()` объединяет элементы из упорядоченного диапазона `[first1, last1)` и упорядоченного диапазона `[first2, last2)`, помещая результат в диапазон, начинающийся с позиции `result`. Полученный диапазон не должен перекрывать ни один из исходных диапазонов. Если в обоих диапазонах встречаются эквивалентные элементы, то элементы из первого диапазона размещаются перед элементами из второго. Возвращаемая величина является итератором к позиции, находящейся за последней позицией полученного после объединения контейнера. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class BidirectionalIterator>
void inplace_merge(
 BidirectionalIterator first,
 BidirectionalIterator middle,
 BidirectionalIterator last);

template<class BidirectionalIterator,
 class Compare>
void inplace_merge(
 BidirectionalIterator first,
 BidirectionalIterator middle,
 BidirectionalIterator last,
 Compare comp);
```

Функция `inplace_merge()` объединяет два последовательных упорядоченных диапазона — `[first, middle)` и `[middle, last)` — в единую упорядоченную последовательность и хранит ее в диапазоне `[first, last)`. Элементы из первого диапазона предшествуют элементам из второго диапазона. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

## Операции над наборами

Операции над наборами работают с любыми упорядоченными последовательностями, включая наборы `set` и `multiset`. Для контейнеров, содержащих более одного экземпляра значений, например `multiset`, определения становятся обобщенными. Объединение двух множественных наборов содержит большее количество элементов, а пересечение — меньшее количество элементов.

Например, предположим, что множественный набор А содержит строку "apple", встречающуюся семь раз, а множественный набор В содержит ее четыре раза. Тогда объединение А и В будет содержать семь экземпляров строки "apple", а пересечение — только четыре экземпляра.

```
template<class InputIterator1,
 class InputIterator2>
bool includes(InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2);

template<class InputIterator1,
 class InputIterator2, class Compare>
bool includes(InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 Compare comp);
```

Функция `includes()` возвращает `true`, если каждый элемент из диапазона `[first2, last2]` также встречается в диапазоне `[first1, last1]`, и `false` — в противоположном случае. В первом варианте для определения порядка следования элементов используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class InputIterator1,
 class InputIterator2,
 class OutputIterator>
OutputIterator set_union(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 OutputIterator result);

template<class InputIterator1,
 class InputIterator2,
 class OutputIterator,
 class Compare>
OutputIterator set_union(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 OutputIterator result,
 Compare comp);
```

Функция `set_union()` создает набор, являющийся объединением диапазонов `[first1, last1]` и `[first2, last2]`, и копирует результат в то место, куда указывает указатель `result`. Полученный диапазон не должен перекрывать ни один из исходных диапазонов. Функция возвращает итератор к позиции, находящейся за последним элементом созданного диапазона. Объединение — это набор, содержащий все элементы, встречающиеся в каждом из наборов. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class InputIterator1,
 class InputIterator2,
 class OutputIterator>
OutputIterator set_intersection(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 OutputIterator result);

template<class InputIterator1,
 class InputIterator2,
 class OutputIterator,
 class Compare>
OutputIterator set_intersection(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 OutputIterator result,
 Compare comp);
```

Функция `set_intersection()` создает набор, являющийся пересечением диапазонов `[first1, last1]` и `[first2, last2]`, и копирует результат по адресу, указанному указателем `result`. Полученный диапазон не должен перекрывать ни один из исходных диапазонов. Функция возвращает итератор к позиции, находящейся за последним элементом созданного диапазона. Пересечение — это набор, содержащий только элементы, общие для обоих исходных наборов. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class InputIterator1,
 class InputIterator2,
 class OutputIterator>
OutputIterator set_difference(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 OutputIterator result);

template<class InputIterator1,
 class InputIterator2,
 class OutputIterator,
 class Compare>
OutputIterator set_difference(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 OutputIterator result,
 Compare comp);
```

Функция `set_difference()` создает набор, представляющий собой разницу диапазонов `[first1, last1]` и `[first2, last2]`, и помещает полученный диапазон по адресу, указанному указателем `result`. Полученный диапазон не должен перекрывать ни один из исходных диапазонов. Функция возвращает итератор к позиции, находящейся за последним элементом созданного диапазона. Разница

ца — это набор, содержащий только те элементы, которые встречаются в первом наборе, но отсутствуют во втором. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class InputIterator1,
 class InputIterator2,
 class OutputIterator>
OutputIterator set_symmetric_difference(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 OutputIterator result);

template<class InputIterator1,
 class InputIterator2,
 class OutputIterator,
 class Compare>
OutputIterator set_symmetric_difference(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 OutputIterator result,
 Compare comp);
```

Функция `set_symmetric_difference()` создает набор, представляющий собой симметричную разницу диапазонов `[first1, last1]` и `[first2, last2]`, и помещает ее по адресу, указанному указателем `result`. Полученный диапазон не должен перекрывать ни один из исходных диапазонов. Функция возвращает итератор к позиции, находящейся за последним элементом созданного диапазона. Симметричная разница — это набор, содержащий только те элементы, которые встречаются в первом наборе, но отсутствуют во втором, и те элементы, которые присутствуют во втором, но не содержатся в первом. Полученный набор совпадает с разницей между объединением и пересечением. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

### Операции над кучей

Куча — это общая форма данных, сформированная таким образом, что первый элемент в ней является наибольшим. Если же первый элемент удален или добавлен новый элемент, то элементы последовательности представляются так, чтобы сохранить отличительное свойство кучи. Куча спроектирована так, что эти две операции выполняются оптимально.

```
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first,
 RandomAccessIterator last);

template<class RandomAccessIterator,
 class Compare>
```

```
void make_heap(RandomAccessIterator first,
 RandomAccessIterator last,
 Compare comp);
```

Функция `make_heap()` создает кучу из элементов диапазона `[first, last)`. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first,
 RandomAccessIterator last);

template<class RandomAccessIterator,
 class Compare>
void push_heap(RandomAccessIterator first,
 RandomAccessIterator last,
 Compare comp);
```

Функция `push_heap()` предполагает, что диапазон `[first, last - 1)` обладает свойством кучи, и добавляет величину в позицию `last - 1` кучи (т.е. в позицию, следующую за последним элементом подразумеваемой кучи), оставляя за диапазоном `[first, last)` свойство кучи. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first,
 RandomAccessIterator last);

template<class RandomAccessIterator,
 class Compare>
void pop_heap(RandomAccessIterator first,
 RandomAccessIterator last,
 Compare comp);
```

Функция `pop_heap()` предполагает, что диапазон `[first, last)` обладает свойствами кучи. Функция заменяет величину из ячейки `last - 1` величиной из ячейки `first` и образовывает из диапазона `[first, last - 1)` действительную кучу. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first,
 RandomAccessIterator last);

template<class RandomAccessIterator,
 class Compare>
void sort_heap(RandomAccessIterator first,
 RandomAccessIterator last,
 Compare comp);
```

Функция `sort_heap()`, исходя из того, что диапазон `[first, last)` обладает свойством кучи, сортирует его. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

### Минимум и максимум

Функции минимума и максимума возвращают соответственно минимальную и максимальную величины для пары или последовательности величин.

```
template<class T> const T& min(const T& a,
 const T& b);

template<class T, class Compare>
const T& min(const T& a,
 const T& b,
 Compare comp);
```

Функция `min()` возвращает меньшую величину из двух исходных. Если же две величины эквивалентны, функция возвращает первую величину. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class T> const T& max(const T& a,
 const T& b);

template<class T, class Compare>
const T& max(const T& a,
 const T& b,
 Compare comp);
```

Функция `max()` возвращает большую величину из двух исходных. Если же две величины эквивалентны, функция возвращает первую величину. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class ForwardIterator>
ForwardIterator min_element(
 ForwardIterator first,
 ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator min_element(
 ForwardIterator first,
 ForwardIterator last,
 Compare comp);
```

Функция `min_element()` возвращает первый итератор `it` в диапазоне `[first, last)`, такой, что ни один элемент из диапазона не меньше, чем `*it`. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class ForwardIterator>
ForwardIterator max_element(
 ForwardIterator first,
 ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element(
 ForwardIterator first,
 ForwardIterator last,
 Compare comp);
```

Функция `max_element()` возвращает первый итератор `it` в диапазоне `[first, last)`, такой, что не один из элементов диапазона не превышает `*it`. В первом варианте для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class InputIterator1,
 class InputIterator2>
```

```
bool lexicographical_compare(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2);

template<class InputIterator1,
 class InputIterator2,
 class Compare>
bool lexicographical_compare(
 InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2,
 InputIterator2 last2,
 Compare comp);
```

Функция `lexicographical_compare()` возвращает `true`, если последовательность элементов в диапазоне `[first1, last1]` лексикографически меньше, чем последовательность элементов из диапазона `[first2, last2]`, и `false` — в ином случае. Лексикографическое сравнение заключается в сравнении первого элемента одной последовательности с первым элементом другой или, другими словами, в сравнении величин `*first1` и `*first2`. Если `*first1` меньше, чем `*first2`, то функция возвращает `true`, если `*first2` меньше, чем `*first1`, функция возвращает `false`. Если же две величины эквивалентны, то функция переходит к сравнению следующих двух элементов последовательностей. Этот процесс продолжается до тех пор, пока не будет найдена пара неэквивалентных элементов или не будет достигнут конец одной из последовательностей. Если две последовательности эквивалентны до конца одной из них, то более короткая считается меньшей. Если же две последовательности эквивалентны и одинаковы по длине, то функция возвращает `false`. В первом варианте функции для сравнения элементов используется операция `<`, а во втором — объект сравнения `comp`. Лексикографическое сравнение обобщает алфавитное сравнение.

### Перестановки

Перестановка последовательности заключается в изменении порядка элементов в ней. Например, совокупность трех элементов имеет шесть возможных вариантов расположения элементов, потому что существует три варианта выбора первого элемента. После фиксирования элемента на первой позиции остается два варианта выбора второго элемента и один вариант выбора третьего элемента. Например, шесть перестановок цифр 1, 3 и 2 будут следующими:

123 132 213 232 312 321

Вообще, последовательность  $n$  элементов имеет  $n * (n-1) * \dots * 1$  или, по-другому,  $n!$  возможных перестановок.

Перестановочные функции подразумевают, что набор всех возможных перестановок можно расположить в лексикографическом порядке, как шесть перестановок из приведенного выше примера. Это обозначает, что для каждой отдельной перестановки существуют предшествующая и следующая за ней перестановки. Поэтому перестановка 213 предшествует 232, а 312 следует сразу за ней. Тем не менее, первая перестановка (в примере это перестановка 123) не имеет предшественника, а перестановка (321) не имеет последователя.

```
template<class BidirectionalIterator>
bool next_permutation(
 BidirectionalIterator first,
 BidirectionalIterator last);

template<class BidirectionalIterator,
 class Compare>
bool next_permutation(
 BidirectionalIterator first,
 BidirectionalIterator last,
 Compare comp);
```

Функция `next_permutation()` преобразует последовательность в диапазоне `[first, last)` в перестановку, следующую за исходной перестановкой в лексиграфическом порядке. Если следующая перестановка существует, функция возвращает `true`. Если же она не существует (иными словами, диапазон содержит лексикографически последнюю перестановку элементов), функция возвращает `false` и преобразует диапазон в лексикографически первую перестановку. В первом варианте функции для определения порядка используется операция `<`, а во втором — объект сравнения `comp`.

```
template<class BidirectionalIterator>
bool prev_permutation(
 BidirectionalIterator first,
 BidirectionalIterator last);

template<class BidirectionalIterator,
 class Compare>
bool prev_permutation(
 BidirectionalIterator first,
 BidirectionalIterator last,
 Compare comp);
```

Функция `previous_permutation()` преобразует последовательность элементов в диапазоне `[first, last)` в перестановку, предшествующую исходной перестановке в лексиграфическом порядке.

Таблица G.11 Сортирующие и относительные операции.

| Функция                            | Описание                                                                                   |
|------------------------------------|--------------------------------------------------------------------------------------------|
| <code>accumulate()</code>          | Подсчитывает нарастающим итогом для величин из диапазона.                                  |
| <code>inner_product()</code>       | Подсчитывает внутреннее произведение двух диапазонов.                                      |
| <code>partial_sum()</code>         | Копирует частичные суммы из одного диапазона в другой.                                     |
| <code>adjacent_difference()</code> | Копирует присоединенные разности, рассчитанные по элементам, из одного диапазона в другой. |

Если предыдущая перестановка существует, функция возвращает `true`. Если не существует (другими словами, диапазон содержит первую перестановку в лексиграфическом порядке), функция возвращает `false` и преобразовывает диапазон в последнюю перестановку в лексиграфическом порядке. В первом варианте для определения порядка используется операция `<`, тогда как во втором варианте используется объект сравнения `comp`.

## Операции с числами

В табл. G.11 собраны операции с числами, описываемые заголовочным файлом `numeric`. Аргументы не указаны, и перегруженные функции перечислены только один раз. Каждая функция имеет версию, использующую операцию `<` для упорядочивания элементов, и вариант, использующий функциональный объект сравнения. Полное описание, включающее прототипы, следует после таблицы. Таким образом, можно получить представление о работе функций, просмотрев таблицу; или выяснить подробности, если функция окажется необходимой.

```
template <class InputIterator, class T>
T accumulate(InputIterator first,
 InputIterator last, T init);

template <class InputIterator, class T,
 class BinaryOperation>
T accumulate(InputIterator first,
 InputIterator last, T init,
 BinaryOperation binary_op);
```

Функция `accumulate()` устанавливает исходное значение `acc` равным `init`; потом она осуществляет операцию `acc = acc + *i` (первый вариант) или `acc = binary_op(acc, *i)` (второй вариант) для каждого итератора `i` в диапазоне `[first, last)` по порядку. Затем она возвращает полученное значение величины `acc`.

```
template <class InputIterator1,
 class InputIterator2, class T>
T inner_product(InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2, T init);

template <class InputIterator1,
 class InputIterator2, class T,
 class BinaryOperation1,
 class BinaryOperation2>
```

```
T inner_product(InputIterator1 first1,
 InputIterator1 last1,
 InputIterator2 first2, T init,
 BinaryOperation1 binary_op1,
 BinaryOperation2 binary_op2);
```

Функция `inner_product()` устанавливает исходное значение величины `acc` равным `init`; затем она осуществляет операцию `acc = *i * *j` (первый вариант) или `acc = binary_op(*i, *j)` (второй вариант) для каждого итератора `i` в диапазоне `[first1, last1]` по порядку и для каждого соответствующего итератора `j` в диапазоне `[first2, first2 + (last1 - first1)]`. Таким образом, функция подсчитывает величину от первого элемента каждой последовательности, после этого от второго элемента каждой последовательности и т.д., пока не будет достигнут конец первой последовательности. (Поэтому вторая последовательность, по крайней мере, должна быть такой же длины, как первая.) После этого функция возвращает результатирующее значение `acc`.

```
template <class InputIterator,
 class OutputIterator>
OutputIterator partial_sum(
 InputIterator first,
 InputIterator last,
 OutputIterator result);

template <class InputIterator,
 class OutputIterator,
 class BinaryOperation>
OutputIterator partial_sum(
 InputIterator first,
 InputIterator last,
 OutputIterator result,
 BinaryOperation binary_op);
```

Функция `partial_sum()` присваивает `*first` указателю `*result`, `*first + *(first + 1)` — указателю `*(result + 1)` (первый вариант) или `binary_op(*first, *(first + 1))` ука-

зателю `*(result + 1)` (второй вариант) и т.д. Поэтому  $n$ -й элемент последовательности, начинающейся с позиции `result`, содержит сумму (или эквивалентное значение `binary_op`) первых  $n$  элементов последовательности, начинающейся с позиции `first`. Функция возвращает итератор на позицию за последним элементом полученной последовательности. Алгоритм позволяет позиции `result` совпадать с позицией `first`, другими словами, при желании полученный результат можно скопировать поверх исходной последовательности.

```
template <class InputIterator,
 class OutputIterator>
OutputIterator adjacent_difference(
 InputIterator first,
 InputIterator last,
 OutputIterator result);

template <class InputIterator,
 class OutputIterator,
 class BinaryOperation>
OutputIterator adjacent_difference(
 InputIterator first,
 InputIterator last,
 OutputIterator result,
 BinaryOperation binary_op);
```

Функция `adjacent_difference()` присваивает `*first` адрес `result` (`*result = *first`). Следующие адреса в результате присваиваются разностям (или эквивалентам `binary_op`) присоединенных адресов в исходном диапазоне. Это обозначает, что следующему адресу в результате ( `result + 1` ) присваивается `*(first + 1) - *first` (первый вариант) или `binary_op(*(first + 1), *first)` (второй вариант) и т.д. Алгоритм позволяет позиции `result` совпадать с позицией `first`, другими словами, при желании полученный результат можно скопировать поверх исходной последовательности.

# Рекомендуемая литература

Booch, Grady. *Object-Oriented Analysis and Design*. Second Edition. Redwood City, CA: Benjamin/Cummings, 1994.

В этой книге описываются концепции ООП, а также методы ООП и приведены примеры их использования. Примеры написаны на языке C++.

Booch, Grady, Jim Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1998.

Эта книга, написанная создателями Unified Modeling Language (Универсальный язык моделирования), раскрывает суть UML на многочисленных примерах его применения.

Ellis, Margaret A., and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.

Данная книга, называемая обычно ARM (Аннотированное справочное руководство по C++), служит базовым документом по языку C++ стандарта ANSI/ISO. Она не предназначена для изучения языка, но в ней даны ответы на большинство технических вопросов, касающихся функционирования языка. В книге приведены не все дополнения, внесенные комитетом в стандарт ANSI/ISO, тем не менее, ознакомиться с ней стоит.

Jacobson, Ivar. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley, 1994.

В этой книге описаны удачные стратегии и методы, применяемые при разработке объектно-ориентированных приложений (OOSE) для создания крупномасштабных прикладных программных систем.

Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Second Edition*. Reading, MA: Addison-Wesley, 1998.

Книга предназначена для специалистов, владеющих навыками программирования на C++, в ней собрано 50 правил и принципов. Одни из них техничны по суще-

ству, например, пояснение, в каких случаях следует определять конструкторы копирования и операции присваивания. Другие, например касающиеся отношения между "быть" и "иметь", носят более общий характер.

Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Reading, MA: Addison-Wesley, 1996.

Эта книга продолжает в традициях Effective C++ (Эффективный C++) разъяснение некоторых наиболее сложных вопросов языка и показывает, как решить различные задачи, например, как разработать "интеллектуальные" указатели. В ней отражен опыт программистов, пишущих на C++, полученный за последние несколько лет.

Murray, Robert B. *C++ Strategies and Tactics*. Reading, MA: Addison-Wesley, 1993.

Данная книга предназначена для помощи в эффективном изучении языка C++ начинающим программистам. В ней рассмотрены вопросы, касающиеся классов, наследования свойств и методов, шаблонов, исключений и многое другое, предложены практические советы и приведены описания основных применяемых технических приемов.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, Bill Lorensen, William Lorenson. *Object-Oriented Modeling and Design*. Englewood Cliffs, Prentice Hall, 1991.

В этой книге раскрывается техника объектного моделирования (OMT — Object Modeling Technique), метод выявления приемлемых объектов в решаемой задаче.

Rumbaugh, James, Ivar Jacobson, and Grady Booch. *Unified Modeling Reference Manual*. Reading, MA: Addison-Wesley, 1998.

Эта книга, написанная создателями универсального языка для моделирования, представляет собой полное описание UML в форме справочного руководства.

Stroustrup, Bjarne. *The C++ Programming Language. Third Edition.* Reading, MA: Addison-Wesley, 1997.

Страуструп является разработчиком языка C++, поэтому книга написана очень содержательно. Тем не менее, материал, изложенный в ней, усваивается лучше, если читатель уже владеет некоторыми знаниями языка C++. В книге не только изложены основы языка, но и приведены многочисленные примеры его применения, а также рассмотрены методологические аспекты ООП. Успех издания этой книги рос вместе с популярностью языка, в третьем издании обсуждаются элементы стандартной библиотеки, такие как STL и strings.

Stroustrup, Bjarne. *The Design and Evolution of C++.* Reading, MA: Addison-Wesley, 1994.

Эту книгу стоит прочесть тем, кто интересуется, как развивался C++.

## Стандарт ISO/ANSI

Окончательный стандарт можно получить, если отправить письмо по следующему адресу:

American National Standards Institute  
11 W. 42nd St.  
New York, NY 10036  
Global Engineering Documents, Inc.  
15 Inverness Way East  
Englewood, CO 80122-5704

Копии более ранних вариантов CD2 могут быть еще доступны в форматах ASCII, PostScript, HTML и Adobe Acrobat PDF на следующих двух Web-сайтах:

<http://www.setech.com/x3.html>  
<http://www.maths.warwick.ac.uk/c++/pub/>

Более свежую информацию можно получить со следующего сайта FAQ (Frequently Asked Questions - Часто задаваемые вопросы):

<http://reality.sgi.com/austern/std-c++/faq.html>

## Преобразование программного кода в соответствии со стандартом ANSI/ISO C++

Возможно, у вас есть много программ, разработанных на языках С или С++ прежних версий (как и программистских привычек), которые, естественно, хотелось бы преобразовать в соответствии с современным стандартом C++. В этом приложении даны некоторые полезные советы и рекомендации, позволяющие решить эту задачу. Одни из них помогают осуществить переход от С к С++, другие — с устаревших версий С++ к стандартному С++.

### Директивы препроцессора

Препроцессор языка С/С++ содержит набор директив. Вообще, в практике С++ принято использовать те директивы, которые управляют процессом компиляции, и избегать использование тех директив, которые обеспечивают подстановку фрагментов программного кода. Например, директива `#include` является существенным компонентом процесса управления программными файлами. Другие директивы, например, такие как `#ifndef` и `#endif`, позволяют проверять успешность компиляции определенных блоков кода. Директива `#pragma` позволяет управлять специфическими опциями компилятора. Все они являются очень полезными, а иногда просто незаменимыми инструментами. Но при использовании директивы `#define` иногда отображается предупреждение.

### При объявлении констант лучше использовать `const` вместо `#define`

Символические константы делают код более удобным для чтения и правки. Имя константы отражает ее содержание, и, если вам необходимо заменить значение, нужно заменить ее всего лишь один раз в определении и затем заново скомпилировать программу. Язык С для этих целей использует препроцессор:

```
#define MAX_LENGTH 100
```

После этого препроцессор еще до компиляции делает текстовую вставку в исходный код программы, заменив все `MAX_LENGTH` на число 100.

Подход языка С++ состоит в том, чтобы применить модификатор `const` к объявлению переменной:

```
const int MAX_LENGTH = 100;
```

В этом фрагменте компилятор понимает `MAX_LENGTH` как величину только для чтения типа `int`.

В подходе с использованием `const` есть несколько преимуществ. Первое — при объявлении явно указывается тип величины. Для директивы `#define`, чтобы указать тип, отличный от `char`, `int` или `double`, необходимо применять к числу многочисленные суффиксы; например, использовать `100L`, чтобы отразить принадлежность к типу `long`, или `3.14F`, чтобы показать принадлежность к типу `float`. Более важно то, что подход `const` просто согласуется с производными типами:

```
const int base_vals[5] = { 1000, 2000,
 3500, 6000, 10000} ;
const string ans[3] = { "yes", "no",
 "maybe"} ; // "да", "нет", "вероятно"
```

И наконец, идентификаторы `const` подчиняются тому же своду правил, что и переменные. Поэтому существует возможность создавать константы глобального действия, называемые константами из диапазона доступа пространства имен `namespace scope`, и константы из блочного диапазона доступа. Если, скажем, вы определите константу в пределах какой-либо функции, нет оснований тревожиться о возможных конфликтах с определениями глобальных констант, введенных в ином месте программы. Рассмотрим следующий пример:

```
#define n 5
const int dz = 12;
...
void fizzle()
{
 int n;
 int dz;
 ...
}
```

Препроцессор в этом случае заменит

```
int n;
на
int 5;
```

и выдаст сообщение об ошибке компиляции. Дело в том, что константа `dz`, определенная в функции `fizzle()`, все равно останется локальной переменной. Тем не менее, `fizzle()` при необходимости может воспользоваться операцией определения диапазона доступа и получить доступ к константе с помощью `::dz`.

Язык C++ унаследовал ключевое слово `const` от C, при этом вариант, используемый в C++, более удобен. Например, вариант C++ использует метод внутреннего связывания с внешними значениями `const` более корректно, чем переменные и константы `const` языка C используют метод стандартного внешнего связывания. Это значит, что для каждого файла программы, использующего `const`, необходимо, чтобы `const` был определен в конкретном поле. Может показаться, что это лишняя работа, но на самом деле она очень упрощает жизнь. Используя метод внутреннего связывания, можно разместить определения `const` в заголовочном файле, используемом большим количеством файлов проекта. В этом случае компилятор обнаружит ошибки при выполнении внешнего, а не внутреннего связывания. К тому же, поскольку константа `const` должна быть определена в использующем ее файле (размещение этого значения в заголовочном файле, доступном для остальных файлов программы, удовлетворяет этим требованиям), то значение типа `const` можно использовать в качестве аргумента, задающего размер массива:

```
const int MAX_LENGTH = 100;
...
double loads[MAX_LENGTH];
for (int i = 0; i < MAX_LENGTH; i++)
 loads[i] = 50;
```

В языке C такой метод не может использоваться, потому что в C определяющее объявление констант `MAX_LENGTH` может находиться в отдельном файле и не будет доступно при компиляции именно этого файла. Следует отметить, что в C для создания внутренне связанных констант можно использовать модификатор `static`, тогда как в C++ этот модификатор задается по умолчанию.

Однако объявление `#define` все же полезно как часть стандартной идиомы для контроля над компиляцией заголовочного файла:

```
// blooper.h
#ifndef _BLOOPER_H_
#define _BLOOPER_H_
// здесь находится код
#endif
```

Тем не менее, для обычных символьических констант, важно использовать `const` вместо `#define`. Еще одной неплохой альтернативой, особенно когда есть набор связанных целочисленных постоянных, является использование `enum`:

```
enum { LEVEL1 = 1, LEVEL2 = 2,
 LEVEL3 = 4, LEVEL4 = 8 };
```

## Для определения небольших функций используйте ключевое слово `inline` вместо `#define`

В языке C традиционный метод создания приближенных эквивалентов встроенной функции состоял в использовании макроопределения `#define`:

```
#define Cube(X) X*X*X
```

Это заставляет препроцессор выполнить подстановку текста — подставить на место X в `Cube()` соответствующий аргумент:

```
// замена на y = x*x*x;
y = Cube(x);
// замена на x + z++*x + z++*x + z++;
y = Cube(x + z++);
```

Поскольку препроцессор использует подстановки текста, вместо того чтобы в действительности передавать аргумент, использование таких макроопределений может привести к неожиданным и неправильным результатам. Такая ошибка может быть вызвана использованием большого числа скобок в макроопределении, поставленных для гарантии правильного порядка операций:

```
#define Cube(X) ((X)*(X)*(X))
```

Указанная подстановка обеспечивает использование методов совсем не так, как операторы типа `z++`.

Подход C++ в использовании ключевого слова `inline` для идентификации встроенных функций гораздо более надежный, так как при этом выполняется реальная передача аргументов. Более того, встроенные функции языка C++ могут быть регулярными функциями или методами классов.

Одна положительная черта макроопределения `#define` состоит в том, что оно не зависит от типа, поэтому его можно использовать для определения значения любого типа, для которого заданы операции. В C++ можно создать шаблоны `inline`, чтобы иметь доступ к функциям, не зависящим от типа и поддерживающим передачу аргумента.

Итак, используйте в C++ возможность макропрограммирования вместо макроопределений `#define` языка C.

## Используйте прототипы функций

На самом деле, выбора-то нет. Хотя прототипирование носит необязательный характер в С, оно неизбежно в C++. Обратите внимание: функции, которые определены перед их первым использованием, например, как функция макропрограммирования, сами служат своими собственными прототипами.

Действительно, лучше использовать **const** в прототипах функций и заголовках, когда это возможно. В особенности используйте **const** вместе с указывающими параметрами и ссылочными параметрами, представляющими данные, которые не должны изменяться. Такой путь не только помогает компилятору выловить ошибки, влияющие на данные, но и придать функции более общий характер. Поэтому функции с указателем или ссылкой **const** могут обрабатывать как данные типа **const**, так и данные, не относящиеся к этому типу, тогда как функция, которой не удается использовать **const** с помощью указателя или ссылки, может обрабатывать только данные, тип которых отличен от **const**.

## Приведение типов

Один из последователей Страуструпа очень сожалел, что в С присутствует "недисциплинированная" операция приведения типов. Действительно, приведение типов часто является необходимым, но стандартная функция приведения типов слишком непредсказуема. В качестве примера рассмотрим следующий фрагмент кода:

```
struct Doof
{
 double feeb;
 double steeb;
 char sgif[10];
};

Doof leam;
short * ps = (short *) & leam;
// старый синтаксис
int * pi = int * (&leam); // новый синтаксис
```

В языке нет таких ограничений, которые бы не позволили привести указатель на один тип к указателю на совсем иной, не связанный с первым типом.

По сути, ситуация напоминает ситуацию с оператором **goto**. Проблема, связанная с оператором **goto** заключается в том, что оператор является слишком гибким, что приводит к усложнению программного кода. Решить эту проблему можно, создав более ограниченный и структурированный вариант **goto** для выполнения задач, в которых **goto** незаменим. Для решения именно этой проблемы были созданы такие элементы языка, как операторы циклов **for** и **while** и оператор **if else**. В стандартном C++ предусмотрено подобное решение проблемы "недисциплинированности" операции приведения

типов: приведение типов ограничено только теми стандартными ситуациями, в которых без этого не обойтись. Они состоят из операций приведения типов, упомянутых в главе 14:

```
dynamic_cast
static_cast
const_cast
reinterpret_cast
```

Поэтому, если вам необходимо создать операцию приведение типов с использованием указателей, используйте, если возможно, одну из приведенных операций. В этом случае документируется цель приведения и обеспечивается проверка того, что приведение выполнено так, как было указано.

## Познакомьтесь со свойствами языка C++

Если в языке использовались функции **malloc()** и **free()**, в C++ вместо них используются новые функции — **new** и **delete**. Если в С, для обработки ошибок использовались функции **setjmp()** и **longjmp()**, в C++ вместо них лучше использовать **try**, **throw** и **catch**. Попробуйте использовать тип **bool** для логических значений.

## Используйте новую структуру заголовочных файлов

В новом стандарте, как уже упоминалось в главе 2, закреплены новые имена для заголовочных файлов. Если в программах использовались устаревшие названия заголовочных файлов, их следует заменить на новые. Замена необходима не только из эстетических соображений, но и потому, что новые файлы обеспечивают новые возможности. Например, заголовочный файл **ostream** обеспечивает поддержку для ввода и вывода расширенных символов. Он также поддерживает новые манипуляторы, такие как **boolalpha** и **fixed** (они описаны в главе 16). Благодаря новому варианту упрощается интерфейс по сравнению с использованием функций **setf()** или **iomanip** для установки многих опций форматирования. Если вы действительно используете **setf()**, лучше при указании постоянных заменить ее функцией **ios\_base** или **ios**; другими словами, записывать **ios\_base::fixed** вместо **ios::fixed**. Новые заголовочные файлы поддерживают также пространство имен.

## Использование пространств имен

Пространства имен помогают так организовать наименование идентификаторов, используемых в программе, чтобы избежать конфликтов между именами. Поскольку стандартная библиотека, вводимая вместе с новой структурой заголовочных файлов, помещает имена в

пространство имен `std`, использование этих заголовочных файлов требует организации взаимодействия с пространствами имен.

В примерах, приведенных в этой книге, ради простоты применяются директивы `using`, чтобы все имена из именного пространства `std` были доступны:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std; // директива использования
```

Тем не менее, не следует выполнять массовый экспорт всех имен в пространство имен.

Вместо этого мы рекомендуем использовать или декларации `using`, или оператор определения диапазона доступа (`::`), благодаря чему программе будут доступны только те имена, которые необходимы. Например,

```
#include <iostream>
using std::cin; // объявление using
using std::cout;
using std::endl;
```

создает величины `cin`, `cout` и `endl` доступными для всего файла. Использование оператора определения диапазона доступа, тем не менее, делает имя доступным только для выражения, в котором используется этот оператор:

```
// использование операции scope resolution
cout << std::fixed << x << endl;
```

Это может быть утомительным, но можно же собрать все обычно используемые вами объявления в один заголовочный файл:

```
// mynames - заголовочный файл
#include <iostream>
using std::cin; // объявление using
using std::cout;
using std::endl;
```

Следующий шаг заключается в том, что все объявления `using` можно объединить в пространствах имен:

```
// mynames - заголовочный файл
#include <iostream>
namespace io
{
 using std::cin;
 using std::cout;
 using std::endl;
}
namespace formats
{
 using std::fixed;
 using std::scientific;
 using std::boolalpha;
}
```

Тогда программа может подключать этот файл и использовать все необходимые ей пространства имен:

```
#include "mynames"
using namespace io;
```

## Использование шаблона `auto_ptr`

Каждое использование оператора `new` влечет за собой использование оператора `delete`. Проблемы могут возникнуть в том случае, если функция, использующая `new`, была завершена раньше из-за возникновения какой-нибудь ошибки. Как уже обсуждалось в главе 15, использование объекта `auto_ptr` для хранения данных, отключающих объект, созданный `new`, приводит к автоматическому выполнению оператора `delete`.

## Использование класса `string`

Недостаток традиционных С-строк заключается в том, что они не относятся к какому-либо существующему типу данных. Строку можно хранить в символьном массиве, другими словами, можно инициализировать массив символов значениями строки. Однако в этом случае невозможно применить оператор присваивания, чтобы присвоить строке символьный массив; вместо этого необходимо не забывать применять `strcpy()` или `strncpy()`. Недопустимо также применять операции отношения для сравнения С-строк. Вместо этого не следует забывать о применении `strcmp()` (а если забыть и применить, скажем, операцию `>`, то компилятор не выдаст сообщение о синтаксической ошибке, но программа сравнивает адреса строк вместо их содержимого.)

Класс `string` (см. главу 15 и приложение F) позволяет использовать для инициализации строковых величин объекты. Тогда операции присвоения, отношения и добавления (для конкатенации) будут определены. Более того, класс `string` обеспечивает автоматическое управление памятью, поэтому нет причин заботиться о том, что кто-нибудь введет строку, превышающую допустимый размер, и она будет обрезана раньше, чем сохранена.

Класс `string` обеспечивает много удобных методов. Например, можно прибавить один строковый объект к другому, но можно также прибавить С-строку или символ к строковому объекту. К функциям, запрашивающим аргумент в виде С-строки, можно применять метод `c_str()`, возвращающий подходящий указатель на величину типа `char`.

Класс `string` не только обеспечивает удачным продуманным набором методом для выполнения задач, связанных с обработкой строковых величин, например поиск подчиненных строк, но и позволяет так приспособливать программный проект к библиотеке STL, что все алгоритмы STL можно применять к строковым объектам.

## Использование STL

Библиотека Standard Template Library, стандартная библиотека шаблонов (см. главу 15 и приложение G) поставляет готовые решения для многих программных задач, поэтому ее стоит использовать. Например, вместо того чтобы объявлять массив величин типа **double** или объектов **string**, можно создать объект **vector<double>** или **vector<string>**. Преимущества весьма напоминают преимущества использования объектов **string** вместо С-строк. Определена операция присвоения, поэтому ее можно использовать, чтобы присвоить один объект **vector** другому. Можно передать векторный объект с помощью ссылки, а функция, получающая такой объект, может применить к нему метод **size()**, чтобы определить количество элементов в нем. Встроенные алгоритмы управления памятью позволяют осуществлять автоматическое изменение размера в рамках метода **pushback()**, прибавляющего элементы к объекту **vector**. И конечно же для программиста предоставляются несколько методов класса и обобщенных алгоритмов.

Если же необходимо работать со списком, двухсторонней очередью (или просто очередью), стеком, регулярной очередью, набором или картой, то в STL для них предусмотрены шаблоны контейнеров. Библиотека алгоритмов разработана так, что для программиста не представляет труда копирование содержимого вектора в список или сравнение содержимого набора и вектора. Библиотека STL, организованная как набор инструментария, обеспечивает основные блоки, из которых можно собрать необходимый проект.

Расширенная библиотека алгоритмов разработана очень продуманно, поэтому специалист может достигнуть высоких результатов, приложив относительно немного усилий со своей стороны. Понятие итератора, введенное для использования алгоритмов, обозначает, что их использование не ограничено только контейнерами STL. В частности, их можно применить к традиционным массивам.

# Ответы на вопросы для повторения

## Глава 2

- Речь идет о функциях.
- Благодаря этому содержимое файла **iostream** подставляется директивой до окончательной компиляции.
- Благодаря этому определения, введенные в пространство имен **std**, доступны программе.
- ```
cout << "Hello, world\n"; // "Привет, мир"
```
- ```
int cheeses;
```
- ```
cheeses = 32;
```
- ```
cin >> cheeses;
```
- ```
cout << "We have " << cheeses << "
varieties of cheese\n";
// "У нас есть", "сыр", "много сыра"
```
- Это значит, что функция **froop()** должна вызываться с одним аргументом типа **double** и что функция возвратит величину типа **int**.
- return** не используется в функции, если последняя возвращает величину типа **void**.

Глава 3

- Имея более двух целочисленных типов, вы можете выбирать тип, наиболее подходящий для достижения конкретных целей. Например, можно использовать тип **short**, если нужно сэкономить дисковое пространство, тип **long** — если необходимо обеспечить достаточный объем для хранения данных, или можно подыскать такой тип, который ускорит вычисления.

```
2. short rbis = 80;
   // или short int rbis = 80;
unsigned int q = 42110;
   // или unsigned q = 42110;
unsigned long ants = 3000000000;
```



ПРИМЕЧАНИЕ

Не рассчитывайте, что тип **int** может хранить число 3000000000.

- Язык C++ не предусматривает средств, защищающих от превышения целочисленных пределов.
- Константа **33L** принадлежит типу **long**, тогда как константа **33** — типу **int**.
- Два оператора в действительности неэквивалентны, хотя эффекты, которые они оказывают на некоторые системы, совпадают. Более важно то, что первый оператор назначает букву "A" для градуировки только в системе, использующей коды ASCII, тогда как второй оператор действует и для других кодов. **65** является константой типа **int**, тогда как '**'A'**' — константой типа **char**.
- Здесь существуют четыре варианта:


```
char c = 88;
cout << c << "\n"; // тип char выводится
                     // на печать в виде символа
cout.put(char(88)); // put() выводит char
                     // как символ
cout << char(88) << "\n"; // новый метод
                           // приведения типа к типу char
cout << (char)88 << "\n"; // старый метод
                           // приведения типа к типу char
```
- Ответ зависит от того, каковы выбранные типы данных. Если **long** содержит 4 байта, то потеря нет. Так происходит потому, что наибольшая величина типа **long** может составлять около 2 миллиардов, или 10 цифр. Поскольку тип **double** обеспечивает, по меньшей мере, 15 существенных цифр, округление производить не придется.
- a. $8 * 9 + 2 - 72 + 2 - 74$
 b. $6 * 3 / 4 - 18 / 4 - 4$
 c. $3 / 4 * 6 - 0 * 6 - 0$
 d. $6.0 * 3 / 4 - 18.0 / 4 - 4.5$
 e. $15 \% 4 - 3$

9. Подходит любой из вариантов:

```
int pos = (int) x1 + (int) x2;
int pos = int(x1) + int(x2);
```

Глава 4

```
1. a. char actors[30];
b. short betsie[100];
c. float chuck[13];
d. long double dipsea[64];

2. int oddly[5] = { 1, 3, 5, 7, 9 };

3. int even = oddly[0] + oddly[4];

4. cout << ideas[1] << "\n"; // или << endl;

5. char lunch[13] = "cheeseburger";
   // число символов + 1; "чизбургер"
```

или

```
char lunch[] = "cheeseburger";
// пусть компилятор подсчитает элементы
```

```
6. struct fish {
    char kind[20];
    int weight;
    float length;
};
```

```
7. fish petes =
{
    "trout",
    13,
    12.25
};
```

```
8. enum Response { No, Yes, Maybe } ;
```

```
9. double * pd = &ted;
cout << *pd << "\n";
```

```
10. float * pf = treacle; // или = &treacle[0]
cout << pf[0] << " " << pf[9] << "\n";
// или использовать *pf и *(pf + 9)
```

```
11. unsigned int size;
cout << "Enter a positive integer: ";
cin >> size;
int * dyn = new int [size];
```

12. Да, так можно. Выражение "Home of the jolly bytes" ("Дом веселых байтов") является строковой константой, поэтому она обрабатывается как адрес начала строки. Объект `cout` интерпретирует адрес типа `char` как приглашение к выводу на печать строки, но операция приведения (`int *`) преобразует адрес к типу указателя на величину `int`, которая позже отображается как адрес. Одним словом, оператор выводит на печать адрес строки.

13. struct fish

```
{
    char kind[20];
    int weight;
    float length;
};

fish * pole = new fish;
cout << "Enter kind of fish: ";
cin >> pole->kind;
```

14. Использование `cin >> address` заставляет программу игнорировать пробелы до тех пор, пока она не найдет символ, не являющийся пробелом. Затем она считывает символы до тех пор, пока снова не найдет пробел. Таким образом, она пропустит символ начала новой строки, следующий за численным вводом, вследствие чего проблем не возникнет. С другой стороны, она будет считывать только одинокие слова, а не всю строку.

Глава 5

1. Условие начала цикла проверяет текстовое выражение до перехода к телу цикла. Если условие изначально ложное, тело цикла никогда не будет выполняться. Условие окончания цикла обрабатывает текстовое выражение после обработки тела цикла. Поэтому тело цикла выполняется один раз, даже если выражение исходно принимает ложное значение. Циклы `for` и `while` являются циклами с предусловием, а цикл `do while` — циклом с постусловием.

2. Будет напечатано следующее:

01234

Обратите внимание, что `cout << "\n";` не является частью тела цикла (нет скобок).

3. Будет напечатано следующее:

0369
12

4. Будет напечатано следующее:

6
8

5. Будет напечатано следующее:

k = 8

6. Простейший способ — воспользоваться операцией `*=`:

```
for (int num = 1; num <= 64; num *= 2)
    cout << num << " ";
```

7. Операторы следует заключить в парные скобки, чтобы сформировать единый оператор или блок.

8. Да, первый оператор правильный. Выражение **1,024** состоит из двух выражений (**1** и **024**), объединенных

оператором запятой. Значение всего выражения является значением правого выражения. Это составляет **024**, или в восьмеричной записи **20**, поэтому в результате объявления присваивается значение **20** величине **x**. Второй оператор также правилен. Тем не менее, из-за приоритета операций выражение выполняется в следующей последовательности:

(y = 1), 024;

Другими словами, в левом выражении у приравнивается к **1**, а значением неиспользуемого выражения является **024** или **20**.

9. Форма **cin >> ch** пропускает пробелы, символы новой строки и метки табуляции, как только они попадаются. Оставшиеся две формы считывают эти символы.

Глава 6

- Оба варианта дают одинаковые ответы, но вариант **if else** более удобен. Рассмотрим, что происходит в случае, если **ch** является пробелом. Вариант 1: после приращения количества пробелов, проверяется, не является ли символ символом новой строки. На это попусту уходит время, потому что программа уже установила, что **ch** является пробелом и поэтому не может быть символом новой строки. Вариант 2 в подобной ситуации пропускает проверку символа новой строки.
- Оба оператора, **++ch** и **ch + 1**, имеют одинаковые числовые значения. Но **++ch** принадлежит к типу **char** и выводится на печать в виде символа, тогда как **ch + 1** прибавляет к величине типа **char** величину **int**, принадлежит к типу **int** и выводится на печать как число.
- Поскольку программа использует **ch = '\$'** вместо **ch == '\$'**, комбинированные ввод и вывод выглядят следующим образом:

```
hi!
$Hi$!$  
$Send $10 or $20 now!
$e$n$d$ $ct1 = 9, ct2 = 9
```

Каждый символ перед выполнением вторичного вывода на печать преобразуется в символ **\$**. К тому же значение выражения **ch = \$** является кодом для символа **\$**, поэтому принимает ненулевое, следовательно, истинное значение; таким образом, значение **ct2** каждый раз увеличивается.

- a. **weight >= 115 && weight < 125**
b. ch == 'q' || ch == 'Q'
c. x % 2 == 0 && x != 26
d. donation >= 1000 && donation <= 2000 || guest == 1
e. (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')

- Не обязательно. Например, если **x** равен **10**, тогда **!x** равен **0**, а **!!x** равен **1**. Тем не менее, если **x** является булевой переменной, тогда **!!x** равен **x**.

- (x < 0)? -x : x**

- switch (ch)**

```
{
    case 'A': a_grade++;
    break;
    case 'B': b_grade++;
    break;
    case 'C': c_grade++;
    break;
    case 'D': d_grade++;
    break;
    default: f_grade++;
    break;
}
```

- Если в программе использованы целочисленные метки, а пользователь вводит нецелочисленную величину, скажем **q**, то программа "зависает", так как не может обработать введенный символ. Если же в программе используются символьные метки, а пользователь вводит целочисленную величину, например **5**, символьный ввод обработает **5** как символ. Затем стандартная часть ключа может ожидать ввода другого символа.

- Вот один из вариантов:

```
int line = 0;
char ch;
while (cin.get(ch) && ch != 'Q')
{
    if (ch == '\n')
        line++;
}
```

Глава 7

- Эти три этапа спределят функцию, обеспечивают прототип, а затем вызывают ее.
- a. **void igor(void);**
b. float tofu(int n); // или float tofu(int);
c. double mpg(double miles, double gallons);
d. long summation(long harray[], int size);
e. double doctor(const char * str);
f. void ofcourse(boss dude);
g. char * plot(map *pmap);

```
3. void set_array(int arr[], int size, int value)
{
    for (int i = 0; i < size; i++)
        arr[i] = value;
}
```

```
4. double biggest (const double foot[], int size)
{
    double max;
    if (size < 1)
    {
        cout << "Invalid array size of "
            << size << "\n";
        cout << "Returning a value of 0\n";
        return 0;
    }
    else // необязательно, так как
          // return завершает программу
    {
        max = foot[0];
        for (int i = 1; i < size; i++)
            if (foot[i] > max)
                max = foot[i];
        return max;
    }
}
```

5. Здесь используется спецификатор **const** вместе с указателями, чтобы защитить от изменения исходные данные, на которые указывает указатель. Когда программа передает величины фундаментальных типов, скажем **int** или **double**, она передает их значения, поэтому эта функция работает с копией. Таким образом, исходные данные уже защищены.

```
6. int fill_array(double ar[], int limit)
{
    double temp;
    for (int i = 0; i < limit; i++)
    {
        cout << "Enter value #" << i + 1
            << ": ";
        if (!(cin >> temp)) // не числовой ввод
        {
            cin.clear(); // переустановка ввода
            while (cin.get() != '\n')
                continue; // заново
            // осуществляется ввод
            break;
        }
        ar[i] = temp;
    }
    return i;
}
```

7. Строку можно сохранить в символьном массиве **char**, ее можно представить с помощью строковой константы, заключенной в двойные кавычки, или с помощью указателя, указывающего на первый символ строки.

```
8. int replace(char * str, char c1, char c2)
{
    int count = 0;
    while (*str) // и пока не достигнет
                // конца строки
    {
        if (*str == c1)
        {
            *str = c2;
            count++;
        }
        str++; // переход на следующий символ
    }
    return count;
}
```

9. Поскольку C++ интерпретирует строку "pizza" как адрес первого элемента строки, в результате применения операции ***** отображается значение первого элемента, являющегося символом р. Так как C++ интерпретирует "taco" как адрес первого элемента этой строки, то запись "taco" [2] интерпретируется как значение элемента, стоящего через две позиции от начального символа, что соответствует символу с. Другими словами, строковая константа действует так, как имя массива.

10. Чтобы передать непосредственно значение, следует просто передать имя структуры **glitz**. Чтобы передать значение с помощью адреса, следует воспользоваться операцией адресации **&glitz**. Передача самой величины автоматически защищает исходные данные, но этот процесс поглощает время и дисковое пространство. Передача с помощью адреса сэкономит время и пространство, но не защитит исходные данные, если, конечно, не применить к параметру функции атрибут **const**. К тому же передача значения позволяет использовать обычный стиль записи элемента структуры, а передача с помощью указателя приводит к тому, что программист должен не забывать об использовании операции косвенного членства.

11. int judge (int (*pf)(const char *));

Глава 8

1. Небольшие, нерекурсивные функции, записываемые в одной строке.

2. a. void song(char * name, int times = 1);

b. Никакой (ни один). Только прототипы содержат информацию о стандартных значениях.

c. Да, предполагается, что программист помнит стандартное значение **times**:

```
void song(char * name = "O, My Papa",
          int times = 1);
```

3. Для вывода на печать кавычек можно использовать или строку "\\"", или символ '\"'. В приведенной ниже функции иллюстрируются оба подхода:

```
#include <iostream.h>
void iquote(int n)
{
    cout << "\"" << n << "\"";
}

void iquote(double x)
{
    cout << '\"' << x << '\"';
}

void iquote(const char * str)
{
    cout << "\"" << str << "\"";
}
```

4. a. Эта функция не должна изменить элементы структуры, поэтому следует использовать атрибут **const qualifier**.

```
void show_box(const box & container)
{
    cout << "Made by " << container.maker
        << "\n"; // "Сделано"
    cout << "Height = " << container.height
        << "\n"; // "Высота"
    cout << "Width = " << container.width
        << "\n"; // "Бес"
    cout << "Length = " << container.length
        << "\n"; // "Длина"
    cout << "Volume = " << container.volume
        << "\n"; // "Объем"
}
```

b. void set_volume(box & crate)
{
 crate.volume = crate.height *
 crate.width * crate.length;
}

5. a. Это можно осуществить с помощью стандартного значения второго аргумента:

```
double mass(double d, double v = 1.0);
```

Можно осуществить, также, с помощью перегрузки:

```
double mass(double d, double v);
double mass(double d);
```

b. Нельзя воспользоваться стандартным значением для итерируемой величины, потому что необходимо предусмотреть стандартные значения в направлении "справа налево". Можно использовать перегрузку:

```
void repeat(int times, const char * str);
void repeat(const char * str);
```

c. Можно использовать перегруженные функции:

```
int average(int a, int b);
double average(double x, double y);
```

d. Так делать нельзя, так как оба варианта приняли бы одинаковые сигнатуры.

e. По крайней мере, один вариант должен быть определен в качестве статической функции в одном из файлов:

```
static int average(int a, int b);
// определение в файле 1
static double average(int a, int b);
// определение в файле 2
```

6. template<class T>
T max(T t1, T t2) // или T max(const T &
 // t1, const T & t2)
{
 return t1 > t2? t1 : t2;
}

7. template<> box max(box b1, box b2)
{
 return b1.volume > b2.volume? b1 : b2;
}

8. a. **homer** автоматически становится автоматической переменной.

b. **secret** следует определить как внешнюю переменную в одном файле и объявить использование **extern** во втором файле.

c. **topsecret** следует определить как статическую внешнюю переменную, предварив объявление внешней переменной ключевым словом **static**.

d. **beencalled** следует определить как локальную статическую переменную, предварив объявление в функции ключевым словом **static**.

9. Объявление **using** создает доступ к уникальному имени из пространства имен, диапазон доступа соответствует области объявления, в которой появилось объявление **using**. Директива **using** создает доступ ко всем именам в пространстве имен. Введение директивы **using** соответствует объявлению имен в наименьшей области объявления, состоящей из декларации использования и самого пространства имен.

Глава 9

1. Класс — это определение типа, введенного пользователем. В объявлении класса указывают, в каком виде должны храниться данные и какие методы (функции-элементы класса) следует использовать для доступа к данными и работы с ними.

2. Класс представляет операции, которые можно осуществить над объектами класса с помощью общего интерфейса методов класса; это абстракция. Класс может использовать приватную видимость (стандарт-

ные значения) элементов данных, имея в виду, что данные можно получить только через функции-элементы; это сокрытие данных. Подробности применения, например представление данных и программирование методов, скрываются; это инкапсуляция данных.

3. Класс определяет тип, в том числе и то, как его можно использовать. Объект класса – это переменная или другой объект данных, который может быть порожден с помощью оператора `new`, создается и используется согласно определению класса. Отношение между ними такое же, как между стандартным типом и переменной этого типа.
4. Если создано несколько объектов данного класса, каждый объект поступает вместе с хранилищем для собственного набора данных. Но все объекты используют один общий набор функций-элементов. (Обычно методы являются общими, а элементы данных – приватными, но это не всегда требуется.)

ПРИМЕЧАНИЕ

Для считывания имен программы использует конструкцию `cin.get(char *, int)` вместо `cin >>`, поскольку оператор `cin.get()` считывает всю строку целиком вместо одного слова (см. главу 4).

```
5. #include <iostream>
using namespace std;
// определение класса
class BankAccount
{
private:
    char name[40];
    char acctnum[25];
    double balance;
public:
    BankAccount(char * client, char *
        num, double bal = 0.0);
    void set(void);
    void show(void) const;
    void deposit(double cash);
    void withdraw(double cash);
};
```

6. Конструктор класса вызывается при необходимости создания объекта вызываемого класса или при явном обращении к конструктору. Деструктор класса вызывается при уничтожении объекта.

7. Обратите внимание, что необходимо включить `cstring` или `string.h`, чтобы можно было применять `strcpy()`.

```
BankAccount::BankAccount(char * client,
    char * num, double bal)
{
    strcpy(name, client, 39);
    name[39] = '\0';
    strncpy(acctnum, num, 24);
    acctnum[24] = '\0';
    balance = bal;
}
```

Не забывайте, что стандартные аргументы поставляются прототипом, а не определением функции.

8. Стандартный конструктор – это конструктор без аргументов или, по-другому, с аргументами, принимающими стандартные значения. Его использование позволяет объявлять объекты, не инициализируя их, даже если инициализирующий конструктор уже определен. Можно также объявить массив.

```
9. // stock3.h
#ifndef _STOCK3_H_
#define _STOCK3_H_

class Stock
{
private:
    char company[30];
    int shares;
    double share_val;
    double total_val;
    void set_tot()
    {
        total_val = shares * share_val;
    }
public:
    Stock();           // стандартный конструктор
    Stock(const char * co, int n, double pr);
    ~Stock() { } // ничего не выполняющий
                  // деструктор
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show() const;
    const Stock & topval(const Stock & s) const;
    int numshares() const { return shares; }
    double shareval() const { return share_val; }
    double totalval() const { return total_val; }
    const char * co_name() const
    {
        return company;
    }
};
```

10. Указатель `this` является указателем, доступным для методов класса. Он указывает на объект, используемый для выполнения метода. Таким образом, `this` – адрес объекта, а `*this` представляет сам объект.

Глава 10

1. Ниже приведен прототип файла определения класса и определение функции для файла метода:

```
// прототип
Stonewt operator*(double mult);

// определение позволяет конструктору
// выполнить свое задание
Stonewt Stonewt::operator*(double mult)
{
    return Stonewt(mult * pounds);
}
```

2. Функция-элемент – часть определения класса, она вызывается определенным объектом. Функция-элемент может получить элемент вызываемого объекта явным образом, не используя оператор принадлеж-

ности. Дружественная функция не является частью класса, поэтому она вызывается путем прямого функционального вызова. Она не имеет явного доступа к элементам класса, поэтому она использует оператор принадлежности, применяемый к объекту, передаваемому в качестве аргумента.

3. Чтобы получить доступ к приватным элементам, необходимо вступать в дружественные отношения, однако совсем необязательно вступать в дружественные отношения, чтобы получить доступ к общим элементам.

4. Ниже приведен прототип для файла определения класса и определение функции для файла методов:

```
// прототип
friend Stonewt operator*(double mult,
                           const Stonewt & s);

// определение — позволяет конструктору
// выполнить работу
Stonewt operator*(double mult,
                     const Stonewt & s)
{
    return Stonewt(mult * s.pounds);
}
```

5. Следующие пять операторов нельзя перегрузить:

```
sizeof . .* :: ?:
```

6. Эти операторы необходимо определить с помощью функций, являющихся элементами.

7. Ниже приведен возможный прототип и определение:

```
// прототип и определение,
// занимающее одну строку
operator double () { return mag; }
```

Тем не менее, обратите внимание, что разумнее использовать метод `magval()`, чем определить эту функцию преобразования.

Глава 11

1. а. Синтаксис применен правильно, но этот конструктор не инициализирует указатель `str`. Конструктор должен или устанавливать указатель на `NULL`, или использовать метод `new []`, чтобы инициализировать указатель.

б. Этот конструктор не создает новую строку; он просто копирует адрес старой строки. Он должен использовать методы `new []` и `strcpy()`.

с. Копируется строка без выделения памяти для ее хранения. Необходимо использовать `new char[len + 1]`, чтобы выделить необходимый объем памяти.

2. Во-первых, когда объект этого типа исчезает, данные, указанные указателем на элемент объекта, остаются в памяти, занимая место и оставаясь недоступными, так как указатель на них давно утерян. Этот можно исправить, если удалить с помощью деструктора класса область памяти, выделенную методами `new` для функций конструктора. Во-вторых, как только деструктор удаляет такие области памяти, он может начать удалять их дважды, если программа инициализирует один такой объект другим. Вот почему в процессе стандартной инициализации одного объекта значениями другого объекта копируются значения указателей, но не данные, на которые эти указатели указывают, и в результате рождаются два указателя на один и тот же объект. Решение проблемы состоит в том, чтобы определить конструктор класса копирования, который объединит процесс инициализации с копированием указанных данных. И в-третьих, присваивание одного объекта другому может привести к появлению, опять же, двух указателей, указывающих на один и тот же объект. Чтобы избежать этого, следует перегрузить оператор присваивания так, чтобы он копировал данные, а не указатели на них.

3. C++ автоматически поддерживает следующие функции-элементы:

- Стандартный конструктор, если ни один конструктор не определен
- Конструктор копирования, если он не определен
- Операцию присваивания, если она не определена
- Стандартный деструктор, если он не определен
- Адресную операцию, если она не определена

Стандартный конструктор не выполняет никаких функций, но он дает возможность объявлять массивы и инициализировать объекты. Стандартный конструктор копирования и стандартная операция присваивания используют метод поэлементного присваивания. Стандартный деструктор не выполняет никаких функций. При выполнении явной адресации возвращается адрес оператора вызываемого объекта (другими словами, значение указателя `this`).

4. Элемент `personality` следует объявить так же, как символьный массив или как указатель на символ. Или можно создать его как объект `String`. Вот два возможных решения, в которых изменения (отличные от операций удаления) выделены жирным шрифтом.

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // необязательный блок
char personality[40]; // задает размер массива
int talents;
public: // необходимые
// методы
nifty();
nifty(const char * s);
friend ostream & operator<<
(ostream & os, const nifty & n);
}; // обратите внимание на завершающий
// знак "точка с запятой"
nifty::nifty()
{
    personality[0] = '\0';
    talents = 0;
}
nifty::nifty(const char * s)
{
    strcpy(personality, s);
    talents = 0;
}
ostream & operator<<(ostream & os,
const nifty & n)
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

Или другой вариант:

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // необязательный блок
    char * personality; // создает указатель
    int talents;
public: // необходимые
// методы
nifty();
nifty(const char * s);
nifty(const nifty & n);
~nifty() { delete personality; }
nifty & operator=(const nifty & n) const;
friend ostream & operator<<
(ostream & os, const nifty & n);
}; // обратите внимание на завершающий
// знак точка с запятой
nifty::nifty()
{
    personality = NULL;
    talents = 0;
}
nifty::nifty(const char * s)
{
```

```
    personality = new char [strlen(s) + 1];
    strcpy(personality, s);
    talents = 0;
}

ostream & operator<<
(ostream & os, const nifty & n)
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}

5.2. Golfer nancy; // стандартный конструктор
Golfer lulu(Little Lulu);
// Golfer(const char * name, int g)
Golfer roy(Roy Hobbs, 12);
// Golfer(const char * name, int g)
Golfer * par = new Golfer;
// стандартный конструктор
Golfer next = lulu;
// Golfer(const Golfer &g)
Golfer hazard = "Weed Thwacker";
// Golfer(const char * name, int g)
*par = nancy; // стандартная операция
// присваивания
nancy = "Nancy Putter";
// Golfer(const char * name, int g),
// а затем выполняется стандартная
// операция присваивания
```

ПРИМЕЧАНИЕ

Некоторые компиляторы дополнительно вызовут стандартную операцию присваивания для операторов #5 и #6.

- b. В классе должна быть определена стандартная операция присваивания, осуществляющая копирование данных, а не ссылок на них.

Глава 12

- Общедоступные элементы базового класса становятся общедоступными элементами производного класса. Защищенные элементы базового класса становятся защищенными элементами производного класса. Приватные элементы базового класса наследуются, но недоступны непосредственно. В ответе на вопрос 2 рассмотрены исключения для этих основных правил.
- Методы конструкторов не наследуются, деструкторы не наследуются, операция присваивания не наследуется, а также не наследуются дружественные элементы.
- Все еще можно использовать значение элемента, но уже невозможно ему что-либо присвоить.

```
ArrayDb tosanjose[4];
double n=tosanhose[2]; //OK
tisanjose[1]=68.8; //больше не действует
```

Выполнение метода осуществлялось бы немного медленнее, так как оператор возврата провоцирует копирование объекта.

4. Конструкторы вызываются в порядке извлечения, первым вызывается конструктор-предок. Деструкторы вызываются в обратном порядке.
5. Да, для каждого класса требуются свои собственные конструкторы. Если в производный класс не добавлены новые элементы, тело конструктора можно оставить пустым, но создать его необходимо.
6. Вызывается только метод производного класса. Оно замещает определение основного класса. Метод базового класса вызывается только в том случае, если производный класс не переопределяет метод. Тем не менее, любую функцию, подлежащую переопределению, следует объявить виртуальной.
7. В производном классе следует определить оператор присваивания, если конструкторы производного класса используют оператор `new` или `new []` для инициализации указателей, являющихся элементами того класса. Вообще, в производном классе следует определить операцию присваивания, если метод стандартного присваивания неприемлем для работы с объектами производного класса.
8. Да, можно присвоить адрес объекта производного класса указателю на объект базового класса. Присвоить адрес объекта базового класса указателю на производный класс (приведение вниз) можно только путем явного приведения типов, но использовать такой указатель небезопасно.
9. Да, можно присвоить объект производного класса объекту базового класса. Однако любые элементы данных, новые для производного типа, не передаются базовому классу. Программу будет использовать оператор присваивания базового класса. Присваивание в обратном порядке (базового объекта к произ-

водному) возможно только в том случае, если производный класс определяет операцию перехода, являющуюся конструктором, у которого единственный аргумент — это ссылка на базовый тип.

10. Так можно сделать, поскольку C++ в ссылке на базовый тип может ссылаться на любой тип, производный для базового типа.
11. При передаче объекта с помощью величины подключается конструктор копирования. Поскольку формальный аргумент — это объект базового типа, вызываемый конструктор класса принадлежит базовому классу. Конструктор копирования имеет в качестве своего аргумента ссылку на базовый тип, а эта ссылка может ссылаться на производный объект, передаваемый как аргумент. Как результат, появляется новый объект базового класса, элементы которого относятся к блоку базовых классов данного производного класса.
12. Передача объекта по ссылке вместо передачи по значению позволяет функции быть доступной для виртуальных функций. К тому же при передаче объекта по ссылке, а не по значению тратится меньше места и времени, особенно это заметно для больших объектов. Основное преимущество передачи по значению состоит в безопасности исходных данных, но этого же можно достичь, передавая ссылки как постоянные типа `const`.
13. Если `head()` — обычная функция, то `ph->head()` вызывает `Corporation::head()`. Если `head()` — виртуальная функция, то `ph->head()` вызывает `PublicCorporation::head()`.
14. Во-первых, ситуация не подходит под модель отношения *is-a*, поэтому метод общедоступного наследования неприменим. Во-вторых, определение `area()` в `House` скрывает вариант `Kitchen area()`, потому что два метода имеют разные сигнатуры.

Глава 13

1. См. табл. J.1.

Таблица J.1

class Bear	class PolarBear	Общедоступный, белый медведь — это разновидность медведя
class Kitchen	class Home	Приватный, дом включает кухню
class Person	class Programmer	Общедоступный, программист — это такой человек
class Person	class HorseAndJockey	Приватный, экипаж из лошади и жокея включает человека
class Person, class Automobile	class Driver	Person — общедоступный, потому что водитель — это человек, умеющий управлять автомобилем

```

2. Gloam::Gloam(int g, const char * s) : glip(g),
   fb(s) { }
Gloam::Gloam(int g, const Frabjous & f) :
   glip(g), fb(f) { }
// обратите внимание, что тут используется
// стандартный конструктор копирования
// Frabjous
void Gloam::tell()
{
   fb.tell();
   cout << glip << '\n';
}

```

```

3. Gloam::Gloam(int g, const char * s)
   : glip(g), Frabjous(s) { }
Gloam::Gloam(int g, const Frabjous & f)
   : glip(g), Frabjous(f) { }
// обратите внимание, что тут используется
// стандартный конструктор копирования
// Frabjous
void Gloam::tell()
{
   Frabjous::tell();
   cout << glip << '\n';
}

```

```

4. class Stack<Worker *>
{
private:
enum { MAX = 10 } ; // константа,
                     // специфичная для класса
Worker * items[MAX]; // содержит
                     // элементы стека
int top; // номер верхнего элемента стека
public:
   Stack();
   Boolean isempty();
   Boolean isfull();
   Boolean push(const Worker * & item);
   // добавляет в стек item
   Boolean pop(Worker * & item);
   // выталкивает item из стека
};

```

```

5. ArrayTP<String> sa;
StackTP< ArrayTP<double> > stck_arr_db;
ArrayTp< StackTP<Worker *> > arr_stk_wpr;

```

6. Если общий предок используется двумя строками, задающими наследование класса, класс получит две копии элементов предка. Создание класса-предка как виртуального базового класса для его непосредственных наследников решает эту проблему.

Глава 14

1. а. Объявление дружественной конструкции должно записываться в следующем виде:

```
friend class clasp;
```

б. Для этого необходимо опережающее объявление, тогда компилятор может обработать void snip(muff &):

```

class muff; // опережающее объявление
class cuff {
public:
   void snip(muff &) { ... }
...
};

class muff {
   friend void cuff::snip(muff &);
...
};

```

с. Во-первых, объявление класса cuff должно предшествовать объявлению класса muff, тогда компилятор сможет понять термин cuff::snip(). Во-вторых, компилятору необходимо опережающее объявление muff, тогда он сможет обработать snip(muff &).

```

class muff; // опережающее объявление
class cuff {
public:
   void snip(muff &) { ... }
...
};

class muff {
   friend void cuff::snip(muff &);
...
};

```

2. Нет. Чтобы А имел дружественную конструкцию, которая является функцией-элементом для В, объявление В должно предшествовать объявлению А. Опережающего определения недостаточно, поскольку при этом А будет знать, что В является классом, но не будет знать об именах элементов класса. Аналогично, если В имеет дружественную конструкцию, которая является функцией-элементом для А, завершенное объявление А должно предшествовать объявлению В. Два этих требования взаимно исключают друг друга.

3. Единственный способ доступа к классу осуществляется через общедоступный интерфейс, это приводит к тому, что единственное разрешенное действие над объектом Sauce заключается в вызове конструктора для его создания. Другие элементы (soy и sugar) являются частными по умолчанию.

4. Предположим, что функция f1() вызывает функцию f2(). Оператор возврата к f2() стимулирует программу к выполнению оператора, следующего за вызовом функции f2() в функции f1(). Оператор throw заставляет программу возвращаться к текущей последовательности вызовов функций до тех пор, пока не найдется блок try, прямо или косвенно содержащий вызов f2(). Такой блок может находиться в f1() или в функции, вызывающей f1() или подобный объект. Оттуда выполнение программы переходит к следующему подходящему блоку catch, а не к первому оператору после вызова функции.

5. Следует разместить блоки `catch` в порядке от старшего производного класса к младшему.
6. Для примера #1 условие `if` принимает истинное значение, если `pg` указывает на объект `Superb` или на объект любого класса, происходящий от `Superb`. В частности, условие также верно, если `pg` указывает на объект `Magnificent`. В примере #2 условие `if` определяется только для объекта `Superb`, но не для объектов, производных от `Superb`.
7. Операция `dynamic_cast` позволяет выполнить верхнее приведение типов в классовой иерархии, тогда как операция `static_cast` позволяет осуществить как нисходящее, так и восходящее приведение типов. Операция `static_cast` позволяет также осуществить преобразования из нечисловых типов в целочисленные типы и наоборот.

Глава 15

```
1. #include <string>
using namespace std;
class RQ1
{
private:
    string st;           // строковый объект
public:
    RQ1() : st("") { }
    RQ1(const char * s) : st(s) { }
    ~RQ1() { };
// дополнительные сведения
};
```

Конструктор явного копирования, деструктор и операция присваивания более не нужны, так как для строкового объекта предусмотрен специальный механизм управления памятью.

2. Можно присвоить один строковый объект другому. Для строкового объекта предусмотрены собственные механизмы управления памятью, поэтому нет необходимости беспокоиться о том, что строка может превысить отведенный для нее размер.

```
3. #include <string>
#include <cctype>
using namespace std;
void ToUpper(string & str)
{
    for (int i = 0; i < str.size(); i++)
        str[i] = toupper(str[i]);
}

4. auto_ptr<int> pia= new int[20];
    // неверно, используйте new вместо new[]
auto_ptr<str>(new string);
    // неверно, нет имени указателя
int rigue = 7;
auto_ptr<int>(&rigue);
    // неверно, new не распределяет память
auto_ptr<dbl>(new double);
    // неверно, пропущено <double>
```

5. Тип LIFO для стека означает, что необходимо убрать много карт, прежде чем будет найдена необходимая карта.
6. Набор сохранит только одну копию каждой величины, поэтому, например, пять отметок 5 сохранились бы как одна оценка 5.
7. Применение итераторов позволяет организовать перемещение по данным, которые организованы иначе, чем массивы, — скажем, как данные в двусвязном списке. При этом объекты используются с помощью интерфейса, напоминающего интерфейс указателей.
8. Подход с применением библиотеки STL позволяет использовать функции STL как с помощью обычных указателей на обычные массивы, так и с помощью итераторов на классы контейнеров STL, повышая, таким образом, степень общности.
9. Можно присвоить один векторный объект другому. Вектор самостоятельно управляет выделенной для него памятью, поэтому можно вставить элемент в вектор, и он автоматически изменит свой размер. С помощью метода `at()` можно автоматически осуществить и проверку пределов.
10. Две функции `sort()` и функция `random_shuffle()` требуют наличия итератора случайного доступа, тогда как объект `list` имеет только реверсивный итератор. Чтобы осуществить сортировку, можно воспользоваться функциями-элементами `sort()` класса шаблонов списка вместо основных предназначенных для этого функций, но функций-элементов, эквивалентных `random_shuffle()`, не существует. Тем не менее, можно скопировать список в вектор, перетасовать компоненты вектора и скопировать результат обратно в список. (Во время написания этой книги большинство компиляторов еще не применяли функцию-элемент `sort()`, которая использует в качестве аргумента объект `Compare`. Отметим также, что реализация класса списков Microsoft Visual C++ 5.0 имеет несколько ошибок, не допускающих осуществление предложенных преобразований.)

Глава 16

1. Файл `iostream` определяет классы, константы и манипуляторы, используемые для управления операциями ввода и вывода. Эти объекты управляют потоками и буферами с помощью дескрипторов ввода/вывода. Файл создает также стандартные объекты (`cin`, `cout`, `cerr` и `clog`, а также их эквиваленты — расширенные символы), используемые для манипуля-

ций стандартными потоками ввода и вывода, связанными с каждой программой.

2. Ввод с клавиатуры порождает серию символов. Ввод числа 121 порождает три символа, каждый из которых занимает 1-байтовую ячейку. Если же величина должна храниться как величина типа `int`, тогда эти три символа необходимо преобразовать в одну заданную двоичную величину, 121.

3. По умолчанию и стандартный поток вывода, и стандартный поток ошибок осуществляют вывод на стандартное устройство вывода, обычно на монитор. Если же операционная система перенаправляет вывод в файл, стандартный поток вывода связывается с файлом вместо монитора, но стандартные сообщения об ошибках продолжают выводиться на экран.

4. Класс `ostream` определяет вариант функции `operator<<()` для каждого базового типа C++. Компилятор интерпретирует выражение как

```
cout << spot
```

как в следующем случае:

```
cout.operator<<(spot)
```

Затем вызов этого метода можно привести в соответствие с прототипом функций, имеющей такой же аргумент.

5. Можно конкатенировать метод вывода, возвращающий тип `ostream &`. Чтобы возвратить тот объект, с его помощью вызывается метод. Затем каждый возвращаемый объект может последовательно вызывать следующие методы.

6. //rq16-6.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << "Enter an integer: ";
    int n;
    cin >> n;
    cout << setw(15) << "base ten"
        << setw(15) << "base sixteen"
        << setw(15) << "base eight"
        << "\n";
    cout.setf(ios::showbase);
    // или cout << showbase;
    cout << setw(15) << n << hex
    << setw(15) << n << oct
    << setw(15) << n << "\n";

    return 0;
}
```

7. //rq15-7.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char name[20];
    float hourly;
    float hours;

    cout << "Enter your name: ";
    cin.get(name, 20).get();
    cout << "Enter your hourly wages: ";
    cin >> hourly;
    cout << "Enter number of hours worked: ";
    cin >> hours;

    cout.setf(ios::showpoint);
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::right, ios::adjustfield);
    // или cout << showpoint
    // << fixed << right;
    cout << "First format:\n";
    cout << setw(30) << name << ": $" 
        << setprecision(2) << setw(10)
        << hourly << ":" << setprecision(1)
        << setw(5) << hours << "\n";
    cout << "Second format:\n";
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(30) << name << ": $" 
        << setprecision(2)
        << setw(10) << hourly << ":" 
        << setprecision(1)
        << setw(5) << hours << "\n";

    return 0;
}
```

8. Будет выдан следующий результат:

```
ctl = 5; ct2 = 9
```

Первая часть программы игнорирует пробелы и символы новых строк; вторая часть не пропускает их. Обратите внимание, что вторая часть программы начинает считывание символа новой строки, следующего за первым символом `q`, и учитывает этот символ как часть общего итога.

9. Форма `ignore()` действует непредсказуемо, если длина вводимой строки превышает 80 символов. В этом случае она просто вводит только первые 80 символов.

Предметный указатель

Символы

* 96
++ 122
-- 122
<< 37, 521
= 41
|| 148

А

Абстрагирование 246
Автоматическая память 225
Алгоритм 508
Аргумент 34, 44
заданный по умолчанию 207
Арифметические операции 70
Ассоциативность 71

Б

Байт 54
Библиотека 27
RTTI 456
STL 594
символьных функций ctype 153
Бит 54
Буфер 517

В

Ввод 87
смешанный строчно-числовой 87
данных 534
текста 134
Ввод/вывод 516, 545, 593
перенаправление 520
Вектор 286, 476
Виртуальные методы 376
Виртуальный базовый класс 411
Вывод данных 37
Выражение 70, 126
сравнения 126

Д

Двоичный поиск 608
Деструктор 255, 264, 362, 372
Диапазон доступа класса 224, 250, 267
Динамическая память 306
Динамическое распределение памяти 236, 365
Динамическое связывание 358
Директива Using 37, 238
Дружественные элементы 431

З

Значащие цифры 67

И

Имена в C++ 236
Индекс 80
Инициализация 55

Инкапсуляция 249
Интегрированная среда разработки (IDE) 27, 29
Исключения 437
Bad_alloc 453
непредвиденное 453
Исходный код 26
Итератор 486
copy() 489
istream_iterator 489
ostream_iterator 489
ввода 486
вывода 486
двусторонний 487
другие 490
произвольного доступа 487
прямой 487

К

Квалификатор 65
const 65
Класс
Allocator 472
Auto_ptr 472
Customer 335
Deque 497
Ellipse 370
Exception 451
Istream 43
List 497
Multimap 503
Ostream 43, 521, 525
Priority_queue 499
Push_back() 496
Push_front() 496
Queue 330, 499
Stack 500
Stock 258
String 307, 320, 465, 583
конструкторы 466
Vector 26, 476, 496
абстрактный базовый 370
базовый 346
виртуальный базовый 411
включающий элементы объектов 381
вложенный 332, 432
диапазон доступа 433
дружественный 424
наследование 345
производный 346
шаблоны классов 394

Классы памяти 227, 235

статические 228

Ключевые слова языка C++ 48, 570

Командная строка 548

Комментарий 35

Компилятор 22

Компиляция 28, 221

раздельная 221

Компоновка 28

Компьютер 40
Конкатенация строк 83
Константы 56, 59, 62
с плавающей точкой 69
символические 56
символьные 62
целочисленные 59
Конструктор 255, 262, 362, 372, 585
копирования 315, 372, 586
стандартный 585
строковый 585
Контейнер 492, 493, 500
ассоциативный 500
Куча 611

Л

Лексема 39
Логические выражения 148
Логические операции 145

М

Малые целые числа 60
Массив 59, 79, 103, 140, 172
Двумерный 140
Динамический 101
Имя массива 106
Индекс 80
Метод
Begin() 485
Copy() 592
End() 485
Erase() 478, 592
Find() 588
Find_first_not_of() 589
Find_first_of() 589
Find_last_not_of() 590
Find_last_of() 589
Get(char &) 541
Get(void) 541
Insert() 591
Read() 555
Replace() 592
Rfind() 589
Swap() 592
Width() 526
Write() 555
виртуальный 376
включения 381
встроенный 252
приватный и защищенный наследования
381
сравнения 590

Множественное наследование 409

Моделирование 338

Модификатор 230, 591
static 230
const 203

Н

Наследование 363, 390, 446
приватное 390

Невхватка памяти 100

О

Область имен 36, 37
Обобщенное программирование 483

Объединение 92
анонимное 93

Объект 23, 37, 245, 520
cin 42, 534
cout 37, 520

Объектно-ориентированное программирование 249

Операнд 70

Оператор 34, 40, 55, 62, 99, 122, 145, 453

? : 154

break 158

continue 158

if 145

if else 146

If else if else 147

new 99, 101, 325, 453

switch 155

декремента (--) 122

"запятая" 124

инкремента (++) 122

комбинированные операторы присваивания 123

объявления 40

приведения типов 462

принадлежности 62

присваивания 41, 375

составной 123

Операция 55, 70, 96, 145, 148, 521

* (косвенное значение) 96

<< 521

арифметическая 70

деления 71

логическая 145

И (&&) 149

ИЛИ (||) 148

НЕ (!) 151

приоритет операций 71

над наборами 609

Операционная система 22

Освобождение памяти 100

Отношение 347, 374, 381

has-a ("содержит объект") 348, 381

is-a ("является объектом") 347, 374, 382

uses-a 348

Очередь 329

П

Память 95, 99, 111, 225, 236, 306

автоматическая 111, 225

динамическая 236, 306

невхватка памяти 100

освобождение памяти 100

распределение 99

свободная 95, 111

статическая 111

Параметр 34, 44

Перегрузка 209
операции 38, 275
<< 282, 328

Переменная 40, 52, 112, 198

Boolean 65

автоматическая 112

внешняя 228, 232

временная 203

глобальная 230

именование 53

локальная 230

простая 52

ссылочная 198

типа register 227

Переносимость 25

Перестановки 612

Перечисления 93

Подпрограмма 47

Поиск

двоичный 608

Полиморфизм 209

Последовательность 495

Поток 517

Преобразование типов данных 73

Пропроцессор C++ 36

Приватное наследование 390

Приведение типов 75

Приоритет операций 71, 575

Присваивание 363

смешанное 364

Программирование 21, 245, 483

обобщенное 24, 483

объектно-ориентированное 21, 23, 245, 249

процедурное 21, 23, 245

сверху вниз 23

снизу вверх 24

структурное 23

Произвольный доступ к файлам 557

Пространство имен 236

Прототип 167

Процедура 47

Псевдоним типа 133

Р

Раздельная компиляция 221

Разрядные поля 92

Распределение памяти

динамическое 236, 365

Расширение типов 558

Рекурсия 188

С

Свободная память 95

Связывание 101, 227, 231, 235, 358

динамическое 101, 105, 358

статическое 101, 105, 357

языковое 235

Сигнатура 210

Символ

-заполнитель 527

новой строки (\n) 38

сигнальной метки 134

Символические константы 56

Система счисления 568

восьмеричная 568

двоичная 569

шестнадцатиричная 568

Слияние 609

Скрытие данных 249

Сортировка 481, 606

Специализация 215, 218, 396, 408

явная 215

Спецификатор const 179

Спецификаторы классов памяти 233

Сравнение строк 128

Среда разработки 27

Ссылка 200, 356

Стандарт 25

ANSI 31

ISO/ANSI 616

Стандартная библиотека шаблонов (STL) 465, 475

Стандартные конструкторы 585

Статический класс памяти 228

Статическое связывание 357

Стек 270, 443

Стону 49

Строка 37, 82

конкатенация строк 83

Структура 88, 183, 250, 332

вложенная 332

динамическая 109

дружественная 280

Т

Таблица 361

Таблица кодов ASCII 571

Тип данных 54, 247, 268

Boolean 65

абстрактный 268

без знака (unsigned) 57

основной (char, short, int и long) 54

преобразование типов данных 73

с плавающей точкой (float, double и long double) 67

символьный (char) 62

целочисленный (short, int и long) 54

У

Указатель 95, 98, 103, 106, 173, 326, 356, 521

this 263

арифметика указателей 105

на функции 189

объявление указателей 105

присвоение значений указателям 105

разыменование указателей 105

Управление доступом 355

Управляющие последовательности языка C++ 63

Ф

Файл 36, 518

iomanip 533

iostream 36, 518

Флаг 528

Форматирование 560

внутреннее 560

Функции 428
 дружественные 428
 Функции-адаптеры 507
 Функциональные объекты (функции) 504
 адаптируемые 507
 предопределенные 506
 Функция
 Accumulate() 613
 Adjacent_difference() 614
 Adjacent_find() 600
 Binary_search() 609
 Cin.get() 540
 Cin.get(ch) 540
 Cin.get(char) 135
 Copy() 489, 601
 Copy_backward() 601
 Count_if() 600
 Count() 600
 Cout.put() 62
 Equal() 600
 Equal_range() 609
 Exit() 252
 Fill() 604
 Fill_n() 604
 Find() 480, 599
 Find_ar() 485
 Find_end() 599
 Find_first_of() 600
 Find_if() 599
 Find_ll() 485
 For_each() 599
 Gcount() 543
 Generator() 604
 Generator_n() 604
 Get() 540
 Get(char *, int, char) 541
 Getline() 85, 540
 Getline(char *, int, char) 542
 Ignore() 540
 Includes() 610
 Inner_product() 614
 Inplace_merge() 609
 Is_open() 550, 552
 isspace() 538
 Iter_swap() 603
 Lexicographical_compare() 612
 Lower_bound() 608
 Main() 33
 Make_heap() 611
 Max() 612
 Max_element() 612
 Merge() 609
 Min() 612
 Min_element() 612
 Next_permutation() 613
 Nth_element() 608
 Operator>() 320
 Partial_sort() 608
 Partial_sort_copy() 608
 Partial_sum() 614
 Partition() 605
 Peek() 543
 Pop_heap() 611
 Pow() 46

Precision() 527
 Previous_permutation() 613
 Printf() 42
 Push_heap() 611
 Putback() 543
 Random_shuffle() 605
 Read() 543
 Remove() 604
 Remove_copy() 604
 Remove_copy_if() 604
 Remove_if() 604
 Replace_if() 603
 Replace_copy() 603
 Replace_copy_if() 603
 Reverse_copy() 605
 Reverse() 605
 Rotate() 605
 Rotate_copy() 605
 Search() 601
 Search_n() 601
 Seekg() 558
 Set_difference() 610
 Set_intersection() 610
 Set_symmetric_difference() 611
 Set_union() 610
 Setf() 528
 Sort() 480, 606
 Sqrt() 45
 Stable_partition() 606
 Stable_sort() 608
 Strcmp() 320
 Swap() 603
 Swap_ranges() 603
 Transform() 603
 Unique() 605
 Unique_copy() 605
 Upper_bound() 608
 WorseThan() 481
 аргументы 169
 встроенная 196
 вызов 166
 обзор 164
 определяемая пользователем 47
 прототипирование 166
 со многими аргументами 221
 сравнения 590
 Функция-элемент 62, 250, 309, 357
 виртуальная 357, 362, 371
 неявная 315
 свойства 377
 статическая 309

Ц

Целочисленная константа 59
 Целые числа 54, 60
 Цикл 115
 Do while 135
 For 115
 While 129
 вложенный 140
 шаг цикла 117
 Цифры
 значащие 67

Ч

Числа 54, 100
 с плавающей точкой 66
 целые 54

Ш

Шаблон 212, 412, 431
 класса 394, 412
 перегруженный 214

Э

Экземпляр 396
 шаблона 218
 Элемент
 статических данных 308
 статического класса 307
 дружественный 431

Я

Язык 22
 Ассемблер 22
 машинный 26
 С 22
 С++ 24

Иностранные термины

ADT – abstract data types 271
 ANSI 31
 FIFO 329
 IDE 29
 LIFO 270, 329
 RTTI 456
 STL – Standard Template Library 465, 594