

В. В. Лантев

C++

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



УЧЕБНОЕ / ПОСОБИЕ

В. В. Лантев

C++

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



Издательская программа
300 лучших учебников для высшей школы
осуществляется при поддержке Министерства образования и науки РФ

ПИТЕР®

**Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск**

2008

ББК 32.973-018.1я7
УДК 004.43(075)
Л24

Лаптев В. В.

Л24 С++. Объектно-ориентированное программирование: Учебное пособие. — СПб.: Питер, 2008. — 464 с.: ил. — (Серия «Учебное пособие»).

ISBN 978-5-91180-200-4

Учебное пособие для студентов посвящено объектно-ориентированному программированию на языке C++.

Описываются объектно-ориентированные конструкции языка, библиотека STL и их практическое применение. На примерах разработки контейнерных классов и итераторов излагаются принципы организации библиотеки STL. В связи с изложением реализации контейнеров много внимания уделено и управлению памятью. Подробно описана библиотека ввода-вывода, причем как процедурная, так и объектно-ориентированная. Дано описание ряда ключевых шаблонов программирования, связанных с конкретными конструкциями C++. В качестве примера рассмотрено приложение, разработанное с использованием полученных знаний и WinAPI. В конце каждой главы — краткое резюме, контрольные вопросы и набор заданий.

Для студентов и преподавателей вузов, читателей, знакомых с основами C++, желающих стать профессиональными программистами.

ББК 32.973-018.1я7
УДК 004.43(075)

Все права защищены. Ничая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-91180-200-4

© ООО «Питер Пресс», 2008

Краткое содержание

Введение	11
Глава 1. Классы и объекты	16
Глава 2. Конструкторы	45
Глава 3. Перегрузка операций	65
Глава 4. Массивы и классы	91
Глава 5. Динамическая память в C++	117
Глава 6. Контейнеры	146
Глава 7. Исключения	171
Глава 8. Наследование	207
Глава 9. Виртуальные функции	230
Глава 10. Множественное наследование и RTTI	253
Глава 11. Шаблоны классов	278
Глава 12. Шаблоны функций	307
Глава 13. Программы и модули	336
Глава 14. Библиотека ввода-вывода	373
Приложение. Строки в C++	441
Литература	453

Содержание

Введение	11
Структура книги	13
Использованные программные продукты	14
Благодарности	15
От издательства	15
Глава 1. Классы и объекты	16
Зачем нужны классы?	17
Определение класса	23
Доступ к элементам класса	24
Определение методов класса	26
Указатель this	29
Перегрузка методов	30
Размеры объектов класса	33
Использование класса	37
Резюме	41
Контрольные вопросы	42
Упражнения	43
Глава 2. Конструкторы	45
Определение конструктора	45
Конструкторы и объекты	49
Конструкторы и параметры	52
Деструктор	54
Конструкторы и константы	55
Константы в классе	57
Инициализация константных полей	58
Резюме	61
Контрольные вопросы	61
Упражнения	62
Глава 3. Перегрузка операций	65
Перегрузка операций внешними функциями	68
Перегрузка операций методами класса	70
«Подводные камни» перегрузки операций	76

Функции-друзья класса	78
Преобразование типов	81
Преобразование типов и конструкторы	82
Функции-операции преобразования	84
Запрет неявных преобразований	86
Классы-оболочки встроенных типов	86
Резюме	87
Контрольные вопросы	88
Упражнения	89
Глава 4. Массивы и классы	91
Поля-массивы в классе	91
Реализация простого класса строк	94
Статические поля-массивы	106
Статические элементы класса	109
Резюме	114
Контрольные вопросы	114
Упражнения	115
Глава 5. Динамическая память в C++	117
Память и объекты	117
Управление динамической памятью	118
Двухмерный динамический массив	120
POD-типы	121
NonPOD-типы	122
Еще одна форма операции new	124
Размеры динамических объектов	124
«Умный» массив	126
Конструкторы	128
Деструктор	131
Операция индексирования	132
Копирование и присваивание	133
Реализация методов	138
Использование «умного» массива	140
Резюме	142
Контрольные вопросы	143
Упражнения	144
Глава 6. Контейнеры	146
Характеристики контейнеров	146
Доступ к элементам контейнера	147
Операции с контейнером	149
Реализация контейнеров	151
Последовательный контейнер	152
Вложенные классы	153
Итератор для последовательного контейнера	155
Реализация контейнера-дека	157
Стек	163
Проблема универсальности	166
Резюме	168
Контрольные вопросы	168
Упражнения	169

Глава 7. Исключения	171
Принципы обработки исключений	172
Генерация исключений	172
Перехват и обработка исключений	173
Спецификация исключений	177
Применение исключений	178
Передача информации в блок обработки	181
Классы и исключения	182
Исключения в списке инициализации конструктора	187
Исключения и деструкторы	189
Стандартные исключения	193
Подмена функций стандартного завершения	196
Нестандартные исключения	198
Резюме	203
Контрольные вопросы	203
Упражнения	205
Глава 8. Наследование	207
Простое наследование	209
Простое открытое наследование	210
Конструкторы, деструкторы и наследование	212
Поля и методы при наследовании	215
Вложенные классы и наследование	217
Принцип подстановки	218
Применение открытого наследования	221
Закрытое наследование	223
Резюме	227
Контрольные вопросы	227
Упражнения	228
Глава 9. Виртуальные функции	230
Зачем нужны виртуальные функции	230
Определение виртуальных функций	233
Переопределение и перегрузка виртуальных функций	234
Размеры классов с виртуальными функциями	237
Виртуальные функции в конструкторах и деструкторах	238
Чистые виртуальные функции	239
Виртуальные деструкторы	242
Чистые виртуальные деструкторы	243
Однокоренная иерархия	245
Виртуализация внешних функций	247
Немного философии	249
Резюме	250
Контрольные вопросы	251
Упражнения	252
Глава 10. Множественное наследование и RTTI	253
Множественное наследование	254
Неоднозначность	256
Виртуальное наследование	257
Принцип доминирования	259
Финальный класс	260
Размеры классов при множественном наследовании	261
RTTI	262

Мультиметоды	268
Использование RTTI	269
Использование только виртуальных функций	271
Резюме	276
Контрольные вопросы	276
Упражнения	277
Глава 11. Шаблоны классов	278
Первое знакомство	279
Определение шаблона класса	283
Внешнее определение методов	284
Параметры шаблона — не типы	286
Инициализация нулем	288
Параметры шаблона по умолчанию	289
Специализация	290
Шаблоны и... шаблоны	293
Поле-шаблон	293
Параметр-шаблон	294
Метод-шаблон	296
Шаблоны и наследование	299
Шаблоны и дружелюбность	302
Резюме	304
Контрольные вопросы	304
Упражнения	305
Глава 12. Шаблоны функций	307
Определение шаблона функции	307
Параметры по умолчанию	311
Параметры шаблона — не типы	312
Перегрузка шаблонов функций	312
Специализация шаблонов функций	313
Реализация обобщенных алгоритмов	314
Указатели на функции и указатели на методы	318
Понятие функтора	322
Адаптеры функторов	326
Адаптер для указателя на метод	330
Резюме	333
Контрольные вопросы	334
Упражнения	334
Глава 13. Программы и модули	336
Сборка исходных текстов	337
«Стражи» включения	339
Системные включаемые файлы	342
Отделение интерфейса от реализации	343
Шаблоны и модули	346
Модель включения	348
Явное инстанцирование	349
Межмодульное взаимодействие	350
Межмодульные переменные и функции	351
Локализация имен в модуле	352
Альтернативные спецификации компоновки	355
Инициализация глобальных объектов	355
Статические элементы в шаблонах	359

Пространства имен	363
Именованные пространства имен	364
Неименованные пространства имен	369
Резюме	370
Контрольные вопросы	371
Упражнения	372
Глава 14. Библиотека ввода-вывода	373
Иерархия классов	374
Принципы организации потоков	375
Стандартные потоки	377
Вывод элементарных типов и строк	378
Ввод элементарных типов	380
Ввод строк	382
Состояния потока	384
Состояние потока и логические условия	386
Состояние потока и исключения	388
Работа с файлами	389
Текстовые файлы	392
Пример обработки текстовых файлов	394
Режимы открытия потоков (файлов)	398
Двоичные файлы	399
Ввод-вывод объектов в двоичный файл	403
Сериализация	404
Форматирование ввода-вывода	407
Флаги форматирования	408
Методы форматирования	409
Манипуляторы	410
Форматированный ввод	412
Написание собственных манипуляторов	413
Применение средств форматирования	415
Строковые потоки	417
Перегрузка операций ввода-вывода	420
Произвольный доступ	425
Буферизация	428
Широкие потоки	429
Ввод и вывод широких символов	430
Кириллица и локальный контекст	431
Широкие строковые потоки	433
Широкие файловые потоки	434
Резюме	436
Контрольные вопросы	437
Упражнения	439
Приложение. Строки в C++	441
Обработка строк в стиле C	441
Функции для работы с массивами типа <code>char[]</code>	443
Функции для работы с массивами типа <code>wchar_t[]</code>	446
Строки в стиле C++	447
Литература	453

Введение

Многие вещи нам непонятны не потому, что наши понятия слабы; но потому, что сии вещи не входят в круг наших понятий.

*Козьма Прутков,
Плоды раздумья, мысль 66*

Мое первое знакомство с объектно-ориентированным программированием на C++ состоялось в 1991 году, когда я прочитал первую книгу Б. Страуструпа «Язык программирования C++», выпущенную в русском переводе издательством «Радио и связь». Хотя к тому времени я был уже опытным программистом с 15-летним стажем разработки разнообразнейших программ, я почти ничего не понял. И это при том, что с языком С я уже был знаком! И дело совсем не в отсутствии упорства или недостатке сообразительности — на западе эту книгу называли «руководством для экспертов».

Много времени спустя, когда я прочитал большое количество других книг по языку C++ и сам стал преподавать его в университете, я пришел к твердому убеждению, что начинать изучать объектно-ориентированное программирование по книгам Б. Страуструпа — все равно что посадить начинающего водителя за руль болида «Формулы-1». Книги мэтра предназначены для крепких профессионалов, уже немало «понюхавших» объектно-ориентированного «пороху».

Сегодняшние реалии требуют начинать изучение «объектно-ориентированности» как можно раньше. Однако опыт преподавания C++ начинающим программистам показал, что C++ не должен быть первым изучаемым языком программирования. Более того, прежде чем изучать объектно-ориентированную часть C++, начинающему требуется овладеть первичными навыками программирования на этом языке — у него не должно вызывать проблем написание функций на C++. Поэтому в тексте книги вы не найдете описания элементарных конструкций C++ — в первой же главе вводится концепция класса.

Совершенно очевидно, что начинающие программисты сначала должны усвоить простую техническую информацию: как определить класс, что такое поле и как определить метод, как создать простой тип данных. По мере усвоения простого

материала можно переходить к более сложным вопросам: перегрузка операций, наследование, виртуальные функции, шаблоны и т. д. На базе усвоенной технической информации можно детально изучать обобщенное и порождающее программирование, использование стандартной библиотеки шаблонов, паттерны проектирования, взаимодействие стандартного языка C++ с API-функциями операционной системы.

Однако сначала я попытался ответить на вопрос: а зачем вообще потребовалось «городить» объектно-ориентированный «огород»? Какими преимуществами обладает он по сравнению с обычным (процедурным) программированием? Аналогичные вопросы часто задают и начинающие программисты. Более того, обычно такие же вопросы возникают по поводу каждого нового понятия или конструкции. Формальное определение в таких случаях практически бесполезно — начинающий может прекрасно его знать, но не понимать. Поэтому изложение построено на таких примерах, которые естественным образом показывают необходимость той или иной объектно-ориентированной конструкции, того или иного основополагающего принципа. Например, вы не найдете в книге формального определения инкапсуляции, однако на нее обращается внимание практически в каждом примере.

Многие понятия, термины и конструкции упоминаются в тексте книги задолго до их подробного описания. Например, операции `new` и `delete` используются уже в главе 1, хотя подробности управления динамической памятью описаны только в главе 5. В главе 5 впервые рассказывается о спецификации исключений, при этом детали механизма исключений раскрываются в главе 7. Наследование рассматривается в главе 8, однако уже в главе 7 приводится иерархия классов исключений. Во-первых, чрезвычайно трудно, если вообще возможно, описать C++ без таких «ссылок вперед» — даже в стандарте [1] ссылки вперед встречаются на каждом шагу. Во-вторых, опыт обучения студентов показал, что такое «предварительное» упоминание или использование конструкций в простых ситуациях в дальнейшем существенно облегчает усвоение сложных технических деталей.

Стандартная библиотека шаблонов (STL) является неотъемлемой частью C++, поэтому каждый программист обязан знать ее и уметь использовать. Однако я убежден, что квалифицированный программист должен хорошо понимать принципы, положенные в основу STL, и прекрасно разбираться в ее «внутренностях» — настолько хорошо, чтобы самостоятельно реализовать динамические контейнеры и итераторы, функторы и обобщенные алгоритмы. Поэтому реализации различного вида динамических контейнеров довольно часто используются в качестве примеров (см. главы 5 и 6). Итераторы упоминаются уже в главе 4 и вводятся в главе 6 как объекты, обеспечивающие последовательный доступ к элементам контейнера. Обобщенные алгоритмы и функторы рассматриваются в главе 12. Практически каждое реализованное решение сравнивается с решениями, принятыми при разработке STL. Это в дальнейшем существенно облегчает изучение самой библиотеки.

И наконец, в настоящее время квалифицированный программист просто обязан ориентироваться в стандарте C++ [1]. Это — не самая простая задача даже для опытного программиста. Поэтому при написании книги я не только постоянно сверялся со стандартом (использовалась вторая редакция от 15 октября 2003 года),

но и приводил ссылки на соответствующие разделы и пункты стандарта. Например, функции форматного ввода описаны в стандарте в пункте 27.6.1.2 — ссылка в книге дается как (см. п. п. 27.6.1.2 в [1]). Если некоторая тема описана в нескольких разных местах стандарта (например, исключения), то ссылки на пункты перечисляются через запятую, например (см. п. п. 18.6, 19.1 в [1]). Если в некотором разделе стандарта много нумерованных пунктов, то ссылка может быть уточнена номером пункта, например (см. п. п. 14.1/2 в [1]) — в этом пункте описывается использование слова `typename` вместо слова `class` при объявлении параметров шаблона.

Структура книги

Начиная писать эту книгу, я рассчитывал уложиться в 500 страниц. Однако работа оказалась настолько увлекательной, что я писал и писал. Материала накопилось значительно больше, чем требовалось, поэтому кое-чем пришлось пожертвовать. В книгу не попали темы, связанные с управлением динамической памятью: интеллектуальные указатели, механизм `malloc()/free()`, перегрузка операций `new` и `delete`. Пришлось отказаться и от описания процедурной библиотеки ввода-вывода `<cstdio>`. Не попал в книгу материал по обобщенному и порождающему программированию. К большому моему сожалению, не получилось «втиснуть» в рамки книги достаточно полное описание стандартной библиотеки шаблонов STL, а краткий «конспект» мало полезен. Библиотека шаблонов — слишком обширная тема, чтобы ее можно было изложить в паре-тройке десятков страниц¹. Однако можно обнаружить довольно частые упоминания о ней при реализации контейнеров, итераторов и алгоритмов: принятые в книге решения сравниваются с решениями STL.

Книга состоит из введения, 14 глав и приложения. Так как она задумана как учебное пособие по объектно-ориентированной части C++, в конце каждой главы приведены контрольные вопросы и небольшой набор практических упражнений по теме главы. Вопросы могут использоваться для подготовки к экзамену по C++.

В главе 1 объясняется, зачем необходимы классы, описываются синтаксис и семантика конструкции класса. В качестве примера приведена реализация класса денег. Затрагиваются проблемы выравнивания и размеров классов.

Глава 2 посвящена реализации конструкторов. Изучаются также вопросы использования констант и константных полей в классах.

В главе 3 излагаются проблемы перегрузки операций. Объясняется, зачем необходимы дружественные функции. Рассматриваются функции преобразования типов и роль конструкторов в преобразовании типов.

Глава 4 посвящена вопросам «взаимоотношений» полей-массивов и классов. Реализуется довольно развитый класс строк, аналогичных строкам в системе Turbo Pascal. Изучаются статические элементы классов (в том числе статические массивы).

¹ См., например, [29,30,31,32,33,35].

В главе 5 рассматриваются простейшие вопросы управления динамической памятью. Реализуется динамический «умный» массив, на примере которого объясняются особенности конструктора копирования, операции присваивания и деструктора.

Глава 6 является введением в тему динамических контейнеров. Подробно рассматриваются операции последовательного доступа к элементам контейнера и реализация их посредством итераторов. Сравняются списковые реализации дека и стека. Изучаются проблемы универсализации контейнеров.

Глава 7 содержит материал по исключениям. Довольно подробно описана «раскрутка» стека. Рассматривается иерархия стандартных исключений. Для означения и сравнения рассматривается механизм структурных исключений, реализованный в Windows.

В главе 8 достаточно подробно описывается простое одиночное наследование — как открытое, так и закрытое. Детально объясняется принцип подстановки.

Виртуальные функции, в том числе и виртуальные деструкторы, изучаются в главе 9.

Глава 10 включает материал о множественном наследовании. Обсуждаются проблемы, возникающие при использовании этого механизма. Рассматривается также механизм RTTI и его применение для динамической идентификации типов во время выполнения программы. Описываются различные способы реализации мультиметодов.

В главе 11 рассматриваются шаблоны классов. Описывается шаблонная реализация некоторых контейнеров. Подробно раскрываются различные варианты использования параметров шаблона и механизм специализации шаблонов.

В главе 12 изучаются шаблоны функций. Разрабатывается обобщенный алгоритм; реализуются функторы и адаптеры функторов, аналогичные стандартным, реализованным в STL.

Глава 13 посвящена вопросам разделения программы на модули. Описывается принцип ODR. На примерах ранее реализованных классов выполняется разделение их на интерфейс и реализацию. Объясняется механизм функционирования стражей включения. Описываются пространства имен. Рассматривается взаимодействие различных элементов программы, объявленных в разных модулях.

Глава 14, посвященная вводу-выводу, — самая большая в книге. Достаточно подробно описываются стандартные средства форматирования, работа с файлами, строковые потоки. Обсуждаются проблемы перегрузки операций ввода-вывода для нового класса вместе с реализацией пользовательских манипуляторов. Рассматриваются широкие потоки.

Использованные программные продукты

Практически все примеры были реально проверены в лицензионной системе Microsoft Visual C++.NET 2003. Большая часть примеров была проверена также в столь любимой российскими программистами системе Borland C++ Builder 6, которая была выбрана как одна из популярнейших в России. К тому же у всех

систем компании Borland прекрасная система встроенной помощи, тогда как Visual C++ требует отдельной установки MSDN.

Операционной системой, естественно, выбрана Windows.

Благодарности

В первую очередь хотел бы выразить благодарность сотрудникам издательства «Питер», без самоотверженного труда которых эта книга просто не могла появиться на свет. Особенную благодарность хочется выразить Анатолию Николаевичу Адаменко за помощь в подготовке книги к изданию.

Большое спасибо Илье Трубу, работа с которым над его книгой [16] показала еще одну сторону применения C++ и способствовала становлению моих взглядов на методику обучения объектно-ориентированному программированию.

Не могу не поблагодарить участников форума RSDN, в общении с которыми я провел много часов. Особая благодарность — признанным знатокам C++: Павлу Кузнецову, Николаю Меркину (Кодт), Валерию Белявцеву (Bell) и Андрею Тарасевичу, чьи подробнейшие разъяснения не раз помогали разобраться в особенностях и нюансах конструкций C++.

Спасибо моим студентам, работа с которыми натолкнула меня на мысль о необходимости написать эту книгу.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу <http://www.piter.com/download>.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства <http://www.piter.com>.

Глава 1

Классы и объекты

Принцип *divide et impera* (разделяй и властвуй) используется в программировании с самого начала — библиотеки стандартных подпрограмм появились «на второй день творения». Процедуры и функции существуют в том или ином виде во всех языках программирования. Сейчас говорят, что язык программирования, включающий в себя средства определения подпрограмм, поддерживает *парадигму* процедурного программирования. С одной стороны, конструкция подпрограммы позволяет нам применять принцип «разделяй и властвуй», разбивая большую программу на функционально независимые части. С другой стороны, подпрограммы с самого начала служили средством расширения языка программирования. Если в языке не хватало той или иной функциональности, создавалась библиотека подпрограмм, эту функциональность обеспечивающая. Это очень хорошо видно на примере Фортрана, для которого было создано огромное количество библиотек.

C++ тоже поддерживает процедурное программирование — традиционную технику написания программ. Функция — это одно из фундаментальнейших средств языка. Более того, от C унаследована обширная библиотека стандартных подпрограмм. Но C++ поддерживает и другие парадигмы: объектно-ориентированное и обобщенное программирование. Это означает, что C++ предоставляет удобные средства и механизмы поддержки этих стилей программирования. Однако прежде, чем изучать эти средства, зададимся простым вопросом: а почему недостаточно процедурного программирования? В программировании, как и вообще в жизни, ничего не происходит просто так. Если объектно-ориентированное программирование появилось, «значит, это кому-нибудь нужно»? Первый ответ на этот вопрос лежит на поверхности: так же, как подпрограммы расширяют функциональные возможности языка, классы и объекты позволяют наращивать множество типов данных, которыми может оперировать программист. Чтобы получить более подробные ответы, углубимся в изучение объектно-ориентированной части C++.

Зачем нужны классы?

Предположим, нам нужно написать программу, в которой требуется оперировать денежными суммами (например, в рублях и копейках). Программисты обычно не любят несколько раз писать одно и то же, и я не исключение. Поэтому хотелось бы запрограммировать такую обработку один раз в максимально общем виде и потом пользоваться готовыми средствами.

Любой тип данных характеризуется двумя составляющими: множеством значений и операциями, которые с этими значениями разрешается выполнять. Например, сами по себе числа не представляют интереса — нужно иметь возможность оперировать ими: складывать, вычитать, вычислять квадратный корень, и т. д.

Очевидно, что деньги представляют собой новый тип данных, поэтому мы должны суметь каким-то способом обеспечить некоторое множество значений и реализовать необходимые операции с этими значениями. Разберемся, как делается это при процедурном подходе.

Так как в C++ денежные суммы не являются встроенным типом данных, нам придется моделировать деньги с помощью других типов. Денежные суммы удобно представлять целыми числами, так как все вычисления можно производить в копейках, явно отделяя рубли от копеек только при вводе-выводе.

Минимальный набор операций с денежными суммами нам тоже хорошо известен: деньги можно складывать и вычитать, умножать на число, делить на число, делить деньги на деньги. Обратите внимание, что деньги на деньги умножать нельзя, хотя делить можно (например, чтобы узнать, во сколько раз одна сумма больше другой). Так как деление нацело без остатка практически невероятно, для правильного округления копеек будем использовать младшую цифру целого числа, представляющего собой денежную сумму. Таким образом, мы мысленно зафиксировали десятичную точку, отделив три знака для дробной части. Такое представление называется представлением с *фиксированной точкой*.

Нам нужны достаточно большие целые числа, чтобы можно было оперировать миллиардами и триллионами рублей. Это означает, что наши целые числа должны содержать не менее 18 десятичных цифр: 15 до точки и 3 — после. Лет 40 назад программист просто использовал подходящий встроенный целый тип данных — целые числа в компьютерах тех лет были достаточно «большими». Тогда большинство операций являются встроенными — писать придется только функции ввода и вывода, да еще функцию преобразования денежной суммы в строку.

Тот же подход можно применить и в C++. Однако в наш век персональных компьютеров на базе 32-разрядных микропроцессоров Pentium ни тип `int`, ни даже тип `long` не обеспечивают нам требуемые 18 цифр. Поэтому придется использовать нестандартный тип `__int64`¹. Это обеспечит нам диапазон целых от -2^{63} ($-9223\ 372\ 036\ 854\ 775\ 808$) до $+2^{63} - 1$ ($+9223\ 372\ 036\ 854\ 775\ 807$). С учетом положения фиксированной точки получаем диапазон денежных сумм от

¹ Этот тип реализован и в системах Visual C++, и в системах фирмы Borland.

–9223 372 036 854 775 руб. 80,8 коп. до +9223 372 036 854 775 руб. 80,7 коп. Вычислительные операции с такими числами уже реализованы в системе, нам остается написать только функции ввода-вывода.

Однако мы не будем заниматься реализацией нашего типа данных таким способом. Моделирование денежных сумм встроенным типом целых обладает рядом очень серьезных недостатков. И дело не только в том, что тип `__int64` не является стандартным — это легко исправить, заменив его стандартным типом `long double`. Самым существенным является то, что числа — это все-таки не деньги. Встроенные операции *не являются* операциями с денежными суммами, поэтому программисту очень легко совершить случайную ошибку, смешав в одном выражении деньги с другими числами, которые деньгами не являются. Компилятор, естественно, не выдаст сообщения об ошибке, даже если программист умножит деньги на деньги, что является бессмысленной операцией.

До некоторой степени опасность ошибок можно уменьшить с помощью оператора `typedef`, объявив синоним типа `__int64`, например:

```
typedef __int64 TMoney;
```

ПРИМЕЧАНИЕ

Чтобы не путать имена типов и имена объектов, договоримся имена типов начинать с префикса *T*. Именно такое соглашение действует в системах фирмы Borland. В системах фирмы Microsoft имена реализованных типов обычно начинаются с префикса *C*.

Все переменные-деньги теперь будут явно обозначены соответствующим словом, однако контроль их правильного использования по-прежнему должен осуществлять программист.

Нам нужна такая реализация, в которой операции были бы более тесно связаны с типом. При этом желательно, чтобы можно было выполнять только допустимые операции, а «неправильные» — невозможно. Поступим по-другому: сконструируем денежный тип из стандартных встроенных типов и напомним соответствующие функции обработки. В процедурном программировании сконструировать новый тип из стандартных можно только одним способом — с помощью структуры. Так как стандартные целые типы не обеспечивают (пока) требуемого диапазона денежных сумм, используем для представления денег тип `long double`¹.

Денежную сумму можно задать обычным дробным числом с двумя знаками после десятичной точки, обозначающим копейки. Например, 12.67 обозначает сумму 12 рублей 67 копеек. Оперировать, однако, будем суммами в копейках, округляя результат опять до копеек — для этого достаточно денежную сумму умножить на 100. Таким образом, хотя деньги представляются дробным числом типа `long double`, оперировать мы будем целой частью этого числа. Кроме того, отрицательные числа будут представлять долг. Наша структура с денежным полем показана в листинге 1.1.

¹ В системах Visual C++ тип `long double` реализован просто как `double`.

Листинг 1.1. Структура для представления денежных сумм

```
struct TMoney
{   long double Summa;           // денежная сумма
};
```

Как сейчас принято говорить, мы создали «оболочку» (wrapper) вокруг стандартного встроенного типа — «обернули» стандартный встроенный тип `long double` в структуру.

Как известно [4], в языке C определенное нами имя `TMoney` называется *тегом* структуры. В C такое объявление позволяет нам объявлять переменные следующим образом:

```
struct TMoney t;
```

Аналогичная запись используется при объявлении параметров и возвращаемых значений в заголовке функции. Чтобы несколько упростить запись, в программах на C обычно используют оператор `typedef`:

```
typedef struct teg_Money
{   long double Summa;           // денежная сумма
} TMoney;
```

Впрочем, в C++ то же самое можно писать даже без тега структуры:

```
typedef struct
{   long double Summa;           // денежная сумма
} TMoney;
```

После таких определений объявление переменных и запись параметров в заголовках можно упростить, не прописывая слово `struct`:

```
TMoney t;
```

В C++ при наличии структуры мы можем сразу объявлять переменные типа `TMoney` без использования оператора `typedef` и слова `struct`. Аналогично в заголовках функций обработки при определении параметров и возвращаемого значения разрешается указывать имя `TMoney` без слова `struct`. Прототипы этих функций могут выглядеть так, как показано в листинге 1.2.

Листинг 1.2. Заголовки функций обработки денежных сумм

```
TMoney AddMoney(const TMoney &a, const TMoney &b);           // сложение сумм
TMoney SubtractMoney(const TMoney &a, const TMoney &b);      // вычитание сумм
double DevideMoney(const TMoney &a, const TMoney &b);         // деление сумм
TMoney DevideByNum(const TMoney &a, const double &b);         // деление на число
TMoney MultByNum(const TMoney &a, const double &b);           // умножение на число
bool isNegative(const TMoney &a);                             // это долг?
int CompareMoney(const TMoney &a, const TMoney &b);           // сравнение сумм
TMoney ReadMoney(void);                                       // ввод
void DisplayMoney(const TMoney &a);                           // вывод на экран
```

Как видим, в таком варианте операции обработки денежных сумм уже гораздо теснее связаны с данными, чем при применении встроенного числового типа. Многое контролируется компилятором. По крайней мере, компилятор не позволит

задействовать вместо параметров типа `TMoney` никакие другие типы, сообщив об ошибке на этапе трансляции. Кроме того, с переменными типа `TMoney` невозможно использовать встроенные операции (например, сложение) по той же причине. Защищенность программы от случайных ошибок значительно повысилась, что не может не радовать.

Давайте в качестве примера рассмотрим реализацию нескольких функций: сложения `AddMoney()`, деления на число `DevideByNum()`, сравнения `CompareMoney()` и вывода на экран `DisplayMoney()`, чтобы почувствовать специфику работы с денежными суммами. Начнем с функции деления денежной суммы на число. Обычно число — это некоторый коэффициент пересчета денежной суммы (например, для перевода в другую валюту или для начисления процентов в банке), и он всегда положительный. После деления сумму надо округлить.

Удивительно, но в стандартной библиотеке C++ нет простой функции округления вроде `round()`, поэтому приходится использовать две другие: «округление вниз» `floor()` и «округление вверх» `ceil()`.

ПРИМЕЧАНИЕ

Чтобы было можно с этими функциями работать, необходимо подключить заголовок `<cmath>` или `<math.h>`.

Первая функция просто отсекает дробную часть или, как говорят, выдает наибольшее целое, меньшее заданного аргумента. Например, `floor(2.8)` выдает 2. Вторая функция, напротив, увеличивает результат до ближайшего целого, большего, чем аргумент. Например, `ceil(2.1)` выдаст 3, так же как и `ceil(2.9)`. Если надо округлить до целого числа положительное значение переменной `v`, то это делается следующим образом:

```
v = (v-floor(v)<0.5)?floor(v):ceil(v);
```

Эти функции правильно работают и с отрицательными числами: `floor(-2.1)` выдает `-3`, а результат `ceil(-2.9)` равен `-2`. Однако для денежной суммы знак «минус» означает только то, что эта сумма является долгом, поэтому оперировать будем абсолютным значением денежной суммы, а знак учитывать после деления и округления. Текст функции `DevideByNum()` представлен в листинге 1.3.

Листинг 1.3. Реализация функции деления денежной суммы на число

```
TMoney DevideByNum(const TMoney &a, const double &b)
{
    TMoney t = a;                                // локальный объект
    if (b > 0) {                                    // коэффициент должен быть > 0
        long double v = fabs(t.Summa);            // абсолютное значение
        v /= b;                                    // поделили
        v = (v-floor(v)<0.5)?floor(v):ceil(v);      // округление
        t.Summa = (t.Summa > 0)? v: -v;             // учли знак
    }
    return t;
}
```

В функции объявляется локальная денежная переменная, которой сразу же присваивается значение аргумента типа `TMoney`. Сумма делится на коэффициент и округляется до целых копеек. Округленное значение выдается в качестве результата. Функция сложения денежных сумм, текст которой представлен в листинге 1.4, в особых комментариях не нуждается.

Листинг 1.4. Функция сложения денежных сумм

```
TMoney AddMoney(const TMoney &a, const TMoney &b)
{
    TMoney t = a;           // локальный объект
    t.Summa += b.Summa;      // суммируем
    return t;                // возвращаем результат
}
```

По традиции, начало которой положено в языке C, функции сравнения выдают одно из трех целых значений:

- если первый аргумент меньше второго, то результат равен `-1`;
- если аргументы равны, то результат равен `0`;
- если первый аргумент больше второго, то результат равен `+1`.

Наша функция сравнения денежных сумм, представленная в листинге 1.5, следует этой традиции.

Листинг 1.5. Функция сравнения денежных сумм

```
int CompareMoney(const TMoney &a, const TMoney &b)
{
    int sign = 0;           // если равны
    if(a.Summa < b.Summa) sign = -1;      // первый меньше
    else if (a.Summa > b.Summa) sign = 1;  // первый больше
    return sign;
}
```

И наконец, наиболее интересная для нас функция — функция вывода на экран¹, текст которой приведен в листинге 1.6.

Листинг 1.6. Функция вывода на экран

```
typedef unsigned int uint;           // для сокращения записи
void DisplayMoney(const TMoney &a)
{
    string s = "";                  // строка результата
    string Digits = "0123456789";
    uint digit;
    bool negative = (a.Summa < 0);  // выясняем знак числа
    long double t = fabs(a.Summa);  // абсолютная величина суммы
    uint kop = fmod(t, 100);         // получаем копейки
    t = floor(t/=100);               // отсекаем копейки
    if (t > 0)                       // если есть рубли
    {
        while (t>=1)                // формируем рубли
        {
            digit = fmod(t, 10);     // очередная цифра
            t/=10;                   // отсеки цифру от суммы
        }
    }
}
```

продолжение ➤

¹ Для работы со строками мы будем пользоваться типом `string` [1-21], который реализован в составе стандартной библиотеки C++ (см. приложение).

Листинг 1.6 (продолжение)

```

        s=Digits[digit]+s;           // прицепили символ цифры к строке
    }
    else s += "0";                   // если рублей нет
    s+= " руб. ";
                                     // формируем копейки
    s+=Digits[kop/10];               s+=Digits[kop%10];
    s+= " коп. ";
    if (negative) s = '-' + s;
    cout << s << endl;
}

```

Сначала нужно «отделить» знак от числа, чтобы иметь дело только с положительным числом. Так как суммы хранятся в копейках, то надо отделить копейки от рублей — это делается функцией `fmod()`.

Затем, если сумма не нулевая, в цикле выполняется выделение цифр рублей и присоединение их к результирующей строке. Отдельная цифра выделяется вполне традиционным способом — как остаток от деления на 10. Поэтому при выделении всех цифр результат — переменная `t` — окажется меньше 1, что и указано в условии завершения цикла. После завершения цикла к строке рублей присоединяются копейки.

ВНИМАНИЕ

Такая реализация функции `DisplayMoney()` работает только в системе Visual C++.NET 2003. Чтобы она работала в системе C++Builder 6, необходимо вместо функции `fmod()` использовать функцию `fmodl()`. То же относится и к функции `DevideByNum()` (см. листинг 1.3), и к вызываемым в ней функциям `fabs()`, `floor()` и `ceil()` — вместо них нужно задействовать `fabsl()`, `floorl()` и `ceil()`.

Теперь осталось объединить все это вместе в один *модуль*, назвав его `TMoney.cpp`. Этот модуль мы сами или другие программисты могли бы использовать в разных программах, подключая¹ его с помощью препроцессора, например:

```
#include "TMoney.cpp"
```

Объединение данных и операций обработки в один модуль способствует повышению надежности программы, так как при возникновении любой ошибки в операциях с денежными суммами мы точно будем знать, где ее искать.

Наш модуль уже почти похож на новый тип данных. Однако мы по-прежнему имеем ряд проблем. Самой большой неприятностью является абсолютная незащищенность нашей структуры. Объявив переменную типа `TMoney`, программист может в обход всех функций обработки присвоить любые значения полю `Summa`. Но с этой проблемой можно частично справиться, написав соответствующую функцию инициализации. Прототип такой функции мог бы выглядеть так:

```
TMoney Init(const long double &r);
```

¹ О разработке многомодульных программ рассказывается в главе 13.

Здесь параметр *г* представляет собой денежную сумму. Вызов такой функции мог бы быть таким:

```
TMoney t = Init(123.67);
```

Однако заставить программиста *всегда* поступать «правильно», то есть использовать для присвоения значений только функцию инициализации, мы не можем. Более того, поле *Summa* можно непосредственно задействовать в арифметических операциях с числами (что мы и делали в приведенных ранее функциях). Конечно, «в здравом уме и твердой памяти» такое делать никто не будет, но отсутствие каких бы то ни было конструкций, защищающих внутреннюю структуру данных, здорово подрывает надежность наших программ. Основная причина такой ситуации состоит в том, что функции и структуры данных, которыми они оперируют, достаточно слабо связаны друг с другом. Объектно-ориентированный подход возник, в первую очередь, чтобы ликвидировать эту «пропасть» между данными и обрабатывающими их операциями.

Определение класса

Вообще-то говоря, программе-клиенту, которая собирается использовать наш тип данных, совершенно безразлично, как внутренне устроены денежные суммы. Клиенту нужно иметь возможность объявлять переменные типа *TMoney*, инициализировать их и оперировать ими, то есть присваивать им значения, вводить и выводить их на экран и в файл. Поэтому совершенно очевидно, что непосредственный доступ к данным должен быть закрыт. Это один из важнейших принципов объектно-ориентированного программирования, название которому — *инкапсуляция*.

Определение нового типа в языке C++ делается посредством конструкции класса (см. п. 9 в [1]). *Класс — это описание определяемого типа*. Синтаксически определение класса выглядит точно так же, как определение структуры, только вместо слова *struct* используется слово *class*¹:

```
class имя_класса  
{ /*... */ };
```

Имея такое описание, мы можем объявлять переменные типа *имя_класса*. *Переменная класса называется объектом, или экземпляром класса*. Класс мы объявляем один раз, а вот объектов обычно создаем столько, сколько необходимо. Причем, как и переменные встроенных типов, это могут быть массивы, одиночные переменные, указатели. Например, пусть мы объявили класс *TMoney*:

```
class TMoney  
{ /*... */ };
```

Тогда мы можем объявлять такие переменные:

```
TMoney t;           // скалярная переменная  
TMoney *p;          // указатель  
TMoney m[100];       // массив
```

¹ В C++ класс определяется тремя ключевыми словами: *class*, *struct* и *union*.

Эти переменные так же, как и переменные встроенных типов, подчиняются правилам видимости, и время их жизни зависит от места объявления. Мы можем также задействовать переменные этого типа в качестве полей структур и других классов, передавать в виде параметров и получать как результат из функции — мы это далее увидим.

Можно объявлять переменные, как и структуру в C, со служебным словом, например:

```
class TMoney t;                // скалярная переменная
class TMoney *p;              // указатель
class TMoney m[100];          // массив
```

Однако писать лишнее слово обычно не хочется — его и не пишут. Тем не менее обратите внимание на прямую аналогию с объявлением переменной-структуры в C.

Можно совместить определение класса и объявление переменных, например:

```
class TClass { /*... */ }      // определение класса
v1, v2;
```

Переменные `v1` и `v2` имеют тип `TClass`. Однако такое объявление менее удобно, чем приведенное ранее отдельное от определения класса объявление переменных, так как не выделяет в явном виде определение класса. Тем не менее после этого мы можем объявить другие переменные типа `TClass`, например:

```
TClass v3, v4[10];
```

Следующая конструкция называется *объявлением* класса:

```
class имя-класса;
```

Объекты такого класса объявлять нельзя — сам класс не определен. А вот указатели (и ссылки) на объекты — вполне можно. Обычно подобное объявление используется в тех случаях, когда один класс зависит от другого, но определение второго класса недоступно — мы увидим такие объявления в дальнейшем.

Доступ к элементам класса

Внутри класса точно так же, как и внутри структуры, можно объявлять поля. Таким образом, наш класс `TMoney` отличается от структуры `TMoney` только одним словом `class` (листинг 1.7).

Листинг 1.7. Класс `TMoney`

```
class TMoney
{   long double Summa;        // денежная сумма
};
```

Но это не единственное, и тем более, не самое важное различие. Гораздо важнее то, что по умолчанию все, что объявлено внутри класса, *недоступно* извне. Это сразу накладывает ограничение на инициализацию. Например, определим класс

Person с полями «Фамилия» и «Сумма» и структуры OtherPerson с такими же полями:

```
class Person
{
    string Fio;    double Summa; };
struct OtherPerson
{
    string Fio;    double Summa; };
```

Теперь определим переменные с инициализацией:

```
Person Kupaev = { "Кунаев М.", 10000.00 };           // ошибка!
OtherPerson Laptev = { "Лантев В.", 20000.00 };
```

В первом случае мы получим ошибку при компиляции. Более того, попробуем объявить следующую переменную без инициализации:

```
Person Kupaev;
```

Тогда оперировать полями, используя запись Kupaev.Summa или Kupaev.Fio, тоже будет невозможно — компилятор выдаст сообщение об ошибке. Можно сказать, что поля класса *невидимы* вне класса. Таким образом, конструкция класса *скрывает* информацию от внешнего мира, реализуя принцип инкапсуляции.

Но в таком случае у нас возникают проблемы: ни одна из ранее определенных функций (см. листинги 1.3–1.6) работать не будет! Однако совершенно очевидно, что операции, выполняющие обработку значений нашего типа данных, *должны* иметь доступ к внутренней структуре класса. Чтобы это стало возможно, наши функции, выполняющие операции с денежными суммами, нужно поместить *внутри* определения класса. Таким образом, функции не только получают доступ к внутренней структуре класса, но и явно связываются с типом данных.

Страуструп Б. в [2] называет такие функции функциями-членами, часто их называют компонентными функциями, однако мы будем придерживаться более традиционной объектно-ориентированной терминологии и назовем их *методами*. Это название принято практически во всех объектно-ориентированных языках, в том числе в Java и C#. Методы класса имеют неограниченный доступ к полям класса независимо от того, закрыты поля или нет.

Однако функция, помещенная внутри класса, тоже становится недоступной извне: она невидима вне класса. Мы просто не сможем ее вызвать! Таким образом, совершенно очевидно, что надо иметь возможность явно управлять доступом к элементам класса (см. п. 11 в [1]). В C++ управление доступом осуществляется с помощью зарезервированных слов `public` и `private`. Ключевое слово `public` открывает доступ, а слово `private` — закрывает. После ключевого слова требуется поставить двоеточие (:).

Так как структура по определению (см. п. 9/4 в [1]) тоже является классом, то эти ключевые слова можно использовать и в структуре. Наша структура TMoney, поля которой закрыты, выглядит так:

```
struct TMoney
{
    private:                // закрываем доступ к полям
        long double Summa;  // денежная сумма
};
```

В классе TMoney можно открыть поля:

```
class TMoney
{
    public:                // открываем доступ к полям
        long double Summa; // денежная сумма
};
```

Такой класс практически не отличается от структуры. Если мы откроем поля в классе Person, то сможем инициализировать поля точно так же, как поля структуры OtherPerson. Естественно, можно будет обращаться к полям Kupaev.Summa и Kupaev.Fio в любом месте программы.

И в классе, и в структуре разрешается написать столько слов public и private, сколько необходимо, и в том порядке, который требуется. Очередное слово действует до следующего. В принципе, если хотим, мы можем индивидуально объявлять каждый элемент класса либо открытым, либо закрытым. Открытая часть класса по традиции называется *интерфейсом* класса.

Определение методов класса

Таким образом, поля данных поместим в закрытую (или, как часто еще говорят, в приватную) часть класса, а методы — в открытую. Однако у нас осталось еще несколько вопросов: как определить метод, как вызывать его, какие параметры и возвращаемые значения разрешается прописывать в заголовке. Очевидно, что и определение метода, и его вызов должны отличаться от определения и вызова обычных функций.

Начнем с функции инициализации. Определение метода инициализации внутри класса выглядит так, как показано в листинге 1.8.

Листинг 1.8. Класс TMoney с методом Init()

```
class TMoney
{
    // закрытые поля
    long double Summa;                // денежная сумма
    public:                          // открытые методы
        void Init(const long double &t=0.0)
        { // переводим в копейки и отсекаем оставшийся "хвостик"
            long double r = floor(((t<0)? -t: t)*100);
            Summa = (t<0)? -r: r;      // учитываем знак
        }
};
```

Как видим, параметры метода так же, как и параметры обычной функции, можно задавать по умолчанию. Сначала мы получаем сумму в копейках — для этого положительное значение параметра умножается на 100 и с помощью функции floor() «отрезается хвост». Затем мы помещаем это значение в сумму, но уже с учетом знака исходной суммы.

Внутри метода имена полей указываются безо всяких дополнительных префиксов. Кроме того, заголовок метода отличается от заголовка аналогичной функции: метод не возвращает значение. Это связано с тем, что вызов метода отличается

от вызова функции: вызов метода осуществляется точно так же, как доступ к полям структуры — с помощью операции-селектора (точка). Пусть, например, у нас объявлен объект

```
TMoney t;
```

Тогда вызов метода `Init()` выглядит так:

```
t.Init(123.67);
```

Такая запись совершенно недвусмысленно говорит, что `Init()` — это метод, вызываемый для объекта `t`. И сразу понятно, что возвращаемое значение методу не нужно — инициализируется тот объект, для которого этот метод вызван.

В общем виде вызов любого метода для конкретного объекта выглядит так:

```
объект.метод(параметры)
```

Если метод возвращает значение некоторого типа, то такое выражение можно использовать в других выражениях, вычисляющих значения. Если значение не возвращается (как в методе `init()`), то выражение записывается как отдельный оператор вызова:

```
объект.метод(параметры);
```

Мы можем определить метод и вне класса. Но тогда надо явно указать, что определяемый метод принадлежит именно классу `TMoney`. А в классе остается только прототип (листинг 1.9).

Листинг 1.9. Внешнее определение метода `Init()`

```
class TMoney
{
    long double Summa;                // денежная сумма
public:
    void Init(const long double &t);   // объявляем прототип
};
void TMoney::Init(const long double &t=0.0)
{ // переводим в копейки и отсекаем оставшийся "хвостик"
    long double r = floor(((t<0)? -t: t)*100);
    Summa = (t<0)? -r: r;              // учитываем знак
}
```

Как видим, явное указание принадлежности классу задается префиксом — именем класса в заголовке определения метода. Тем не менее внутри метода имена полей (и других методов класса) задаются без всяких префиксов. Различие внешнего и внутреннего определений заключается только в том, что метод, определенный внутри класса, считается по умолчанию подставляемой (*inline*) функцией (см. п. 7.1.2/3 в [1]). Однако на деле компилятор отнюдь не всегда транслирует внутренний метод как подставляемый — это зависит целиком и полностью от реализации компилятора. В качестве подставляемых лучше определять небольшие методы, выполняющие простую работу и не содержащие циклов или рекурсивных вызовов. В остальном внешнее определение ничем не отличается от внутреннего. Вызов тоже точно такой же.

Теперь рассмотрим метод вывода на экран. Можно было бы его просто переписать как метод, однако мы поступим по-другому. Чтобы не иметь в дальнейшем проблем, связанных с добавлением новой валюты (например, долларов или евро), разделим нашу функцию вывода на экран на две:

- вспомогательная функция `toString()` формирует строку-число;
- основная функция вызывает первую и добавляет к строке-числу название денег, в нашем случае — рубли.

Функция преобразования числа в строку — хороший кандидат для закрытой части класса, а вторая функция — это наш метод вывода на экран, который, естественно, должен быть открыт. Таким образом, функцию `toString()` невозможно вызвать вне определения класса, но любой метод может ее беспрепятственно использовать для своих нужд.

Так как обе функции имеют доступ к полю суммы непосредственно, то параметры им не нужны. Таким образом, заголовок внешнего определения функции `toString()` выглядит так:

```
string TMoney::toString()
```

А заголовок метода вывода на экран должен быть таким:

```
void TMoney::DisplayMoney()
```

Тогда вызов для объявленного ранее объекта `t` выглядит так:

```
t.DisplayMoney();
```

Реализация обеих функций представлена в листинге 1.10.

Листинг 1.10. Функции вывода на экран

```
string TMoney::toString()
{
    string s = "";
    string Digits = "0123456789";
    uint digit;
    long double t = fabs(Summa);
    bool negative = (Summa < 0);
    uint kop = fmod(t, 100);
    t = floor(t/=100);
    if (t > 0)
    {
        while (t>=1)
        {
            digit = fmod(t, 10);
            t/=10;
            s=Digits[digit]+s;
        }
    }
    else s+="0";
    s+=".";
    s+=Digits[kop/10];
    if (negative) s = '-' + s;
    return s;
}
```

// строка-результат
// цифры для результата
// выделенная цифра
// делаем положительным
// запоминаем знак
// выделили копейки
// отсекаем копейки
// если есть рубли
// формируем рубли
// получаем цифру
// отсекаем цифру
// прицепляем символ-цифру

// нет рублей
// завершаем целую часть
// выделяем копейки
// учли знак

```

void TMoney::DisplayMoney()
{ string s = toString();           // получили строку-число
  s+=" руб.";                      // добавили обозначение рублей
  cout << s << endl;              // вывели на экран
}

```

Алгоритм фактически не изменился, поэтому детально его описывать нет необходимости.

Указатель this

Займемся теперь реализацией остальных методов. Начнем с функции сложения. Как и ранее написанная функция сложения (см. листинг 1.4), метод очевидно должен возвращать значение типа TMoney. Функция AddMoney() получала два аргумента-суммы. Но метод уже имеет доступ к одной сумме — это поле Summa в классе. Таким образом, параметр у метода AddMoney() должен быть единственным. Параметр — это правый аргумент сложения. Чтобы разобраться, почему именно правый, вспомним, как вызываются методы класса:

объект.метод(параметры)

Вызов метода AddMoney() для некоторого объекта t типа TMoney может быть таким (естественно, результат нужно записать в другую переменную типа TMoney):

```

TMoney t, p;
t.Init(100);
p.Init(100);
TMoney r = t.AddMoney(p);           // r = t + p

```

Таким образом, в качестве левого операнда сложения используется денежная сумма того объекта, для которого вызывается метод, а второй операнд (правый) — это аргумент метода. С учетом этих соображений реализация метода (листинг 1.11) мало отличается от реализации функции.

Листинг 1.11. Определение метода AddMoney()

```

TMoney TMoney::AddMoney(const TMoney &b)
{   TMoney t = b;                  // локальный объект
    t.Summa += Summa;              // просуммировали с суммой левого аргумента
    return t;                      // возврат результата
}

```

Метод AddMoney() можно реализовать по-другому, используя тот объект, который стоит слева от точки-селектора (листинг 1.12).

Листинг 1.12. Другая реализация метода AddMoney()

```

TMoney TMoney::AddMoney(const TMoney &b)
{   TMoney t = *this;              // текущий объект - левый аргумент
    t.Summa += b.Summa;            // сложили с суммой правого аргумента
    return t;                      // возврат результата
}

```


В методе мы объявили и инициализировали локальную переменную `t` текущим объектом, используя ключевое слово `this` (см. п. п. 9.3.2 в [1]). Вот что пишет об этом Б. Страуструп [2]: «Каждая функция-член „знает“, для какого объекта она вызвана, и может явно на него ссылаться..., ключевое слово `this` является указателем на объект, для которого вызвана функция». Фактически это означает, что каждый метод получает неявный дополнительный параметр — указатель на текущий объект.

ВНИМАНИЕ

Из этого общего правила есть исключения — статические методы, которые мы рассмотрим в главе 4.

Иногда доступ к полям класса делается через этот указатель, например:

```
this->Summa
```

Естественно, `*this` является значением, на которое указывает этот указатель. Это значение можно использовать в качестве возвращаемого, что мы и увидим в дальнейшем. Инициализация при объявлении в данном случае обеспечила нам присваивание полей текущего объекта полям объекта `t`.

Перегрузка методов

Методы так же, как и обычные функции, можно перегружать — это одно из проявлений принципа *полиморфизма* в C++. Метод `AddMoney()` — хороший кандидат для перегрузки. Параметры второго варианта метода должны отличаться от параметров первого: можно складывать деньги с числом (которое будем считать денежной суммой-константой). Тогда реализация второго метода может быть такой, как показано в листинге 1.13.

Листинг 1.13. Определение перегруженного метода `AddMoney()`

```
TMoney TMoney::AddMoney(const long double &r)
{
    TMoney t;                // локальный объект
    t.Init(r);                // инициализировали локальный объект
    return t.AddMoney(*this); // сложение с текущим объектом
}
```

Здесь мы применили важнейший принцип профессионального программирования: для реализации нового метода использовались уже проверенные и гарантированно правильно работающие методы того же класса. Поэтому риск написать ошибочный текст существенно снижается. Кроме того, при будущих возможных изменениях класса `TMoney` эта функция вряд ли изменится — разве что мы изменим имена методов инициализации и сложения.

Обратите внимание еще и на то, что аргументом при сложении является текущий объект — он имеет тип `TMoney`, поэтому у компилятора «вопросов не возникает». Последний оператор можно написать и по-другому:

```
return (*this).AddMoney(t);
```

Скобки вокруг `*this` писать обязательно, так как операция «точка»¹ имеет более высокий приоритет, чем операция «звездочка», поэтому без скобок мы получим указатель, а не объект типа `TMoney` — компилятор, естественно, «заругается». Реализуем еще метод умножения на константу. Этот метод отличается от метода деления на константу только операцией. И в том, и в другом требуется округлять результат (до целых копеек), поэтому мы реализуем приватную функцию округления `round()` — округляется абсолютное значение денежной суммы. Текст функции и метода приведен в листинге 1.14.

Листинг 1.14. Метод умножения на константу

```
// Функция округления - в приватной части класса
long double round(const long double &r)
{ long double t = fabs(r);           // абсолютная сумма
  t = (t-floor(t)<0.5)?floor(t):ceil(t); // округляем
  return (r<0)?-t:t;                // возвращаем со знаком
}

// метод
TMoney TMoney::MultByNumber(const double &b)
{ TMoney t = *this;
  if (b>0) t.Summa = round(Summa*b); // в методе деления Summa/b
  return t;
}
```

Обычно константой является некоторый коэффициент перевода одной валюты в другую, и это число всегда положительное. Именно поэтому в методе проверяется «положительность» аргумента.

Остальные методы реализуются так же просто — покажем это в листинге 1.15 вместе с интерфейсом класса `TMoney`. Кроме того, функция округления позволяет нам существенно упростить реализацию метода `Init()` (см. листинги 1.8 и 1.9). Новая реализация тоже представлена в листинге 1.15.

Листинг 1.15. Класс `TMoney`

```
typedef unsigned int uint;
class TMoney
{ long double Summa;
  // вспомогательные приватные функции
  string toString();           // перевод в строку
  long double round(const long double &r)
  { long double t = fabs(r);    // абсолютная сумма
    t = (t-floor(t)<0.5)?floor(t):ceil(t); // округляем
    return (r<0)?-t:t;         // возвращаем со знаком
  }
public:
  void Init(const long double &t);           // инициализация
  TMoney AddMoney(const TMoney &a);         // сложение денег
  TMoney AddMoney(const long double &r);    // сложение с числом
  TMoney SubtractMoney(const TMoney &b);    // вычитание денег
```

продолжение ➤

¹ Об операциях C++ мы поговорим в главе 3.

Листинг 1.15 (продолжение)

```

    int CompareMoney(const TMoney &a);           // сравнение денег
    TMoney MultByNumber(const double &b);        // умножение на число
    TMoney DevideByNum(const double &b);         // деление на число
    double DevideMoney(const TMoney &b);        // деление денег
    bool isNegative()                           // это долг?
    { return (Summa<0); }
    void ReadMoney();                           // ввод денег
    void DisplayMoney();                        // вывод денег
};
// метод сравнения
int TMoney::CompareMoney(const TMoney &a)
{
    int sign = 0;                               // если суммы равны
    if(Summa < a.Summa) sign = -1;
    else if (Summa > a.Summa) sign = 1;
    return sign;
}
// метод вычитания денежных сумм
TMoney TMoney::SubtractMoney(const TMoney &b)  // аналогично AddMoney
{
    TMoney t = *this;
    t.Summa -= b.Summa;
    return t;
}
// метод деления денежных сумм
double TMoney::DevideMoney(const TMoney &b)
{
    return fabs(Summa)/fabs(b.Summa);          // во сколько раз
}
// метод деления на число
TMoney TMoney::DevideByNum(const double &b)    // аналогично MultByNumber
{
    TMoney t = *this;
    if (b>0) t.Summa = round(Summa/b);
    return t;
}
// метод инициализации
void TMoney::Init(const long double &t=0.0)
{
    Summa = round(t*100);                      // знак учитывается при округлении
}
// метод ввода денежных сумм
void TMoney::ReadMoney()                      // ввод денег
{
    uint k;
    cout << "Рубли>"; cin >> Summa;           // ввод рублей
    bool negative = (Summa<0);                 // сумма отрицательная
    Summa = floor(fabs(Summa)*100);            // формирование рублей
    cout << "Коп>"; cin >> k;                  // ввод копеек
    if (k < 100) Summa+=k;                     // добавление копеек
    Summa = negative? -Summa: +Summa;          // учитываем знак
}

```

Метод деления денег обычно используется, чтобы выяснить, во сколько раз одна сумма превосходит другую. Поэтому делятся абсолютные значения денежных сумм.

Надо сказать еще несколько слов о методе ввода. Ввод рублей и копеек разделен. При вводе рублей мы отсекаем дробную часть, если пользователь нечаянно ее задаст.

Если вы заметили, методы (кроме методов `ReadMoney()` и `Init()`) не изменяют значения полей текущего объекта. Именно поэтому мы почти во всех методах объявили локальную переменную типа `TMoney`, значение которой и возвращаем в качестве результата. Если в дальнейшем у нас появятся методы, изменяющие состояние полей текущего объекта, то мы сможем возвращать значение `*this` текущего объекта оператором `return`:

```
return *this;
```

Размеры объектов класса

В связи с тем, что методы можно объявлять внутри класса, нужно разобраться, влияет ли размер метода на размер класса в памяти? И сколько байтов выделяет компилятор для класса и для структуры — одинаковы ли эти значения? Как обычно, это можно сделать с помощью функции `sizeof(тип)` (см. п. п. 5.3.3 в [1]). Интересно также выяснить, выделяется ли память для «пустых» классов, не содержащих ни полей, ни методов. Напишем простую программу, текст которой приведен в листинге 1.16.

Листинг 1.16. Размеры классов и структур

```
#include <iostream>
using namespace std;
class ct1 {}; // пустой класс
struct st1 {}; // пустая структура
class ct2{ char t;}; struct st2{ char t;}; // одно поле
class ct3{ int t;}; struct st3{ int t;};
class ct4{ long t;}; struct st4{ long t;};
class ct4l{ double t;}; struct st4l{ double t;};
class ct5{ long double t;}; struct st5{ long double t;};
// два поля
class ct6{ char r; long t;}; struct st6{ char r; long t;};
class ct7{ char r; long double t;}; struct st7{ char r; long double t;};
// один метод
class ct10{ void f(void) { int a = 1; } };
struct st10{void f(void) { int a = 1; } };
// поле и метод
class ct12{ int t; void f(void) { int a = 1; } };
struct st12{int t; void f(void) { int a = 1; } };
int main()
{
    cout <<"Empty ";
    cout <<"class " << sizeof(ct1) << " "; structure " << sizeof(st1)<< endl;
    cout <<"char ";
    cout <<"class " << sizeof(ct2) << " "; structure " << sizeof(st2)<< endl;
    cout <<"int ";
    cout <<"class " << sizeof(ct3) << " "; structure " << sizeof(st3)<< endl;
    cout <<"long ";
    cout <<"class " << sizeof(ct4) << " "; structure " << sizeof(st4)<< endl;
    cout <<"long double ";
    cout <<"class " << sizeof(ct5) << " "; structure " << sizeof(st5)<< endl;
    cout <<"char, long ";
}
```

продолжение ➤

Листинг 1.16 (продолжение)

```

    cout <<"class "<< sizeof(ct6) << "; structure "<< sizeof(st6)<< endl;
    cout <<"char,long dbl ";
    cout <<"class "<< sizeof(ct7) << "; structure "<< sizeof(st7)<< endl;
    cout <<"only method ";
    cout <<"class "<< sizeof(ct10) << "; structure "<< sizeof(st10)<< endl;
    cout <<"int, method ";
    cout <<"class "<< sizeof(ct12) << "; structure "<< sizeof(st12)<< endl;
    getch(); // остановка, чтобы посмотреть результаты
    return 0;
}

```

Результат работы этой программы в системе Visual C++.NET 2003 выглядит так:

```

Empty          class 1; structure 1
char           class 1; structure 1
int            class 4; structure 4
long           class 4; structure 4
double         class 8; structure 8
long double    class 8; structure 8
char,long      class 8; structure 8
char,long dbl  class 16; structure 16
only method    class 1; structure 1
int, method    class 4; structure 4

```

Так как мы не меняли никаких режимов трансляции, это означает, что в *отладочном режиме по умолчанию компилятор работает следующим образом*:

- ☐ для класса и структуры память выделяется совершенно одинаково;
- ☐ даже «пустой» класс занимает в памяти некоторое количество байтов; в данном случае — один;
- ☐ если в классе объявлено только одно поле, то памяти выделяется ровно столько, сколько занимает значение соответствующего типа в памяти;
- ☐ в Visual C++.NET 2003 размеры типов `double` и `long double` одинаковы и равны восьми;
- ☐ если в классе несколько полей, то выделяемый объем памяти кратен наибольшему размеру;
- ☐ метод в классе места не занимает.

Те же самые числа выдаются и в режиме трансляции `Release`. Однако при запуске той же программы в системе Borland C++ Builder 6 на экран выводятся другие значения:

```

Empty          class 8; structure 8
char           class 1; structure 1
int            class 4; structure 4
long           class 4; structure 4
double         class 8; structure 8
long double    class 16; structure 16
char,long      class 8; structure 8
char,long dbl  class 24; structure 24
only method    class 8; structure 8
int, method    class 4; structure 4

```

Как видите, и в этой системе метод не занимает места в классе, однако для «пустого» класса выделяется аж 8 байт! Кроме того, кратность выделения памяти для нескольких полей равна 8, а тип `long double` занимает 16 байт.

Чтобы заставить компилятор выделять для элементов классов и структур ровно столько памяти, сколько требуется, нужно использовать директиву препроцессора `#pragma`¹ (см. п. 16.6 в [1]). Хотя конструкция реализована как одна из директив препроцессора, на самом деле она предназначена для установок режимов компилятора и компоновщика. В частности, для *выравнивания по границе байта* необходимо задать аргумент `pack(1)`. Директива работает с момента объявления, поэтому сразу после директивы `#include` можно поместить любую из следующих двух конструкций:

```
#pragma pack(1)
#pragma pack(push, 1)
```

После этого в системе Visual C++ получим на выходе такой результат:

```
Empty          class 1; structure 1
char           class 1; structure 1
int            class 4; structure 4
long           class 4; structure 4
double         class 8; structure 8
long double    class 8; structure 8
char, long     class 5; structure 5
char, long dbl class 9; structure 9
only method    class 1; structure 1
int, method    class 4; structure 4
```

Как видите, в этом случае выделяется ровно столько памяти, сколько занимают соответствующие встроенные типы. По-прежнему для пустого класса выделяется 1 байт, а размер `long double` равен 8, что совпадает с размером «простого» типа `double`.

Вывод программы в системе Borland C++ Builder 6:

```
Empty          class 1; structure 1
char           class 1; structure 1
int            class 4; structure 4
long           class 4; structure 4
double         class 8; structure 8
long double    class 10; structure 10
char, long     class 5; structure 5
char, long dbl class 11; structure 11
only method    class 1; structure 1
int, method    class 4; structure 4
```

Этот вывод отличается только тем, что для типа `long double` выделяется 10 байт, а не 8, как для типа `double`. Это в точности соответствует размерам аппаратного типа в микропроцессоре Intel.

¹ Такого же эффекта можно добиться, установив в интегрированной среде соответствующий режим трансляции программы.

Интерес представляет также использование «пустого» класса в качестве поля другого класса. Если вы думаете, что «пустые» классы не имеют практического применения, то вы заблуждаетесь: очень часто с помощью «пустого» класса объявляется новый тип *исключения*¹.

Поэтому добавим в нашу программу следующие строки:

```
// в объявлении классов
class ct20 { ct1 a; }; struct st20 { ct1 a; };
class ct21 { st1 a; }; struct st21 { st1 a; };
class ct22 { ct1 a,b; }; struct st22 { ct1 a,b; };
class ct23 { ct1 a,b; int d; }; struct st23 { ct1 a,b; int d; };
// в функции main
cout << "Empty in " << endl;
cout << "class " << sizeof(ct20) << "; structure " << sizeof(st20) << endl;
cout << "class " << sizeof(ct21) << "; structure " << sizeof(st21) << endl;
cout << "class " << sizeof(ct22) << "; structure " << sizeof(st22) << endl;
cout << "class " << sizeof(ct23) << "; structure " << sizeof(st23) << endl;
```

И снова мы получаем разные результаты в системах Visual C++ и в C++ Builder. В первой при наличии директивы `#pragma` на экран выводится следующее:

```
Empty in
class 1; structure 1
class 1; structure 1
class 2; structure 2
class 6; structure 6
```

Это означает, что *каждое* «пустое» поле добавляет в класс 1 байт. В системе C++ Builder получаем несколько другой результат:

```
Empty in
class 1; structure 1
class 1; structure 1
class 1; structure 1
class 4; structure 4
```

Это означает, что в C++ Builder в отладочном режиме по умолчанию «пустое» поле *не занимает место в классе* — как был размер пустого класса равен 1, так и остался, несмотря на наличие «пустых» полей. Можно сказать, что по умолчанию компилятор Borland проводит *оптимизацию пустого класса* [28]. Однако можно установить режим трансляции, при котором «пустые» поля будут вести себя точно так же, как в Visual C++.

При «выключении» выравнивания ситуация по умолчанию остается аналогичной. В системе Visual C++.NET 2003 получаем на экране

```
Empty in
class 1; structure 1
class 1; structure 1
class 2; structure 2
class 8; structure 8
```

¹ Исключения рассматриваются в главе 7.

Вывод C++ Builder 6:

```
Empty in
class 8; structure 8
class 8; structure 8
class 8; structure 8
class 8; structure 8
```

Таким образом, и в этом случае система C++ Builder 6 по умолчанию не трогает место на «пустые» поля.

Использование класса

Имея определение класса `TMoney`, можем использовать его для объявления переменных, параметров и возвращаемых функциями результатов. Рассмотрим несколько примеров¹, первый из которых представлен в листинге 1.17.

Листинг 1.17. Объявление переменных типа `TMoney`

```
int main()
{
    TMoney t,p,s;                // объявление переменных
    t.Init(1000.67);             // инициализация
    p.Init(1000.67);
    TMoney x = t;                // инициализация другой переменной
    TMoney y(t);                 // инициализация другой переменной
    z = t.AddMoney(y);           // инициализация выражением
    s.Init(0.0);                 // обнуление
    z.Init();                     // обнуление

    // сложение денег
    s = t.AddMoney(p);           s.DisplayMoney();
    t.AddMoney(p).DisplayMoney(); // 1 - тоже сложение денег

    // деление на константу
    s = t.DevideByNumber(2);      s.DisplayMoney();

    // умножение на константу
    s = p.MultByNumber(3.23);     s.DisplayMoney();

    // ввод денег
    s.ReadMoney();                s.DisplayMoney();

    // деление сумм
    double d = p.DevideMoney(s);  cout << d << endl;
    return 0;
}
```

Пример демонстрирует определение и инициализацию переменных. Как видите, инициализировать переменные типа `TMoney` можно тремя способами: с помощью метода `Init()`, другой переменной или выражением, результатом вычисления которого является значение типа `TMoney`. Однако пока мы не можем определить инициализацию переменной типа `TMoney` в привычном виде:

```
TMoney u = 200.00;
```

¹ Во всех примерах предполагается, что текст программы включает в себя определение класса `TMoney` и все необходимые директивы `#include`.

Такая запись возможна только при наличии в классе *конструктора*. С конструкторами мы познакомимся в следующей главе, а пока перепишем метод инициализации `Init()` — реализуем возврат текущего объекта в качестве значения типа `TMoney` (листинг 1.18).

Листинг 1.18. Модифицированный метод `Init()`

```
TMoney TMoney::Init(const long double &t=0.0)
{
    Summa = round(t*100);
    return *this;                // возврат текущего объекта
}
```

Тогда мы сможем инициализировать переменные двумя способами. Прежний способ:

```
t.Init(432.78);
```

Новый способ:

```
TMoney t = z.Init(200);
```

Однако надо помнить, что в этом случае мы инициализируем сразу две переменные: объявляемую и ту, для которой вызван метод `Init()`.

Требуется объяснения также строка, помеченная цифрой 1:

```
t.AddMoney(p).DisplayMoney();    // 1 - тоже сложение денег
```

Ничего сложного или необычного в ней нет, если мы вспомним, что в качестве аргументов функций мы можем задавать выражения. Вызов функции `DisplayMoney()`, оперирующей структурой типа `TMoney` (см. листинг 1.2), может быть таким:

```
DisplayMoney(AddMoney(t,p));
```

В этом случае сначала выполняется вызов функции `AddMoney()`. Результат сложения имеет тип `TMoney`, но так как мы его никуда не помещаем, создается временный *анонимный объект* (см. п. 12.2 в [1]). Этот объект передается как аргумент функции `DisplayMoney()`.

В строке, помеченной цифрой 1, записано точно то же самое, только в «объектно-ориентированном» виде. Сначала выполняется сложение `t` и `p`. Результат метода сложения имеет тип `TMoney`, но так как мы его никуда не помещаем, создается временный анонимный объект. Этот объект и используется как левый аргумент метода `DisplayMoney()`. В результате на экране появляется точно такая же сумма, как и в предыдущей строке.

Мы смогли использовать в программе (см. листинг 1.17) не только определенные нами функции-методы, но и *операцию присваивания*. Для любого определяемого класса эта операция реализуется по умолчанию, если не определена программистом явно. Слева и справа от знака присваивания могут стоять объекты типа `TMoney`.

Точно так же, не заботясь о проблемах реализации массивов денег, мы можем использовать все операции, связанные с массивами. Простой пример, представленный в листинге 1.19, демонстрирует это.

Листинг 1.19. Использование массива денег

```

int main()
{
    TMoney A[5], sa;                                // объявление массива
    for(int i = 0; i < 5; i++)                        // инициализация элементов массива
        A[i].Init(i);
    for(int i = 0; i < 5; i++)                        // вывод элементов массива
        A[i].DisplayMoney();
    sa.Init(0.0);                                    // обнуление суммы
    for(int i = 0; i < 5; i++)                        // суммирование элементов массива
        sa=sa.AddMoney(A[i]);
    sa.DisplayMoney();
    sa.Init(0.0);                                    // второй вариант
    for(int i = 0; i < 5; i++)                        // суммирование элементов массива
        sa=A[i].AddMoney(sa);
    sa.DisplayMoney();
    return 0;
}

```

Наряду с обычными переменными мы можем задействовать в программе и указатели типа TMoney. В следующем примере, представленном в листинге 1.20, показано объявление и использование указателей типа TMoney.

Листинг 1.20. Использование указателей типа TMoney

```

int main()
{
    TMoney *pt = new TMoney(), *pp, *ps;            // объявление указателей
    pp = new TMoney(); ps = new TMoney();           // и создание объектов
    pt->Init(1000.67); pt->DisplayMoney();            // инициализация и вывод
    pp->Init(2000.78); pp->DisplayMoney();
    ps->Init(0.0); ps->DisplayMoney();
    *ps = (*pt).AddMoney(*pp);                      // сложение
    ps->DisplayMoney();
    ps->ReadMoney(); ps->DisplayMoney();              // ввод и вывод
    delete pp; delete pt; delete ps;               // уничтожение объектов
    return 0;
}

```

Пример простой; самое важное, что он демонстрирует, — способы доступа к методам через указатель. Чаше используется операция `->`, но доступ к методу сложения выполняется с помощью другой операции — разыменования `*`. И снова обратите внимание на скобки вокруг указателя `pt` — ситуация совершенно аналогична приведенной ранее для указателя `this`.

Для каждого указателя сначала операций `new` создается динамический объект типа TMoney, и только после этого выполняется инициализация — в данном случае без создания динамического объекта инициализировать нельзя. Это и понятно: у нас объявлены только указатели, а не сами объекты.

Объект создается вызовом конструктора — об этом сигнализируют круглые скобки после имени TMoney(). Отметим, что разрешается создавать объект, не показывая «лишних» скобок, например:

```

TMoney *pt = new TMoney, *pp, *ps;                // объявление указателей
pp = new TMoney; ps = new TMoney;                  // создание объектов

```

Однако тут нас, как всегда, подстерегает очередной «сюрприз», которыми так богат язык C++. Например:

```
TMoney *pt = new TMoney();
```

Это объявление не эквивалентно следующему:

```
TMoney *pt = new TMoney;
```

В первом случае выделяемая память инициализируется нулями, а во втором случае — нет. В этом легко убедиться, выполнив вывод значений на экран:

```
TMoney *pt = new TMoney(); pt->DisplayMoney();
TMoney *ps = new TMoney;   ps->DisplayMoney();
```

У меня на домашнем компьютере (Visual C++.NET 2003, режим Debug) на экран выводится следующее:

```
0.00 руб.
-62774385622041946822886424226644088866868024608228064048624264404.48 руб.
```

Совершенно очевидно, какой способ создания динамических переменных следует предпочесть.

В конце работы объекты надо, естественно, уничтожить, возвратив память системе. Мы можем задействовать класс и для объявления полей в другом классе. В объектно-ориентированном программировании использование некоторого класса для определения полей в другом классе называется *включением*, или *композицией*. Например, в банковской системе может быть реализован класс Счет, обязательными полями которого являются номер счета и сумма на счете. Упрощенное определение такого класса представлено в листинге 1.21.

Листинг 1.21. Определение класса Счет

```
typedef unsigned long ulong;
class TCount
{
    TMoney Summa;                // сумма на счете
    ulong NumberCount;           // номер счета
public:
    void DisplayCount()           // вывод на экран
    { cout << "Number: "<< NumberCount<<". Summa: ";
      Summa.DisplayMoney();
    }
    // инициализация
    void Init(ulong Number, const TMoney &s)
    { NumberCount = Number;
      Summa = s;                  // присваиваются деньги
    }
    // сложение денежных сумм на счетах
    TCount AddSumma(TMoney s)
    { TCount t = *this;
      t.Summa = t.Summa.AddMoney(s); // "вложенный" доступ
      return t;
    }
};
```

Мы определили минимальный набор методов: инициализация, вывод на экран и добавление денег на счет. Эти методы служат только для демонстрации использования нашего класса `TMoney` в качестве поля другого класса. Обратите внимание на то, что в методах класса `TCount` мы применяем методы класса `TMoney`. А в методе `AddSumma()` продемонстрирован «двойной вложенный» доступ к методу сложения денег.

Работу методов класса `TCount` демонстрирует простая программа, представленная в листинге 1.22.

Листинг 1.22. Использование класса `TCount`

```
int main()
{
    TMoney t,p,s;    p.Init(100.12);    // инициализация денег
    TCount tt, pp;
    tt.Init(1, p);    // инициализация счета
    tt = tt.AddSumma(p);    // добавление денег на счет
    tt.DisplayCount();
    pp.Init(2, p.MultByNumber(1.5));    // инициализация счета выражением
    pp.DisplayCount();
    return 0;
}
```

Программа работает, и результат ее работы на экране выглядит так:

```
1000.07 руб.
Number: 1. Summa: 200.24 руб.
Number: 2. Summa: 150.18 руб.
```

Резюме

Существуют различные парадигмы программирования. Объектно-ориентированный подход в программировании возник как попытка справиться со все возрастающей сложностью программных систем. В частности, было замечено, что инкапсуляция данных снижает количество ошибок, этому также способствует объединение данных и операций в одной конструкции — классе. Концепция класса обычно представляет собой отражение некоторой сущности реального мира и определяет структуру своих объектов. Объекты включают в себя как данные, так и функции обработки этих данных.

Доступ к элементам класса может быть разграничен спецификаторами доступа. Ключевое слово `public` объявляет элемент класса доступным вне класса, а ключевое слово `private` закрывает доступ извне. Скрытие информации о внутренней структуре класса называется инкапсуляцией. Функция, являющаяся элементом класса, называется методом класса. Методы класса, в отличие от внешних функций, имеют неограниченный доступ к элементам класса. Методы класса, как и внешние функции, можно перегружать. Перегрузка методов — одно из проявлений полиморфизма в C++.

Каждый метод имеет доступ к текущему объекту посредством предопределенного указателя, который обозначается зарезервированным словом `this`. Как обычно,

обозначение `*this` представляет собой значение, поэтому с текущим объектом можно работать, используя это обозначение.

Таким образом, класс представляет собой новый тип данных, реализованный средствами C++. Объекты этого типа можно применять практически так же, как и объекты встроенных типов, например: передавать в качестве параметра в функции, получать в качестве результата, объединять в массивы, включать в структуры и другие классы. Использование объекта одного класса в качестве поля другого класса называется композицией.

Каждый объект некоторого класса занимает определенное количество байтов памяти, где хранятся значения полей данных для этого объекта. Но методы не занимают места в памяти объекта и хранятся в единственном экземпляре. Объем памяти, выделяемой системой программирования под объект, зависит от реализации и конкретных установок компилятора.

Контрольные вопросы

1. Что определяет класс? Чем отличается класс от объекта?
2. Можно ли объявлять массив объектов? А массив классов?
3. Разрешается ли объявлять указатель на объект? А указатель на класс?
4. Можно ли совместить определение класса с объявлением объекта?
5. Объясните разницу между определением класса и объявлением класса.
6. Объясните, чем различаются два объявления указателя:

```
TClass *p = new TClass;  
TClass *p = new TClass();
```
7. Как называется использование объекта одного класса в качестве поля другого класса?
8. Является ли структура классом? Чем класс отличается от структуры?
9. Какие ключевые слова в C++ обозначают класс?
10. Объясните принцип инкапсуляции.
11. Для чего нужны ключевые слова `public` и `private`? Можно ли использовать ключевые слова `public` и `private` в структуре?
12. Существуют ли ограничения на использование ключевых слов `public` и `private` в классе? А в структуре?
13. Обязательно ли делать поля класса приватными? Как инициализировать приватные поля класса?
14. Что такое «метод»? Как вызывается метод? Может ли метод быть приватным?
15. Как определить метод непосредственно внутри класса? А вне класса?
16. Объясните, что понимается под интерфейсом класса.
17. Что обозначается ключевым словом `this`? Для чего может использоваться конструкция `*this`?

18. Что такое композиция?
19. Разрешается ли внутри метода объявлять объекты «своего» класса? Как присваивать таким объектам начальное значение?
20. Сколько места в памяти занимает объект класса? Как это узнать?
21. Каков размер «пустого» объекта? Влияют ли методы на размер объекта?
22. Одинаков ли размер класса и аналогичной структуры?
23. Что такое выравнивание и от чего оно зависит? Влияет ли выравнивание на размер класса?
24. Покажите, как осуществить выравнивание полей класса по границе двух байтов.
25. Разрешается ли параметрам методов присваивать значение по умолчанию?
26. Объясните, почему методы, реализующие бинарные операции (например, сложение), должны иметь один параметр.
27. Объясните назначение директивы `#pragma`.
28. Какой принцип объектно-ориентированного программирования проявляется в перегрузке методов?

Упражнения

Во всех упражнениях, помимо указанных методов, обязательно должны быть реализованы метод инициализации `Init()`, метод ввода данных с клавиатуры `Read()`, метод вывода данных на экран `Display()`.

Для демонстрации работы с объектами нового типа во всех заданиях требуется написать главную функцию.

1. Создать класс `Трапеция` с методами вычисления площади и периметра. Определить, какие поля необходимы в классе. Площадь трапеции вычисляется по формуле

$$S = (a + b) \cdot h / 2.$$

2. Реализовать класс `Power` с двумя полями: `first` и `second`. Поле `first` — дробное число; поле `second` — дробное число, показатель степени. Реализовать метод `power()` — возведение числа `first` в степень `second`.
3. Реализовать функцию с именем `make_Power()`. Функция должна получать в качестве параметров значения для полей класса `Power` из предыдущего задания, а возвращать объект типа `Power`. При передаче ошибочных параметров — выводить сообщение и заканчивать работу.
4. Создать класс `Double`, имитирующий стандартный тип `double`. Помимо метода инициализации `init()`, методов ввода `read()` и вывода `display()`, реализовать методы сложения и вычитания, а также метод возведения в произвольную степень.
5. Создать класс `Fraction` для работы с дробными числами. Число должно быть представлено двумя полями: целая часть — целое со знаком, дробная часть —

беззнаковое целое. Реализовать арифметические операции сложения, вычитания, умножения и сравнения на равенство.

6. Создать класс `Decimal` для работы с десятичными целыми числами со знаком произвольной длины. Число должно быть представлено строкой типа `string`, каждый символ которой — десятичная цифра, знак стоит слева первым и представлен символами `+` (плюс) или `-` (минус). Младшая цифра имеет младший индекс. Реализовать арифметические операции сложения и вычитания, а также сравнения на равенство и на больше.
7. Реализовать класс `Bill`, представляющий собой разовый платеж за телефонный разговор. Класс должен включать в себя поля номера телефона, тарифа за минуту разговора, скидки (в процентах), времени разговора (в минутах) и суммы к оплате. Реализовать метод вычисления суммы к оплате. В главной программе продемонстрировать создание, инициализацию и обработку массива объектов типа `Bill` с различными исходными данными для вычисления сумм к оплате. Вычислить общую сумму к оплате.
8. Реализовать класс `Money` с двумя полями: `first` и `second`. Поле `first` — целое положительное число, номинал купюры; номинал может принимать значения 1, 2, 5, 10, 50, 100, 500. Поле `second` — целое положительное число, количество купюр данного достоинства. Реализовать метод `summa()` — вычисление суммы денег. В главной программе продемонстрировать создание, инициализацию и обработку массива объектов типа `Money` с различными номиналами купюр. Вычислить общую сумму денег.
9. Создать класс `Time` с двумя полями, представляющими собой часы и минуты. Разработать метод, вычисляющий время через заданное количество часов и минут, а также метод, вычисляющий количество минут между двумя моментами времени.
10. Реализовать структуру `Trio` с тремя полями (целыми без знака). В структуре реализовать методы `init()`, `read()` и `display()`. Реализовать класс `Date`, используя в качестве поля объект типа `Trio`. В классе реализовать методы получения и изменения отдельных полей даты, метод инициализации строкой вида «год.месяц.день» (например, «2006.04.07»), метод вычисления даты через заданное количество дней.
11. Создать класс `BitString` для работы с битовыми строками произвольной длины. Битовая строка должна быть представлена строкой типа `string`, один символ этой строки представляет собой один бит и принимает значение 0 или 1. Младший бит имеет младший индекс. Реализовать традиционные операции для работы с битами (`and`, `or`, `xor`, `not`).
12. Реализовать класс `Account`, представляющий собой банковский счет. В классе должны быть пять полей: фамилия владельца, номер счета, процент начисления, дата открытия счета и сумма в рублях. Для представления даты использовать класс `Date` из упражнения 10. Необходимо выполнять следующие операции: сменить владельца счета, снять некоторую сумму денег со счета, положить деньги на счет, начислить проценты, перевести сумму в доллары.

Глава 2

Конструкторы

Инициализация объектов типа `TMoney`, который был разработан в предыдущей главе, не похожа на инициализацию переменных встроенных типов. Как мы знаем, любую переменную встроенного типа (например, типа `int`) можно инициализировать при объявлении, например, так:

```
int a(5);  
int b = 6;
```

Хотелось бы при объявлении переменных типа `TMoney` тоже иметь возможность писать что-то вроде

```
TMoney D(100.67);  
TMoney G = 1000.00;
```

Инициализацию нам обеспечит один из фундаментальных механизмов C++ — конструкторы (см. п. 12.1 в [1]).

Определение конструктора

Конструктор — это особый метод, имеющий имя, совпадающее с именем класса.

ПРИМЕЧАНИЕ

В стандарте (см. п. 12.1 в [1]) написано, что конструктор — это метод, не имеющий собственного имени.

Количество и типы параметров конструктора могут быть любыми, но обычно параметры используются для заполнения полей класса. Однако конструктор имеет одно важнейшее свойство, существенно отличающее его от всех остальных методов: конструктор *не возвращает результата* — нельзя писать даже `void`.

Напишем конструктор нашего класса `TMoney`, который обеспечит нам инициализацию переменных-денег при объявлении. Имя конструктора совпадает с именем

класса, а параметры, очевидно, такие же, как у метода `Init()`. Тело конструктора тоже, конечно же, выполняет работу функции инициализации. Таким образом, реализация конструктора «внешним» способом выглядит так, как показано в листинге 2.1.

Листинг 2.1. Конструктор класса `TMoney`

```
TMoney::TMoney(const long double &r)
{ Summa = round(r*100);
}
```

Мы снова применили все тот же прием: использовали уже реализованную и отлаженную функцию округления. Определив такой конструктор, мы получили возможность объявлять переменные и инициализировать их, например:

```
TMoney t(1000.67);
TMoney d = 10.67;
```

Заголовок конструктора, очевидно, должен быть задан в открытой части класса, так как иначе мы не сможем создавать объекты. Перенесем прототип конструктора в закрытую часть класса:

```
class TMoney
{ long double Summa;
  TMoney(const long double &t);
public:
};
```

Тогда в ответ на все приведенные объявления переменных типа `TMoney` система Borland C++ Builder 6 выдаст сообщение E2247 о том, что следующий метод недоступен:

```
TMoney::TMoney(const long double &)
```

Аналогичное сообщение выдает и система Visual C++.NET 2003. Как мы в дальнейшем увидим, такое поведение иногда бывает полезно, чтобы запретить создавать объекты.

Однако неожиданно выясняется, что даже при открытом конструкторе объявления без инициализации, наподобие следующего, являются ошибочными:

```
TMoney t;
```

Borland C++ Builder 6 выдает сообщение об ошибке E2285:

```
Could not find a match for 'TMoney::TMoney()'
```

Это сообщение говорит о невозможности найти соответствие для `TMoney::TMoney()`.

Аналогичное сообщение выдает и Visual C++.NET 2003. Между тем, пока в классе не был определен конструктор, проблем с объявлениями не возникало. Дело в том, что по правилам C++ (см. п. п. 12.1/5 в [1]) простейший конструктор без аргументов создается автоматически. Но стоит нам определить хотя бы один

конструктор явным образом, C++ «умывает руки» и полагается на программиста. Вспомним, каким образом мы можем объявить структуру:

□ без инициализации:

```
TMoney t;
```

□ с инициализацией полей:

```
TMoney t = {0.0};
```

□ с инициализацией другой, уже объявленной структурой:

```
TMoney r = t;
```

Соответственно и видов конструкторов в классе должно быть столько же. Первый конструктор, который не имеет аргументов (см. п. п. 12.1/5 в [1]) и создается автоматически, если конструкторы не определены, называется *конструктором по умолчанию*. Создаваемый системой конструктор имеет вид

```
class(){} 
```

Его наличие в классе делает возможным объявления первого вида. Второй — это конструктор инициализации, именно его мы реализовали. Наличие такого конструктора обеспечивает инициализацию полей класса. Если он определен, то использовать стандартную форму инициализации полей нельзя — надо писать вызов конструктора. Вообще-то говоря, конструкторов инициализации, как правило, бывает несколько — мы увидим это в дальнейшем.

Третий конструктор называется конструктором копирования (см. п. п. 12.1/10 в [1]). Такой конструктор копирует поля уже существующего объекта в поля объявляемого. Объект передается конструктору в качестве параметра. Для нашего класса `TMoney` он мог бы быть таким, как показано в листинге 2.2.

Листинг 2.2. Конструктор копирования

```
TMoney::TMoney(const TMoney &r)
{ *this = r; }
```

Обратите внимание на то, что параметр передается по ссылке — передавать по значению нельзя! Попробуем объявить заголовок в классе:

```
TMoney(const TMoney r);
```

Даже без реализации просто при объявлении заголовка в классе с параметром, передаваемым по значению, Visual C++.NET 2003 выдает сообщение об ошибке C2652:

```
'TMoney' : illegal copy constructor: first parameter must not be a 'TMoney'
```

Это сообщение говорит о недопустимости такого конструктора копирования — его первый параметр не должен иметь тип `TMoney`.

Аналогичное сообщение выдает и C++ Builder 6.

Вообще-то параметр необязательно должен быть константным, достаточно, чтобы он передавался по ссылке. Конструктор копирования, приведенный в листинге 2.2,

обычно создается системой по умолчанию, если только не определен программистом явно. Однако мы можем объявить собственный конструктор с параметром, передаваемым по неконстантной ссылке. Более того, в классе может быть объявлено несколько конструкторов копирования, и стандарт (см. п. 12.8 в [1]) разрешает определять несколько параметров, если им присваивается значение по умолчанию, например:

```
class X
{
    //...
    public:
        X(int t);                // конструктор инициализации
        X(X &t, int r = 0);      // конструктор копирования
        // ...
}
X a(1);                        // работает конструктор инициализации
X b(a,2);                     // работает конструктор копирования
X d = b;                       // работает конструктор копирования
```

Как было отмечено ранее, определять конструктор копирования не обязательно, так как он создается по умолчанию. Однако в дальнейшем мы увидим, что его иногда приходится определять явно, если создаваемый по умолчанию конструктор нас не устраивает.

Точно так же по умолчанию создается операция присваивания:

```
TMoney& operator=(const TMoney &r)
{ *this = r; return *this; }
```

Не задумываясь пока о необычности имени в этом прототипе¹, отметим, что и параметр, и возвращаемое значение передаются как ссылки на тип `TMoney`. Обратите внимание, что параметр задан таким же способом, как и в конструкторе копирования. Работает такая операция аналогично конструктору копирования: копирует поля правого объекта в левый. Именно поэтому мы смогли использовать ее в приведенных в главе 1 примерах (см. листинги 1.17 и 1.19).

Таким образом, наш класс `TMoney` с тремя конструкторами (без аргументов, инициализации и копирования) выглядит теперь так, как показано в листинге 2.3.

Листинг 2.3. Класс `TMoney` с конструкторами

```
class TMoney
{
    long double Summa;
    public:
        TMoney(){ Summa = 0.0; }
        TMoney(const long double &t);
        TMoney(const TMoney &r);
};
```

Конструктор по умолчанию ввиду отсутствия аргументов просто обнуляет поля. Тело конструктора инициализации совпадает с телом метода `Init()`, поскольку

¹ Ничего необычного на самом деле нет: аналогичным образом определяется любая перегружаемая операция (см. главу 3).

они выполняют одну и ту же работу. Так как конструктор копирования определять не обязательно, мы можем оставить в классе только конструктор инициализации, если объявим параметры по умолчанию (листинг 2.4).

Листинг 2.4. Класс `TMoney` с одним конструктором

```
class TMoney
{
    long double Summa;
public:
    TMoney(const long double &t);
};
TMoney::TMoney(const long double &r=0.0)
{ Summa = round(r*100);
}
```

Как и в обычных функциях, параметры по умолчанию могут быть заданы в прототипе, который прописан в классе. Тогда в реализации их указывать нельзя.

Конструкторы и объекты

Показанное в листинге 2.4 определение класса `TMoney` дает нам возможность объявлять объекты любым из трех упомянутых ранее способов и по-разному инициализировать переменные типа `TMoney`:

```
TMoney d1;                // нулевые поля
TMoney d2(100.67);        // явный вызов конструктора инициализации
TMoney d0 = 100;          // инициализация полей
TMoney d5 = TMoney(100);   // явный вызов конструктора
TMoney d3(d2);             // конструктор копирования
TMoney d4 = d2;            // конструктор копирования
```

Переменная `d1` обнуляется, так как по умолчанию параметры конструктора нулевые. Объявление переменной `d2` — это явный вызов конструктора инициализации. Такую же форму инициализации добавили в C++ и для встроенных типов, чтобы формально обеспечить единообразие для конструируемых и встроенных типов.

Инициализация переменных `d5`, `d3` и `d4` — это вызов конструктора копирования, причем для `d5` сначала явно вызывается конструктор инициализации и создает временный объект. С переменной `d0` дело обстоит несколько сложнее — мы инициализируем переменную `TMoney` числом, используя синтаксис, при котором должен работать конструктор копирования. Поэтому сначала неявно должен быть вызван конструктор инициализации, который создаст временный объект, а затем уж этот объект копируется в переменную `d0`.

Однако на практике такая последовательность действий выполняется далеко не всегда — для ускорения работы программы компилятором часто применяется та или иная оптимизация, и разные системы поступают по-разному. Чтобы в этом убедиться, достаточно добавить в класс явный конструктор копирования (см. листинг 2.2), но сделать его закрытым. Тогда при трансляции класса в системе Visual C++ .NET 2003 для переменных `d5`, `d3` и `d4` выдается сообщение об ошибке C2248:

```
'TMoney::TMoney' : cannot access private member declared in class 'TMoney'
```

Это сообщение говорит о том, что запись `TMoney::TMoney` не обеспечивает доступ к закрытому члену класса `TMoney`.

Система C++ Builder 6 при наличии в классе закрытого конструктора копирования выдает сообщение об ошибке E2247:

```
'TMoney::TMoney(const TMoney &)' is not accessible
```

Это сообщение говорит о том, что конструктор копирования `TMoney::TMoney(const TMoney &)` недоступен ни для переменных `d3`, `d4`, `d5`, ни для переменной `d0`.

Чтобы проверить, какой конструктор реально вызывается, добавим в конструктор инициализации строку

```
cout << "Init ";
```

Кроме того, вставим в класс `TMoney` описанный ранее открытый конструктор копирования, добавив в него строку

```
cout << "Copy ";
```

Тогда при запуске программы¹ выясняется, что для переменных `d0` и `d5` вызывается конструктор инициализации, а для `d3` и `d4` — конструктор копирования.

Наличие конструктора обеспечивает нам инициализацию динамических переменных типа `TMoney` при любой форме записи, поэтому следующие два объявления работают одинаково:

```
TMoney *p = new TMoney;
TMoney *p = new TMoney();
```

В обоих случаях динамическая память обнуляется (`*p = 0`), то есть вызывается объявленный нами конструктор инициализации с параметром, равным 0. Мы можем теперь присваивать начальные значения динамическим переменным, например:

```
TMoney *p = new TMoney(100.67);
```

Объявим два конструктора. Конструктор без аргументов:

```
TMoney()
{ Summa = 0.0; }
```

Конструктор инициализации:

```
TMoney(const long double &r=0.0)
{ Summa = round(r*100); }
```

Тогда при объявлении указателей без инициализации динамического объекта вызывается конструктор без аргументов (который обнуляет поля):

```
TMoney *p = new TMoney;           // вызывается конструктор без аргументов
TMoney *p = new TMoney();         // вызывается конструктор без аргументов
```

В то же время для инициализированного динамического объекта работает конструктор инициализации.

¹ В Visual C++.NET 2003.

Ранее мы упоминали о возможности присваивать объекты, так как операция присваивания создается по умолчанию. Однако наличие в классе конструктора инициализации позволяет присваивать объектам `TMoney` даже числа, например:

```
d1 = 100.24;
```

Более того, справа от операции присваивания разрешается задавать числовые выражения любой сложности, например:

```
d2 = 100.24-5.45/1.2+12.37;
```

Выражение вычисляется, и результат попадает в переменную типа `TMoney`. Как легко убедиться, перед присваиванием вызывается конструктор инициализации, который создает временный анонимный объект. Этот объект и присваивается переменной, указанной слева от знака присваивания. В этом случае конструктор инициализации выполняет *неявное преобразование типа*¹. Заметим, что выражения (справа от знака присваивания) с объектами типа `TMoney` не транслируются. Точно так же вызывает ошибки компиляции использование операций с присваиванием вроде `+=`. Ошибки вызваны тем, что в классе отсутствуют определения соответствующих операций, а по умолчанию они не создаются.

В предыдущей главе (см. листинг 1.19) мы показали, что даже без конструкторов можно определять массив объектов и использовать индексированные выражения в качестве аргументов методов и объектов, для которых эти методы вызываются. Разберемся с этим вопросом подробнее. Определим в программе массив, например:

```
TMoney M[100];
```

Легко убедиться, что конструктор вызывается при создании *каждого элемента массива* (см. п. п. 8.5/5 в [1]). Точно так же конструктор вызывался и в листинге 1.19, но мы этого не наблюдали, так как работал создаваемый системой по умолчанию конструктор без аргументов.

Наш конструктор обеспечивает привычную инициализацию массива:

```
TMoney A[5] = { 1, 2, 3, 4, 5};
```

Элементы массива `A` получают значения 1 руб., 2 руб., 3 руб., 4 руб., 5 руб. Для каждого целого числа, представленного в списке, вызывается конструктор инициализации (см. п. п. 12.6/3 в [1]), поэтому на самом деле инициализация массива представляет собой следующее:

```
TMoney A[0]=TMoney(1);  
TMoney A[1]=TMoney(2);  
TMoney A[2]=TMoney(3);  
TMoney A[3]=TMoney(4);  
TMoney A[4]=TMoney(5);
```

Аргумент конструктора неявно переводится из целого `int` в тип `long double`.

¹ С определением *неявных преобразований* и перегрузкой операций мы подробно познакомимся в следующей главе.

Если мы определим в классе `TMoney` два конструктора: один с аргументом, другой без аргумента (как показано ранее для указателей), то убедимся, что при создании массива без инициализации для каждого элемента вызывается конструктор без аргументов; если же массив инициализируется, то для каждого элемента вызывается конструктор инициализации.

ВНИМАНИЕ

Запомните, что при создании массива конструктор вызывается для создания (и инициализации) каждого элемента.

Объявление поля типа `TMoney` в классе ничем не отличается от рассмотренного ранее, поэтому еще раз мы его рассматривать не будем.

Конструкторы и параметры

Мы уже неоднократно пользовались тем, что объекты реализованного класса можно передавать как параметры (любыми допустимыми способами) и возвращать в качестве результатов. Однако наличие конструкторов вносит дополнительные нюансы в этот процесс. Определим несколько простых функций с параметром типа `TMoney`, передаваемым в них различными способами:

```
void f1(TMoneу t)           // по значению
{ t.DisplayMoney(); }
void f2(TMoneу &t)          // по ссылке
{ t.DisplayMoney(); }
void f3(TMoneу *t)          // по указателю
{ t->DisplayMoney(); }
```

Добавим в класс конструктор копирования и вставим в конструкторы диагностический вывод на экран. Объявим объект типа `TMoneу` и вызовем функции:

```
TMoneу d2(100.67);         // инициализация
f1(d2);                     // по значению - вызывается конструктор копирования
f2(d2);                     // по ссылке - не вызываются
f3(&d2);                     // по адресу - конструкторы не вызываются
```

Как выясняется, при передаче аргумента типа `TMoneу` по значению конструктор копирования вызывается, а в остальных случаях — нет.

Теперь разберемся, сколько и каких конструкторов вызывается, если параметр — выражение. Выражение можно передавать как параметр в функции `f1()`.

```
f1(100);                     // вызывается только конструктор инициализации
f1(100+100);                 // вызывается только конструктор инициализации
```

Оказывается, если в качестве параметра передается константное выражение (или просто константа), то компилятор¹ поступает очень «мудро»: выражение вычисляется (на стадии компиляции), и один раз вызывается конструктор инициализации, который конструирует из полученного числового значения передаваемый в функцию объект типа `TMoneу`.

¹ В Visual C++.NET 2003.

ПРИМЕЧАНИЕ

Если необходимо, выполняется неявное преобразование из типа `int` в тип `long double`.

Ситуация аналогична показанной ранее при использовании операции присваивания. Это очень важное наблюдение: *вызов функции может сопровождаться вызовом конструкторов для преобразования типов параметров*. Таким образом, вывод из этого следует совершенно определенный: передача параметров в функцию может сопровождаться существенными накладными расходами — вызываются конструкторы инициализации и (или) копирования.

При возврате объекта по значению тоже вызывается конструктор копирования. Именно это и предписано в стандарте (см. п. п. 12.8/1 в [1]): при передаче в функцию параметра по значению (и при возврате результата) конструктор копирования должен быть вызван для создания скрытой «внутренней» копии объекта.

ВНИМАНИЕ

Запомните, что передача аргументов в функцию и возврат результата могут сопровождаться работой конструктора.

Конструкторы позволяют нам естественным образом задавать параметрам значения по умолчанию. В функции `f1()` в качестве начального значения могут быть заданы константы, например:

```
void f1(TMoney t = 100)
{ t.DisplayMoney(); }
```

Как видите, объявление параметра ничем не отличается от объявления инициализируемой переменной типа `TMoney`. При отсутствии конструкторов присвоение начальных значений вызывает некоторые трудности, так как присвоить можно только выражение типа `TMoney`. Естественно, вызов функций даже без параметра в обоих случаях сопровождается вызовом конструктора инициализации. Если же мы хотим инициализировать параметры переменными, то они, конечно, должны быть видны в точке объявления прототипа.

Точно так же можно присвоить начальное значение указателю в функции `f2()`, например:

```
void f2(TMoney *t = new TMoney)
{ t->DisplayMoney(); }
```

Или

```
void f2(TMoney *t = new TMoney(300))
{ t->DisplayMoney(); }
```

И в этом случае вызывается конструктор инициализации. Надо только учесть, что в этой функции при таком объявлении может произойти утечка памяти, если вызвать функцию без параметра: параметр-указатель передается по значению, и в функции создается внутренняя локальная копия указателя. При завершении функции локальная копия указателя просто уничтожается и возникает потерянная ссылка.

Естественно, для присвоения начального значения можно вызвать и функцию, возвращающую значение типа `TMoney`. Интересно, что язык C++ позволяет вызывать определяемую функцию для присвоения значения собственному параметру. Немного изменим функцию `f1()` — пусть она возвращает значение типа `TMoney`:

```
TMoney f1(TMoney t)
{ cout << "f1="; t.DisplayMoney(); t = t.AddMoney(1);
  return t;
}
```

Тогда мы можем проделать фокус, который иллюстрирует листинг 2.5.

Листинг 2.5. Присвоение начального значения параметру той же функцией

```
TMoney f1(TMoney t);
TMoney f1(TMoney t = f1(100))    // вызов определяемой функции
{ cout << "f1="; t.DisplayMoney(); t = t.AddMoney(1.0);
  return t;
}
```

Посмотрите внимательно на этот пример: для присвоения начального параметра в заголовке `f1()` используется функция `f1()`! И это не является рекурсивным вызовом. Прототип функции писать обязательно: если его не будет, при трансляции возникают ошибки.

Проверка:

```
f1().DisplayMoney();      cout << endl;        // - 1 -
f1(200).DisplayMoney();   cout << endl;        // - 2 -
```

Эта проверка показывает, что функция работает совершенно правильно. На экран выводится:

```
f1=100.00                // вызов в прототипе
f1=101.00                // вызов в - 1 - без параметра
102.00                  // вывод результата вызова
f1=200.00                // вызов в - 2 - с явным параметром
201.00                  // вывод результата вызова
```

В [10] можно найти несколько примеров на эту тему для разных видов параметров.

Выражение `f1().DisplayMoney()` является совершенно правильным, ведь функция `f1()` возвращает объект типа `TMoney`. Поэтому в данном случае, как обычно, создается временный анонимный объект, для которого и вызывается метод вывода на экран `DisplayMoney()`.

Деструктор

При создании объекта работают конструкторы, которые в любом классе всегда присутствуют: либо создаются системой по умолчанию, либо явно определяются программистом. Однако объекты не только «рождаются», но и «умирают». Поэтому «для симметрии» в классе нужно иметь средства, которые задействуются при уничтожении объекта. В C++ есть стандартный механизм, предназначенный

как раз для таких целей, — *деструктор*. Конструктор создает объект, деструктор его уничтожает (см. п. 12.4 в [1]).

Деструктор — это метод, имеющий имя класса, но первым символом должен быть символ ~ (тильда). Деструктор не возвращает результат и не имеет параметров. Деструктор в классе может быть только один — перегрузка деструкторов не разрешается. Деструктор вызывается автоматически *при уничтожении любого объекта* своего класса. Для локальных объектов это делается при выходе из блока, а для динамических, которые клиент создает посредством операции `new` (или `new[]`), — при каждом вызове операции `delete` (или `delete[]`).

Если деструктор не определен явно, он создается системой автоматически, как и конструктор по умолчанию. Деструктор по умолчанию имеет вид

```
~class() {}
```

Во всех примерах, которые мы рассматривали, явный деструктор нам не потребовался. Обычно деструктор явно реализуется только тогда, когда при уничтожении требуется выполнить некоторые специальные действия (например, закрыть файл, открытый в конструкторе).

Конструкторы и константы

Внимательный читатель заметил, что мы ни в одном из примеров не объявляли констант. Более того, при рассказе о передаче параметров в функции не было ни одного примера с передачей параметра по константной ссылке. Это не случайно: попытки вывода константы на экран посредством метода `DisplayMoney()` вызывают ошибки компиляции! Разберемся в этом вопросе подробнее.

Конструкторы обеспечивают привычную форму объявления констант; все переменные, которые мы объявляли инициализированными, можно объявить константами, указав перед ними слово `const`, например:

```
const TMoney d2(100,67);  
const TMoney d0 = 100;  
const TMoney d5 = TMoney(100);  
const TMoney d3(d2);
```

Как видим, это фактически совпадает с объявлением констант встроенных типов. Однако вывести константу на экран не удастся! Например:

```
d2.DisplayMoney();
```

Этот оператор вызывает ошибку трансляции C2662¹:

```
cannot convert 'this' pointer from 'const TMoney' to 'TMoney &'
```

Это сообщение говорит о невозможности преобразовать указатель `this` из типа `const TMoney` в тип `TMoney &`.

¹ В Visual C++.NET 2003.

Для вывода констант на экран надо объявить *константный метод* (см. п. п. 9.3.1/3 в [1]). Константный метод — это не метод, возвращающий константу в качестве результата, а метод, который можно вызывать для объекта-константы «своего» класса. Объявление константного метода делается просто: надо указать слово `const` в заголовке функции *после* списка параметров, но перед телом, например:

```
class T
{   T f(void);           // неконстантный метод
    T f(void) const;     // константный метод
};
```

Константные и неконстантные методы не являются эквивалентными, даже если у них полностью совпадают прототипы (кроме слова `const` после списка параметров). В классе можно иметь и константный, и неконстантный метод с одинаковым прототипом, так как константный метод отличается по типу от аналогичного неконстантного метода.

Кандидатами в константные методы являются методы, не изменяющие состояние объекта, то есть не присваивающие новых значений полям класса. Компилятор следит за этим, поэтому при обнаружении явных операторов изменения полей (присваивание и ввод значений) немедленно выдает сообщение об ошибке. Конечно, компилятор, как всегда, можно «обмануть» разными способами (например, за счет косвенного доступа по указателю), но не стоит этого делать — зачем тогда объявлять метод константным?

Константные методы могут работать как с константными объектами, так и с обычными объектами-переменными. Поэтому при разработке классов нужно определить, какие методы не будут изменять состояние полей класса, и сделать эти методы константными. В нашем случае это, очевидно, метод `DisplayMoney()`. Однако он использует приватную неконстантную функцию `toString()`, поэтому сделать его константным, просто указав ключевое слово `const` в заголовке, не получится — надо делать константной и функцию `toString()`. Таким образом, прототипы (и заголовки) этих методов будут такими:

```
string toString() const;
void DisplayMoney() const;
```

После этого на экран выводятся и константы, и переменные. Теперь мы можем использовать этот метод в функции, которой параметр передается по константной ссылке. Изменим прототип показанной ранее функции `f1()` (см. листинг 2.5):

```
TMoney f1(const TMoney &t);
```

Однако теперь протесты компилятора вызывает оператор

```
t = t.AddMoney(1.0);
```

И это естественно, так как константе нельзя присваивать значение. Изменим нашу функцию так:

```
TMoney f1(const TMoney &t = f1(100))    // вызов определяемой функции
{   cout << "f1="; t.DisplayMoney();
    TMoney r = t; r = r.AddMoney(1.0);
    return r;
}
```

Теперь все работает точно так же, как и в приведенном ранее примере.

Собственно, все методы класса TMoney, кроме Init() и Read(), можно сделать константными, так как они не изменяют поля текущего объекта.

Константы в классе

До сих пор мы писали в классах обычные поля-переменные. Но иногда бывает необходимо задать в классе поле-константу. Однако инициализировать поля «в лоб» непосредственно в классе (и в структуре) запрещено. Тем не менее есть несколько способов объявить константу в классе (см. п. п. 9.4.2 в [1]). Во-первых, в классе разрешается объявить перечислимый тип с помощью ключевого слова enum. Классический пример — класс для работы с датами, в котором объявлен перечислимый тип для месяцев:

```
class TDate
{ public:
    enum Month {Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
};
```

Обращение к таким константам осуществляется либо по имени объекта, либо по имени класса, например:

```
cout << TDate::Jan << endl;           // квалификатор - класс
TDate dd;
cout << dd.Dec << endl;               // квалификатор - объект
```

Другой способ объявить константу в классе — объявить и проинициализировать *статическое поле* целочисленного типа (см. п. п. 9.4.2/4 в [1]). Для статических константных целочисленных полей (и только для них) сделано исключение: такие поля разрешено инициализировать непосредственно в классе (листинг 2.6).

Листинг 2.6. Объявление констант в классе

```
class TConstant
{ public:
    static const int      c1 = 7;
    static const char     c2 = 'a';
    enum Month
    { Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
    static const Month     c3 = Dec;
};
```

Обращение к статическим константам обычно делается по имени класса, например:

```
cout << TConstant::c1 << endl;        // выводит 7
cout << TConstant::c2 << endl;        // выводит 'a'
cout << TConstant::c3 << endl;        // выводит 12
```

Статические константы существуют в единственном экземпляре независимо от того, сколько объектов данного типа объявлено. Такие константы являются частью класса, но не являются частью объекта этого класса. Кроме того, такие

константы так же, как и константы перечислимого типа, *не занимают места в классе*. В этом легко убедиться, как всегда, с помощью функции `sizeof()`:

```
cout << sizeof(TConstant) << endl;
```

Класс `TConstant` занимает столько же памяти, сколько пустой класс, не содержащий ни одного поля.

Инициализация константных полей

Статические константы, объявляемые в классе, так же, как и константы перечислимого типа, могут быть *только целочисленного типа* (integral type). К целочисленным типам относятся булев тип, символьные, целые и перечислимые типы. Это значит, что таким способом невозможно объявить ни дробную константу, ни тем более константу более сложного типа, определенного пользователем. Кроме того, таким способом нельзя объявлять нестатические константы. Тем не менее иногда требуется иметь в классе константу нецелочисленного типа. Инициализацию таких полей-констант должен выполнять конструктор. Однако если мы попробуем в теле конструктора присвоить значение полю-константе, получим, естественно, сообщение об ошибке. Для таких случаев используется *список инициализации конструктора* (см. п. п. 12.6.2 в [1]). Список инициализации задается сразу после списка параметров; в начале списка ставится двоеточие, отделяющее его от параметров; инициализаторы полей пишутся через запятую (листинг 2.7).

Листинг 2.7. Инициализация константных полей в конструкторе

```
class TConstants
{
    const int a;
    const float t;
public:
    TConstants(const float &r = 0)    // заголовок конструктора
        :t(r), a(1)                // список инициализации
    { };                            // тело конструктора
};
```

Как видно, один инициализатор представляет собой имя поля-константы, вслед за которым в скобках указывается инициализирующее выражение. При задании списка инициализации надо помнить о нескольких простых правилах:

- ❑ Инициализировать таким образом можно не только константные поля, но и обычные поля-переменные.
- ❑ Независимо от порядка перечисления полей в списке инициализации поля получают значения в порядке объявления в классе. Это означает, что для инициализации следующих полей можно использовать уже инициализированные поля.
- ❑ В качестве инициализирующего выражения можно задать любое вычисляемое выражение (не обязательно константное), результатом которого является значение того же (или приводимого) типа, что и тип инициализируемого поля. Это, в частности, означает, что можно использовать функции, как стандартные, так и определяемые программистом. Для полей типа класса выполняется вызов конструктора.

- ❑ Для инициализации поля встроенного типа *нулем* может быть использована специальная форма инициализатора `имя_поля()` — выражение не задается. Для поля-объекта класса это означает *вызов конструктора по умолчанию*.
- ❑ На месте инициализирующего выражения в скобках можно, как обычно, через запятую задать список выражений, последнее выражение должно иметь тип поля или приводиться к этому типу.
- ❑ Список инициализации выполняется *до начала выполнения тела* конструктора.

Немного модифицируем класс `TConstants`, чтобы продемонстрировать использование выражений в качестве инициализаторов (листинг 2.8).

Листинг 2.8. Выражения в списке инициализации

```
class TConstants
{
    const int a;
    double b;
    const float t;
    float ff;
public:
    TConstants(const float &r = 0)
        :t(ff=a*r),           // выражение
        a(floor(r)),         // функция
        b()                  // инициализация нулем
        { };
};
```

Порядок инициализации в данном примере следующий:

1. Инициализируется целое константное поле `a`, так как это поле прописано в классе первым. Для инициализации вызывается функция `floor()`.
2. Обнуляется поле `b`.
3. Значение `a` используется в выражении для инициализации константного поля `t`. При вычислении инициализирующего выражения выполняется операция присваивания и инициализируется неконстантное поле `ff`.

Немного более «объемный» пример, показанный в листинге 2.9, демонстрирует практически все правила работы списка инициализации.

Листинг 2.9. Вызов конструктора в списке инициализации

```
class TDate
{
    int year, month, day;           // год, месяц, день
    int LastDay(int m)              // функция, вычисляющая последний день месяца
    {
        switch(m)
        {
            case 1:case 3: case 5:case 7: case 8: case 10: case 12: return 31;
            case 4: case 6: case 9: case 11: return 30;
            case 2: return ((year%100)&&(year%4==0)|| (year%400==0)? 29:28);
            default: return 0;
        }
    }
};
```

Листинг 2.9 (продолжение)

```

public:
    TDate (int y = 1, int m = 1)
    :year(y),           // простая инициализация
      month((0<m)&&(m<13)?m:0), // условное выражение
      day(LastDay(m))    // вызов функции
    { cout << "TDate " << endl; }
    void DisplayDate()    // вывод на экран
    { cout << year << ' ' << month << ' ' << day ;
    }
};

class TPerson
{
    const int N;           // номер счета
    TDate d;               // дата открытия
    string name;           // фамилия
public:
    TPerson(const char *n, const int &N)
    :N(N), name(n),
      d()                  // вызов конструктора без аргументов
    { cout << "TPerson " << endl; // тело конструктора
    }
    void DisplayPerson()    // вывод на экран
    { cout << name << ' ' << N << ' ';
      d.DisplayDate();
    }
};

int main()
{
    TPerson I("LaptevVV", 54321); // инициализация объекта
    I.DisplayPerson();
    return 0;
}

```

В этом примере сначала объявлен класс `TDate`, в котором определен конструктор с двумя параметрами, задаваемыми по умолчанию. Инициализация полей выполняется в списке инициализации. Году (`year`) просто присваивается значение параметра, значение месяца (`month`) проверяется на допустимость, а значение дня (`day`) вычисляется приватной функцией.

В классе `TPerson` продемонстрирован неявный вызов конструктора `TDate` в списке инициализации для поля `d`. Отладочный вывод на экран показывает, что сначала вызывается конструктор для присвоения значения полю `d`, а затем уже выполняется тело конструктора `TPerson`.

Вызов конструктора можно задать и явным образом, например:

```
d(TDate(1628,11))
```

СОВЕТ

Всегда, когда возможно, инициализируйте поля класса в списке инициализации.

Список инициализации довольно интенсивно используется при наследовании (см. главу 8).

Резюме

Конструктор — это специальный метод класса, имя которого совпадает с именем класса. Конструктор выполняется каждый раз при создании нового объекта. Конструктор не возвращает значений, но может принимать параметры, которые обычно используются для инициализации полей класса. Конструкторы допускают перегрузку, поэтому инициализация объекта может выполняться несколькими способами. Конструкторы бывают трех видов: конструктор без аргументов, конструктор копирования с одним параметром-ссылкой на объект такого же типа и конструктор инициализации. Если аргументы конструктора не объявлены, то автоматически создается конструктор без аргументов и конструктор копирования. Если объявлен хоть один конструктор, то конструктор без аргументов автоматически не создается. Конструктор копирования создается всегда, если он не определен явно.

Конструкторы позволяют естественным образом объявлять объекты класса с инициализацией. Конструкторы позволяют инициализировать и динамические переменные. При объявлении массива объектов конструктор вызывается для каждого элемента массива. Конструкторы могут неявно вызываться и при передаче параметров в функцию, и при возврате результата.

Еще один специальный метод — деструктор. Деструктор в классе может быть только один — перегрузка деструкторов запрещена. Деструктор не имеет параметров, не возвращает значение. Имя деструктора совпадает с именем класса, только начинается с символа ~ (тильда). Конструктор создает объект, деструктор его уничтожает. Если деструктор не определен явно, то он создается системой автоматически.

В классе можно объявлять константные поля, инициализацию которых можно выполнить только посредством списка инициализации конструктора. Список инициализации выполняется до начала выполнения тела конструктора. Поля инициализируются в порядке объявления в классе, а не в порядке перечисления в списке инициализации. В качестве инициализирующего выражения для элемента инициализации в списке допускается использовать любое выражение, приводимое к типу инициализируемого поля. При отсутствии инициализирующего выражения для встроенных типов выполняется обнуление (инициализация нулем), для объектов невстроенных типов вызывается конструктор без аргументов.

В классе можно задавать целочисленные константы двумя способами: как перечислимые и как статические. Объявленные константы не занимают места в классе.

В классе могут быть объявлены константные методы. Для объекта-константы может быть вызван только константный метод. Константный метод не изменяет полей класса. Константный метод отличается от аналогичного неконстантного метода.

Контрольные вопросы

1. Дайте определение конструктора. Каково назначение конструктора?
2. Перечислите отличия конструктора от метода.

3. Сколько конструкторов может быть в классе? Допускается ли перегрузка конструкторов?
4. Какие виды конструкторов создаются по умолчанию?
5. Может ли конструктор быть приватным? Какие последствия влечет за собой объявление конструктора приватным?
6. Приведите несколько случаев, когда конструктор вызывается неявно.
7. Как инициализировать динамическую переменную?
8. Как объявить константу в классе? Можно ли объявить дробную константу?
9. Каким образом разрешается инициализировать константные поля в классе?
10. В каком порядке инициализируются поля в классе? Совпадает ли этот порядок с порядком перечисления инициализаторов в списке инициализации конструктора?
11. Какие конструкции C++ разрешается использовать в списке инициализации в качестве инициализирующих выражений?
12. Влияет ли наличие целочисленных констант-полей на размер класса?
13. Объясните, что такое «инициализация нулем».
14. Что такое деструктор? Может ли деструктор иметь параметры? Допускается ли перегрузка деструкторов?
15. Зачем нужны константные методы? Чем отличается определение константного метода от определения обычного?
16. Может ли константный метод вызываться для объектов-переменных? А обычный метод — для объектов-констант?

Упражнения

Во всех упражнениях, помимо указанных методов, обязательно должны быть реализованы набор необходимых конструкторов, метод ввода данных с клавиатуры `Read()`, метод вывода данных на экран `Display()`. Конструкторы должны проверять корректность задаваемых значений параметров. Подходящие методы должны быть реализованы как константные.

Для демонстрации работы с объектами нового типа во всех заданиях требуется написать главную функцию.

1. Реализовать класс `Power` с двумя полями: `first` и `second`. Поле `first` — дробное число; поле `second` — дробное число, показатель степени. Реализовать конструктор без аргументов, присваивающий полям нулевые значения, и конструктор инициализации, присваивающий второму полю значение 0 по умолчанию. Реализовать метод `power()` — возведение числа `first` в степень `second`.
2. Создать класс `Double`, имитирующий стандартный тип `double` (см. упражнение 4 в главе 1), реализовав подходящий конструктор.

3. Создать класс `LongInteger` с двумя полями: `high` и `low`, представляющими собой старшую и младшую части числа. Реализовать конструкторы инициализации целым любого типа со значениями по умолчанию и строкой цифр. Реализовать методы сравнения на больше и на равенство, а также метод сложения. Реализовать функцию преобразования в строку `toString()` (см. листинг 1.6).
4. Реализовать класс `Date` с тремя полями: день, месяц и год. Реализовать три конструктора инициализации: с тремя целыми аргументами (минутам и секундам присвоить значения по умолчанию), строкой вида «год.месяц.день» (например, «2006.04.07»), одним целым числом вида ггггммдд. Для представления месяцев определить в классе перечислимый тип `month`. В классе реализовать методы получения и изменения отдельных полей даты, метод вычисления даты через заданное количество дней, метод вычисления количества дней между датами.
5. Создать класс `Time` для работы со временем в формате «час:минута:секунда». Класс должен включать в себя три конструктора инициализации: числами, строкой (например, «23:59:59»), секундами. Конструктор инициализации тремя числами должен присваивать значения по умолчанию минутам и секундам. Реализовать операции вычисления разницы между двумя моментами времени в секундах, сложения времени и заданного количества секунд, вычитания из времени заданного количества секунд, сравнения моментов времени. Реализовать функцию преобразования в строку `toString()` (см. листинг 1.6).
6. Реализовать класс `Account` (см. упражнение 12 в главе 1), используя класс `Money` (см. листинг 1.15) для представления суммы счета и класс `Date` (см. упражнение 4) для представления даты. Реализовать необходимые конструкторы инициализации.
7. Реализовать класс `Fraction` (см. упражнение 5 в главе 1), используя структуру `Pair`. Структура включает в себя два поля подходящего типа: `first` и `second`. Обеспечить реализацию конструктора инициализации структуры `Pair` и конструкторов инициализации класса `Fraction` с присвоением значений по умолчанию. Реализовать все арифметические операции и операции сравнения. Реализовать функцию преобразования в строку `toString()` (см. листинг 1.6).
8. Реализовать в классе `Decimal` из упражнения 6 в главе 1 конструкторы инициализации числами любого целого типа и строкой.
9. Создать класс `BitString` для работы с битовыми строками длиной 64 бита. Битовая строка должна быть представлена двумя полями типа `unsigned long`. Конструкторы инициализации должны обеспечивать инициализацию полей целыми числами любого типа и строкой типа `string`. Реализовать традиционные операции для работы с битами (`and`, `or`, `xor`, `not`).
10. Рациональная (несократимая) дробь представляется парой целых чисел (a, b), где a — числитель, b — знаменатель. Создать класс `Rational` для работы с рациональными дробями. Реализовать конструкторы инициализации с параметрами по умолчанию. Один из конструкторов должен принимать

параметр-структуру с двумя полями, `first` и `second`, целого типа. Обязательно должны быть реализованы операции:

○ сложения `add`:

$$(a, b) + (c, d) = (ad + bc, bd);$$

○ вычитания `sub`:

$$(a, b) - (c, d) = (ad - bc, bd);$$

○ умножения `mul`:

$$(a, b) \cdot (c, d) = (ac, bd);$$

○ деления `div`:

$$(a, b) / (c, d) = (ad, bc);$$

○ сравнения `equal` (равно), `greate` (больше), `less` (меньше).

Должна быть также реализована приватная функция сокращения дроби `reduce`, которая обязательно вызывается при выполнении арифметических операций.

Глава 3

Перегрузка операций

В главе 1 мы сконструировали класс, позволяющий нам оперировать денежными суммами. Мы можем использовать тип `TMoney` почти так же, как и встроенные типы. Однако реализация операций не совсем удовлетворительна. Например, сложение было у нас реализовано следующим образом:

```
a=b.AddMoney(d);
```

Согласитесь, что сложение денежных сумм более естественно записать так:

```
a=b+d;
```

Эту проблему можно решить за счет *перегрузки операций* (см. п. 13.5 в [1]). Перегрузка операций — это развитие механизма перегрузки функций и еще одно проявление *полиморфизма* в C++.

Нужно сказать, что в C++ мы постоянно пользуемся перегруженными операциями. Символ `+` (плюс) обозначает операцию сложения целых и дробных чисел. Этот же символ используется для обозначения операции сцепления строк типа `string`. Для вывода денежных сумм на экран мы использовали операцию `<<`, которая определена еще в C как операция сдвига влево. А операция присваивания по умолчанию перегружена вообще для всех типов данных, как встроенных, так и определяемых.

C++ во многом ограничивает перегрузку операций. Во-первых, разрешается перегружать только встроенные (табл. 3.1) операции (см. п. 5 в [1]) — новых определять нельзя. Во-вторых, даже встроенные операции разрешено перегружать не все. Запрещена перегрузка следующих операций:

- селектор компонента объекта (`.`);
- разыменование указателя на компонент класса (`.*`);
- условная операция (`? :`);
- указание области видимости (`::`);
- определение размера аргумента (`sizeof`).

Не допускается также перегрузка операций препроцессора # и ##.

В-третьих, операции можно перегружать только для нового типа данных — нельзя перегрузить операцию для встроенного типа. Например, нельзя перегрузить операцию ^ как операцию возведения в степень ни для double, ни для другого стандартного встроенного числового типа. Чтобы определить операцию возведения в степень, нам придется сначала объявить новый тип данных. В C++ новый тип данных можно образовать с помощью конструкций enum, union, struct и class.

Таблица 3.1. Операции C++ в порядке убывания приоритетов

Операция	Назначение	Использование
::	Область видимости	::name class::name namespace::name
	Селектор компонента класса	object.member
->	Доступ к члену класса по указателю	pointer -> member
[]	Индексирование	variable[expr]
()	Вызов функции	function()
++	Инкремент постфиксный	lvalue++
--	Декремент постфиксный	lvalue--
sizeof	Размер объекта	sizeof expr
sizeof	Размер типа	sizeof(type)
++	Префиксный инкремент	++lvalue
--	Префиксный декремент	--lvalue
~	Побитовое НЕ	~expr
!	Логическое НЕ	!expr
-	Унарный минус	-expr
+	Унарный плюс	+expr
*	Разыменование	*expr
&	Адрес	&expr
()	Приведение типа	(type)expr
new	Выделение памяти	new type new type(exprlist)
new[]	Выделение памяти под массив	new type[]
delete	Освобождение памяти	delete pointer
delete[]	Освобождение памяти из-под массива	delete[] pointer
->*	Разыменование указателя на компонент класса по указателю	pointer->*pointer_to_member
.*	Разыменование указателя на компонент класса	object.*pointer_to_member

Операция	Назначение	Использование
*	Умножение	expr * expr
/	Деление	expr / expr
%	Деление по модулю	expr % expr
+	Сложение	expr + expr
-	Вычитание	expr - expr
<<	Сдвиг влево	expr << expr
>>	Сдвиг вправо	expr >> expr
<	Меньше	expr < expr
<=	Меньше или равно	expr <= expr
>	Больше	expr > expr
>=	Больше или равно	expr >= expr
==	Равно	expr == expr
!=	Не равно	expr != expr
&	Побитовое И	expr & expr
^	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ	expr ^ expr
	Побитовое ИЛИ	expr expr
&&	Логическое И	expr && expr
	Логическое ИЛИ	expr expr
?:	Условная операция	expr ? expr * expr
=	Присваивание	L-значение = expr
*, /=, %=, +=, -=, <=, >=, &=, =, ^=	Составное присваивание (относительный порядок приоритетов соответствует порядку приоритетов операций без присваивания)	L-значение += expr и т. д.
,	Запятая	expr, expr

Операции можно перегружать либо как независимые внешние функции (и только такой способ перегрузки возможен для `enum`), либо как методы класса. Но и тут C++ имеет свои ограничения. Четыре операции допускается перегружать только методами класса:

- ☐ присваивание (`=`);
- ☐ вызов функции (`()`);
- ☐ индексирование (`[]`);
- ☐ доступ по указателю (`->`).

Причем операция присваивания единственная создается системой автоматически, если не определена программистом явно. Таким образом, эти операции в принципе нельзя перегрузить для конструкции `enum`. Однако и без перегрузки эти операции почти всегда корректно работают с новым типом. Остальные

операции допускается перегружать как методами класса, так и независимыми внешними функциями.

Прототип функции-операции выглядит следующим образом:

```
тип operator@(список параметров);
```

Здесь @ — символ операции. Слово `operator` является зарезервированным словом и может использоваться только в определении или в функциональной форме вызова операции. Синтаксис и приоритет перегружаемой операции изменить нельзя. Это означает, что унарную операцию нельзя сделать при перегрузке бинарной и наоборот. Следовательно, при перегрузке унарных и бинарных операций *нельзя* использовать параметры по умолчанию. При перегрузке таких операций нет никакой возможности задать список параметров переменной длины. Единственная операция, которая не имеет фиксированного количества аргументов, — это операция вызова функции `operator()`. Остальные операции должны иметь либо один, либо два аргумента. Операции `+`, `-`, `*` и `&` допускается перегружать и как унарные, и как бинарные.

Перегрузка операций внешними функциями

Прототип бинарной операции, перегружаемой независимой внешней функцией, выглядит так:

```
тип operator@(параметр_1, параметр_2);
```

Естественно, параметры можно передавать любым удобным способом, но наиболее часто аргументы передаются по константной ссылке. Обращение к определенной таким образом операции выполняется двумя различными способами:

□ инфиксная форма:

```
параметр_1 @ параметр_2;
```

□ функциональная форма:

```
operator@(параметр_1, параметр_2);
```

Прототип унарной операции, перегружаемой независимой внешней функцией, отличается только количеством параметров:

```
тип operator@(параметр);
```

Обращение же к перегруженной операции выглядит так:

□ инфиксная форма:

```
@параметр;
```

□ функциональная форма:

```
operator@(параметр);
```

Покажем перегрузку для перечислимого типа `enum`. Где могут понадобиться такие операции? Ну, например, при реализации класса для работы с датами месяца обычно представляются как перечислимый тип данных:

```
enum Month {Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

При сложении месяца с целой константой надо отслеживать переход через год — после декабря должен наступить январь (следующего года). Аналогично, если мы вычитаем единицу из января, то должны получить декабрь (предыдущего года). Собственно говоря, операции должны выполняться по модулю 12. Покажем реализацию операции сложения месяца с числом (листинг 3.1). Операций должно быть реализовано две, чтобы обеспечить коммутативность.

Листинг 3.1. Перегрузка операций для перечислимого типа

```
enum Month {Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
Month operator+(const Month &m, const int &b)
{
    Month t = Month(b + m);
    return (t = Month((t>Dec)?t-Dec:t));
}
Month operator+(const int &b, const Month &m)
{
    return (m+b);
}
int main()
{
    Month m = Dec;
    cout << m+2 << endl;           // первая операция выдает 2
    cout << 1+m << endl;           // вторая операция выдает 1
    return 0;
}
```

Обратите внимание на то, что порядок следования слагаемых в скобках обратный относительно их порядка следования в списке параметров:

```
Month t = Month(b + m);
```

При «нормальном» порядке следования возникает ошибка при выполнении — переполнение стека:

```
Month t = Month(m + b);
```

Дело в том, что в этом случае получается рекурсивный вызов операции сложения — типы аргументов-то как раз такие, как в заголовке. Если же порядок обратный, то работает встроенная операция сложения, и рекурсии нет.

Можно использовать «нормальный» порядок следования слагаемых, если «обмануть» компилятор за счет преобразования типа, например:

```
Month t = Month(int(m) + b);
```

В этом случае тоже выполняется встроенная операция сложения целых вместо определяемой. Обратите внимание, как мы опять использовали первую операцию для реализации второй — еще раз подчеркнем, что это — очень полезный прием.

Перегрузку внешними функциями для класса покажем на примере нашей структуры TMoney и функции сложения (листинг 3.2).

Листинг 3.2. Перегрузка операции сложения для структуры внешней функцией

```
struct TMoney
{
    long double Summa;           // денежная сумма
};
TMoney operator+(const TMoney &a, const TMoney &b)
{
    TMoney t = a;
```


Листинг 3.2 (продолжение)

```

    t.Summa += b.Summa;
    return t;
}

```

Как видите, эта функция отличается от определенной нами ранее функции `AddMoney()` (см. листинг 1.4) только именем. После такого определения мы можем выполнять операцию сложения либо так:

```

TMoney a, b, c;
// ...
a = b + c;                // инфиксная форма

```

либо так:

```

a = operator+(b,c);       // функциональная форма

```

Второй вариант вызова операции сложения представлен в функциональной форме.

Перегрузка операций методами класса

Остальные операции со структурой перегружаются аналогичным образом, поэтому не будем на этом останавливаться — нас больше интересует перегрузка операций как методов класса. Вернемся к нашему классу `TMoney`. Хотя методов у нас было немного, операций мы можем реализовать гораздо больше, что позволит оперировать денежными суммами более естественными способами. Во-первых, перегрузим в виде операций уже реализованные нами методы сложения, вычитания, умножения и деления. Помимо обычных операций, реализуем те же операции с присваиванием, а также инкремент и декремент. Во-вторых, вместо одного метода сравнения лучше реализовать обычные операции сравнения. И в-третьих, операции ввода-вывода тоже лучше иметь в привычном для C++ виде.

Как мы помним, у методов один параметр — текущий объект — определен по умолчанию. Поэтому унарные операции выполняются для текущего объекта, который и является единственным аргументом. Следовательно, при перегрузке как методы класса они не имеют параметров. Заголовок унарной операции, реализованной как метод класса, выглядит так:

```
тип operator@()
```

Здесь @ — символ операции.

ПРИМЕЧАНИЕ

Постфиксные операции инкремента и декремента являются исключением из этого правила. Для того чтобы отличить постфиксную операцию от префиксной, в постфиксной операции прописывают фиктивный аргумент, который реально не используется.

Заголовок бинарной операции, соответственно, имеет один аргумент и выглядит так:

```
тип operator@(параметр)
```

Параметры тем не менее разрешается передавать любым удобным нам способом: по значению, по ссылке или по указателю. Вызов унарной операции для объекта `t` выглядит так:

```
@t
t@
```

Вариант выбирается в зависимости от того, какой вид операции определен: префиксный или постфиксный. Функциональная форма вызова выглядит одинаково для обоих вариантов:

```
t.operator@()
```

Функциональная форма вызова для бинарной операции, определенной как метод класса, выглядит аналогично:

```
t.operator@(p)
```

Такая запись подчеркивает, что у бинарной операции, перегружаемой методом класса, только один параметр, так как левым аргументом *обязательно* должен быть объект определяемого класса. Инфиксная форма:

```
t @ p
```

Эта форма фактически является просто сокращением функциональной — удалены конструкция `.operator` и скобки вокруг второго аргумента.

Рассмотрим реализацию операции сложения денежных сумм. Внешнее определение, представленное в листинге 3.3, ничем, за исключением заголовка, не отличается от реализованного нами ранее метода `AddMoney()` (см. листинг 1.12), в котором аргументом является тоже объект типа `TMoney`.

Листинг 3.3. Метод-операция сложения денежных сумм

```
TMoney TMoney::operator+(const TMoney &b)
{
    TMoney t = *this;
    t.Summa += b.Summa;           // работает встроенная операция
    return t;
}
```

Теперь мы, наконец, сможем писать сложение денежных сумм в привычном виде:

```
a = b + c;
```

Вывод суммы на экран можно осуществлять, не сохраняя результат в переменной:

```
(t+p).DisplayMoney();
```

Выглядит необычно, но перепишем вызов операции в функциональной форме:

```
(t.operator+(p)).DisplayMoney();
```

Совершенно очевидно, что это выражение эквивалентно описанному ранее в листинге 1.17 (помечено цифрой 1):

```
(t.AddMoney(p)).DisplayMoney();
```

Перегрузим теперь операцию сложения с присваиванием (`+=`). Желательно, чтобы семантика перегруженной операции совпадала с семантикой встроенной:

программист в этом случае совершает меньше ошибок при ее использовании. Напомним семантику встроенной операции: значение правого аргумента складывается со значением левого аргумента, и результат опять помещается в левый аргумент. Так как операция бинарная, то в заголовке должен быть указан один параметр. Левым аргументом у нас является текущий объект, а правым — параметр метода-операции. Таким образом, эта операция должна изменять значения полей текущего объекта.

Единственный вопрос, который пока не ясен: что возвращать в качестве результата. Попробуем возвращать значение. Тогда внешнее определение метода будет таким, как показано в листинге 3.4.

Листинг 3.4. Реализация операции сложения с присваиванием

```
TMoney TMoney::operator+=(const TMoney &b)
{
    Summa += b.Summa;
    return *this;
}
```

В качестве результата возвращается текущий объект. Проверим теперь различные варианты использования операции. Простой однократный вариант, естественно, работает правильно:

```
TMoney t(2), p(3.12), s;
t+=p; t.DisplayMoney();
```

При выполнении этого фрагмента мы получим на экране сумму *t* и *p*. Однако встроенная операция допускает запись следующего вида:

```
t+=b+=c;
```

При этом переменной *b* присваивается сумма переменных *b* и *c*, а в переменную *t* попадает сумма прежнего значения *t* и нового значения *b*. Проверим нашу операцию на многократность присваивания — оказывается не все так просто. Для сравнения покажем работу встроенной операции с целыми числами:

```
TMoney t(2.00), rr(0);
rr+=t+=t; // должно быть t = 4.00, rr = 4.00
rr.DisplayMoney(); // выводит 4.00
t.DisplayMoney(); // выводит 4.00
t = 2; rr = 0;
(rr+=t)+=t; // должно быть t = 4.00, rr = 4.00
rr.DisplayMoney(); // выводит 2.00
t.DisplayMoney(); // выводит 2.00
int a = 0, d = 2;
(a+=d)+=d; // должно быть d = 2, a = 4
cout << a << endl; // выводит 4
cout << d << endl; // выводит 2
```

Как видим, наша операция выполняется не так, как встроенная: значение самой правой переменной в левую переменную не попадает, если слева стоят скобки. При этом выражение без скобок работает совершенно правильно:

```
rr+=t+=t; // rr должно стать 4.00
```

Это и понятно: операция `+=` выполняется справа налево, поэтому сначала вычисляется выражение `t+=t`, а потом полученное значение попадает в `гг`.

Чтобы разобраться в проблеме, вернемся к проблемному выражению:

```
(гг+=t)+=t;
```

Давайте перепишем его в функциональной форме:

```
(гг.operator+=(t)).operator+=(t);
```

Сначала выполняется операция в скобках — значение переменной `t` прибавляется к `гг`. Результатом операции в скобках является *значение*, помещаемое в переменную `гг`. Поэтому создается анонимный временный объект, с которым выполняется правая операция. Чтобы результат попадал в самую левую переменную, результатом операции должна быть *ссылка*. Таким образом, выражение `(a+=b)` может стоять в левой части другой операции `+=`. Такие выражения в C++ называются *левосторонними значениями*, или *l-значениями* (l-value). Хотя в стандарте (см. п. 3.10 в [1]) этому понятию уделено более страницы, нам достаточно понять, что любое выражение, которое может стоять слева от знака любой операции присваивания, называется l-значением. Соответственно, выражение, которому нельзя присвоить и которое может стоять только справа от знака операции присваивания, называется r-значением (r-value). Собственно, ссылки (см. п. п. 8.3.2 в [1]) были добавлены в C++ во многом именно для обеспечения возможности перегрузки операций присваивания [43].

Таким образом, правильный заголовок метода-операции в данном случае должен быть таким¹:

```
TMoney& TMoney::operator+=(const TMoney &b)
```

Реализация операции остается без изменений. Вообще говоря, любая операция, возвращающая неконстантную ссылку, является многократной (повторяемой). Но повторное использование можно и явным образом запретить, если возвращать константную ссылку, например:

```
const TMoney& TMoney::operator+=(const TMoney &b)
```

Модификатор `const` запрещает дальнейшую модификацию результата.

С помощью операции `+=` можно реализовать «чистое» сложение (листинг 3.5).

Листинг 3.5. Реализация операции `+` с помощью операции `+=`

```
TMoney TMoney::operator+(const TMoney &b)
{
    TMoney t=*this;
    t+=b;                // работает реализованная операция
    return t;
}
```

Аналогично реализуются операции вычитания, поэтому текст этих функций приводить не будем.

¹ Вспомните прототип операции присваивания, приведенный в главе 1.

Конечно, в данном случае это — менее эффективное решение, чем приведенное в листинге 3.2. Однако для более «сложных» классов, чем денежный, использование реализованного метода по разным причинам может оказаться выгоднее применения встроенных операций. Опытные программисты всегда так поступают: реализуют одну операцию «вручную», а родственные — на основе уже реализованной. Таким способом снижается вероятность совершить ошибку, так как использование уже реализованной операции упрощает реализацию остальных. Кроме того, при возможных дальнейших изменениях операции сложения с присваиванием «чистую» операцию сложения изменять не потребуется.

СОВЕТ

При перегрузке операций для нового класса реализуйте сначала минимальный набор операций, а для реализации дополнительных операций используйте уже работающие.

Реализуем теперь операции деления (/) и деления с присваиванием (/=). Есть два варианта деления денежных сумм: деление денег на деньги и деление денег на дробную константу. Первый вариант деления имеет результатом дробную константу и не изменяет значение левого аргумента (текущего объекта), поэтому не может быть делением с присваиванием. Следовательно, деление с присваиванием — это деление на дробную константу. Как мы уже знаем, возвращаемым результатом должна быть ссылка на объект.

«Чистая» операция деления тогда у нас будет перегружена: деление денег на дробную константу и деление денежных сумм. Первая реализуется с помощью операции деления с присваиванием, а вторая — независимо. Текст всех трех функций приведен в листинге 3.6.

Листинг 3.6. Методы-операции деления

```
TMoney& TMoney::operator/=(const double &b)
{
    if (b > 0)
    {
        bool negative = (Summa < 0);
        Summa = round(Summa/b);
        if (negative) Summa = -Summa;
    }
    return *this;
}

TMoney TMoney::operator/(const double &b)
{
    TMoney t = *this;
    t/=b;           // реализованная операция
    return t;
}

long double TMoney::operator/(const TMoney &b)
{
    long double t = 0.0;
    if (b.Summa!=0) t = fabs(Summa)/fabs(b.Summa);
    return t;
}
```

В «чистой» операции деления на константу мы не проверяем параметр — это делает операция деления с присваиванием, с помощью которой мы и вычисляем требуемое значение.

Операции умножения на константу отличаются от операций деления только непосредственно знаком операции, поэтому приводить текст этих функций не будем.

И наконец, реализуем операции сравнения. Можно для этого использовать метод `CompareMoney()` (см. листинг 1.15), сделав его закрытым, однако операции сравнения просты и обычно реализуются одна через другую. Результатом всех операций, естественно, будет значение типа `bool`. Тексты функций приведены в листинге 3.7.

Листинг 3.7. Операции сравнения

```
bool TMoney::operator==(const TMoney &b)
{ return (Summa == b.Summa); }           // сравниваются числа
bool TMoney::operator!=(const TMoney &b)
{ return !(*this == b); }                 // сравниваются деньги
}
bool TMoney::operator<(const TMoney &b)
{ return (Summa < b.Summa); }             // сравниваются числа
bool TMoney::operator>=(const TMoney &b)
{ return !(*this < b); }                  // сравниваются деньги
}
bool TMoney::operator>(const TMoney &b)
{ return !((*this < b)&&(*this == b)); }   // сравниваются деньги
}
bool TMoney::operator<=(const TMoney &b)
{ return (*this < b)||(*this == b); }     // сравниваются деньги
}
```

Проверка этих операций сложностей не представляет, поэтому оставляем ее читателю. Отметим только следующее: при реализации операций `==` и `<` в теле методов используются те же самые операции. Аналогичная картина наблюдается и при реализации других операций, например операции `+=` (см. листинг 3.3). Вообще-то говоря, такая запись является рекурсивным вызовом, однако в данном случае рекурсии не возникает. Это объясняется типами аргументов: реализуются операции для денежного типа, а в теле функции они используются с аргументами встроенного типа `long double`.

Нам осталось реализовать всего несколько операций: ввода-вывода, инкремента и декремента. Рассмотрим более подробно операцию инкремента. Как известно, она имеет префиксную и постфиксную формы. Семантика этих форм несколько различается. Например:

```
double d = ++x;
```

Это присваивание выполняется так:

```
x+=1;
double d = x;
```

Теперь постфиксная форма:

```
double d = x++;
```

Постфиксная форма выполняется совершенно по-другому:

```
double d = x;
x+=1;
```

Если мы заглянем в стандарт C++ (см. п. п. 13.5.7 в [1]), то обнаружим, что префиксная форма должна возвращать ссылку, а постфиксная — значение. Чтобы можно было отличить одно определение от другого, постфиксная форма операции должна иметь фиктивный аргумент типа `int`. Обе операции изменяют текущий объект. Пусть наш инкремент в префиксной форме увеличивает сумму на 1 рубль, а в постфиксной — прибавляет одну копейку. В этом случае определения операций могли бы выглядеть так, как показано в листинге 3.8.

Листинг 3.8. Операции инкремента

```
TMoney& operator++()           // префиксная форма
{   return (*this+=100);      // увеличение рублей
}
TMoney operator++(int)         // постфиксная форма
{   TMoney t = *this;
    *this+=1;                  // увеличение копеек
    return t;
}
```

Однако у нас не реализована операция сложения с присваиванием `+=` для числового аргумента. Давайте реализуем более общий вариант — сложение с дробным числом типа `long double`. Выражение `x+=1.0` тогда будет означать увеличение рублей, а выражение `x+=0.01` — увеличение копеек. Если не проверять допустимость параметра, то определить операцию большого труда не составляет, что и продемонстрировано в листинге 3.9.

Листинг 3.9. Реализация сложения с присваиванием дробного числа

```
TMoney& TMoney::operator+=(const long double &b)
{   Summa += round(b*100);    // работает встроенная операция
    return *this;
}
```

С учетом этой операции надо скорректировать соответствующие операторы в листинге 3.8:

```
*this+=1.0;           // префиксный инкремент - для рублей
*this+=0.01;          // постфиксный инкремент - для копеек
```

Проверка операции:

```
TMoney t = 2;
++t; t.DisplayMoney();
t++; t.DisplayMoney();
(++t).DisplayMoney(); // без скобок - ошибка трансляции
t++.DisplayMoney();
```

Проверка показывает, что операции делают именно то, что мы и хотели.

«Подводные камни» перегрузки операций

В главе 1 (см. листинг 1.15) мы реализовали два метода `AddMoney()`: сложение денег с деньгами и сложение денег с дробной константой, которая в этом случае

представляет собой денежную сумму-константу. Перепишем теперь «простое» сложение с одним аргументом — дробным числом (листинг 3.10).

Листинг 3.10. Операция сложения с дробным числом

```
TMoney TMoney::operator+(const long double &b)
{
    TMoney t=*this;
    t+=round(b*100);          // работает реализованная нами операция
    return t;
}
```

Обратите внимание на то, что в следующем операторе из листинга 3.9 операция += выполняется с переменными типа long double, поэтому работает встроенная операция:

```
Summa += round(b*100);      // работает встроенная операция
```

В листинге же 3.10 действия выполняются с денежными суммами, поэтому работает реализованный в классе TMoney метод-операция:

```
t+=round(b*100);           // работает реализованная нами операция
```

Полиморфизм — в действии!

Однако при проверке реализованной операции сложения неожиданно оказывается, что вторая строка вызывает ошибку трансляции:

```
s = t+2; s.DisplayMoney();
s = 2+t; s.DisplayMoney();    // ошибка трансляции!
```

Ситуацию проясняет функциональная запись. Первое выражение запишется так:

```
s = t.operator+(2);
```

Второе же выражение ошибочно, так как левый аргумент не является объектом типа TMoney:

```
s = 2.operator+(t);
```

Получается, что операция сложения, реализованная как метод класса, не является коммутативной! Точно такая же ситуация с уже реализованной нами операцией умножения на число:

```
s = t*2; s.DisplayMoney();    // работает
s = 2*t; s.DisplayMoney();    // не работает!
```

Теперь рассмотрим операции ввода-вывода¹ << и >>. Обе операции перегружают изначально определенные в языке C операции сдвига влево и вправо, поэтому операции ввода-вывода — бинарные, как и сдвиги (см. п. 5.8 в [1]). Операции ввода-вывода, очевидно, являются операциями многократными, поэтому должны возвращать ссылку. Однако самым важным является не это. Левым аргументом у операции ввода является объект cin типа istream, а у операции вывода — объект cout типа ostream. Реализовав ввод-вывод методами класса TMoney, мы не сможем этого обеспечить, поскольку левым аргументом любого метода является

¹ Подробности см. в главе 14.

объект «своего» типа. Давайте тем не менее попробуем реализовать, например, операцию вывода и посмотрим, что из этого выйдет. У функции-метода, очевидно, должен быть один явный параметр — поток вывода. Реализация метода почти совпадает с реализацией метода-функции `DisplayMoney()` за исключением того, что должна возвращаться ссылка на объект типа `ostream`. Таким образом, возможная реализация выглядит так, как показано в листинге 3.11.

Листинг 3.11. Вывод как метод класса

```
ostream& TMoney::operator<<(ostream& t)
{ string s = toString();
  s+=" руб.";
  return (t << s << endl);
}
```

Определение вопросов не вызывает, однако вызов операции вывода сбивает с толку:

```
d << cout;           // запись "наоборот"
```

Хотя операция работает совершенно правильно, это совсем не то, что мы хотели. Проблема именно в том, что левым аргументом метода должен быть текущий объект. В функциональной форме записи это отчетливо видно:

```
d.operator<<(cout);
```

Аналогичная картина наблюдается и при реализации операции ввода.

Функции-друзья класса

Вывод из всего сказанного может быть только один: не все операции должны быть реализованы как методы класса. Очевидно, что кандидаты в методы — это, в первую очередь, те операции, у которых левым аргументом является текущий объект. Например, это все операции с присваиванием. И напротив, если в операции левым аргументом может быть объект, тип которого отличается от реализуемого типа, такую операцию лучше не включать в методы, а реализовать как внешнюю функцию. К таким операциям, в первую очередь, относятся операции ввода-вывода, коммутативные операции сложения, вычитания, умножения и ряд других. Брюс Эккель в [11] приводит рекомендации Роба Мюррея по выбору формы перегрузки (табл. 3.2).

Таблица 3.2. Форма перегрузки операций

Операция	Рекомендуемая форма перегрузки
Все унарные операции	Метод класса
<code>= [] () -> ->*</code>	Обязательно метод класса
<code>+= -= *= /= %= &= = ^= <<= >>=</code>	Метод класса
Остальные бинарные операции	Внешняя функция

Однако, как мы знаем, внешние функции не имеют доступа к приватной части класса. Страуструп Б., очевидно, тоже столкнулся с такими проблемами, поэтому

он их решил, включив в язык механизм «друзей» (см. п. 11.4 в [1]). «Друзья» класса — это внешние функции, другие классы или методы других классов, которые имеют доступ к закрытым частям нашего класса.

Для того чтобы внешняя функция стала другом класса, необходимо в интерфейсе класса объявить ее прототип, добавив спереди слово `friend`. Например, дружественная функция умножения денежной суммы на число может быть описана в интерфейсе класса следующим образом:

```
friend TMoney operator*(const long double &a, const TMoney &b);
```

Реализация же функции не должна содержать в заголовке ни слова `friend`, ни префикса класса `TMoney`: :, например:

```
TMoney operator*(const long double &a, const TMoney &b)
{
    TMoney t = b;
    t*=a;
    return t;
}
```

И опять обратите внимание на то, что мы при реализации новой операции использовали уже проверенный метод класса — операцию с присваиванием `*=`.

СОВЕТ

Реализуйте в качестве методов класса только те операции, которые имеют своим левым аргументом объект класса. Реализуйте как дружественные функции те бинарные операции, которые могут иметь левым аргументом не объект класса.

Рассмотрим операции нашего класса `TMoney` и определим, какие из них переводить в «друзья». Операции с присваиванием, естественно, в «друзья» переводить не будем: это полноценные методы без всяких подвохов. А вот «простые» операции требуется тщательно проанализировать.

- ❑ **Сложение.** Когда мы складываем денежные суммы, левым аргументом является объект типа `TMoney`. Однако нам иногда требуется добавлять целые или дробные константы к денежной сумме. Результатом должна быть денежная сумма. Чтобы не зависеть от перестановки слагаемых, сделаем функции сложения (три варианта) друзьями.
- ❑ **Умножение.** Выполняется только умножение на константу. Но зато умножать можно и слева, и справа, поэтому эта операция — кандидат в друзья.
- ❑ **Вычитание.** Либо эта операция выполняется для двух денежных сумм, либо из денежной суммы вычитается некоторая константа. Результат — денежная сумма. Вычитание денежной суммы из константы с получением денежной суммы в качестве результата выглядит неестественно, поэтому не применяется. Таким образом, в этой операции левым аргументом всегда будет денежная сумма, что позволяет нам реализовать операцию как метод класса.
- ❑ **Деление.** Либо эта операция выполняется для двух денежных сумм, либо денежная сумма делится на число. Операция деления числа на денежную сумму, аналогично операции вычитания, не применяется, поэтому эту операцию реализуем как метод.

- *Сравнение.* Любая операция сравнения может иметь аргумент-число и справа, и слева от знака операции. Поэтому эти операции — хорошие кандидаты в друзья.
- Операции *ввода-вывода*, левыми аргументами которых являются системные объекты, тоже нужно сделать дружественными.

Реализация функций ввода-вывода должна быть такой, как показано в листинге 3.12.

Листинг 3.12. Функции ввода-вывода, реализованные как дружественные

```
ostream& operator<<(ostream& t, const TMoney &r)
{ string s = r.toString();
  return (t << s);
}
istream& operator>>(istream& t, TMoney &r)
{ long double rr = 0.0;
  t >> rr;
  r.Summa = fabs(rr);
  r.Summa = r.round(r.Summa * 100);           // округляем
  if (rr < 0) r.Summa = -r.Summa;             // учитываем знак
  return t;
}
```

Имена всех полей и методов класса `TMoney` теперь заданы с префиксом — именем объекта-параметра. Как видите, обработка при вводе довольно сложная: сначала вводится произвольное число, затем выполняется округление абсолютного значения, потом присоединяется знак денежной суммы.

Все подсказки, если они требуются, должна выводить на экран программа-клиент. Соответственно, в операции вывода на экране появляется только число без всяких дополнительных обозначений валюты. Это позволит нам в дальнейшем специализировать эти операции в классах-наследниках¹.

Функции сложения и умножения на число должны иметь следующие прототипы в интерфейсе класса:

```
friend TMoney operator+(const TMoney &a, const TMoney &b);
friend TMoney operator+(const TMoney &a, const long double &b);
friend TMoney operator+(const long double &a, const TMoney &b);
friend TMoney operator*(const TMoney &a, const long double &b);
friend TMoney operator*(const long double &a, const TMoney &b);
```

Обратите внимание: операция сложения перегружена для всех сочетаний аргументов. Это не очень приятно, поскольку аналогичное количество функций придется писать и для операций сравнения — по три штуки для каждой операции. Во-первых, это увеличивает размер и исходной, и работающей программ. Во-вторых, просто хотелось бы избежать такой однообразной рутинной работы. И это можно сделать — а помогут нам в этом конструкторы (см. п. 12.1 в [1]).

¹ См. главу 8.

Преобразование типов

Как известно, в C++ реализованы явные и неявные преобразования типов (см. п. 4 в [1]). Неявные преобразования выполняются при вычислениях выражений и присваивании. Коротко правила преобразования по умолчанию можно описать такой цепочкой (см. п. п. 5/9 в [1]):

```
[[char,short]->int]->unsigned int->long->unsigned long->float->double->
long double
```

Это не означает, что преобразование типа `int` в `double` обязательно выполняется строго последовательно, как прописано в цепочке. Просто менее «объемный» тип преобразуется к более «объемному». Типы `char` и `short` перед выполнением операции всегда сначала преобразуются в `int`, а затем выполняется операция. При этом могут выполняться другие неявные преобразования, например `int` в `double`. Кроме того, к типу `int` по умолчанию приводится перечислимый тип `enum`.

По умолчанию может выполняться и обратное преобразование, например `float` в `int`. Однако подобного рода преобразования связаны с потерей информации (дробная часть просто отбрасывается), поэтому они называются *сужающими приведениями*. Хороший компилятор обязательно предупреждает о таких операциях.

Явные преобразования язык C++ поначалу унаследовал от C. Операция преобразования в стиле C может записываться в форме

(тип) выражение

Первоначально в C++ явное преобразование записывалось тоже в короткой форме, хотя и более привычной:

тип (выражение)

С точки зрения результата эти две формы абсолютно эквивалентны. Брюс Эккель предупреждает [11], что «вы намеренно открываете брешь в системе безопасности типов C++ и запрещаете компилятору сообщать о неправильных преобразованиях типов». Ему вторит Б. Страуструп в [2], отмечая, что «явное преобразование типа ... используется неоправданно часто и является основным источником ошибок». Поэтому в стандарт C++ включены новые операторы явного преобразования типов, которые не являются взаимозаменяемыми (см. п. 5.4 в [1]):

```
static_cast<тип>(выражение)           // см. п. п. 5.2.9 в [1]
reinterpret_cast<тип>(выражение)       // см. п. п. 5.2.10 в [1]
dynamic_cast<тип>(выражение)          // см. п. п. 5.2.7 в [1]
```

Кроме того, есть еще оператор отмены константности `const_cast<>` (см. п. п. 5.2.11 в [1]). Как видим, операторы достаточно «длинные», поэтому прежде, чем писать преобразование, любой программист подумает, а надо ли это делать.

Рассмотрим оператор явного приведения типа

```
static_cast<T>(e)
```

В стандарте C++ сказано (см. п. п. 5.2.9/2 [1]), что такое приведение типа возможно только в том случае, если имеет смысл конструкция `T t(e)`. Здесь `e` — выражение,

приводимое к типу *T*, тип *T* — это либо встроенный тип, либо реализованный класс, а *t* — некоторая временная переменная. С практической точки зрения это означает, что данный оператор выполняет приведение типов с *проверкой* во время компиляции. Например, все преобразования в указанной ранее цепочке можно выполнять явно при помощи оператора `static_cast<>`.

Оператор `reinterpret_cast<>` выполняет преобразование типов без проверки при компиляции. На практике такое преобразование часто необходимо тогда, когда точно известно, что объекты разных типов занимают в памяти одинаковое количество байтов и *фактического* преобразования делать как раз не нужно. Например, с помощью этого оператора можно выполнять преобразования указателей (что довольно часто и делается). Б. Страуструп в [2] и стандарт (см. п. п. 5.2.10/3 в [1]) предупреждают, что выражение преобразования с оператором `reinterpret_cast<>` зависит от реализации и «практически никогда не переносимо». Примеры применения этих операторов мы еще увидим.

Оператор `const_cast<>` позволяет временно отменить константность. Рассмотрим несколько искусственный пример функции с константной ссылкой в качестве параметра:

```
int ff(const int &a)           // константный параметр
{  const_cast<int &>(a) = 6;    // отмена "константности"
  return a;
}
```

Система Visual C++.NET 2003 транслирует данное определение без ошибок, и следующий вызов выведет на экран число 6:

```
cout<<ff(8)<<endl;
```

Совершенно корректным будет и вызов

```
cout<<ff(k)<<endl;
```

Здесь *k* — некоторая целочисленная переменная. Однако данный вызов имеет *побочный эффект*: переменная *k* изменяет значение и становится равной 6. Интересно, что изменения не происходит, если переменная *k* объявлена как константа:

```
const int k = 3;
```

Visual C++.NET 2003 никаких сообщений (и даже предупреждений) не выдает.

Еще один оператор преобразования — `dynamic_cast<>` — выполняет преобразование во время работы программы и применяется для преобразования указателей и ссылок родственных *полиморфных классов*¹ — его мы изучим в дальнейшем.

Преобразование типов и конструкторы

Конструкторы вносят свою лепту в преобразование типов (см. п. п. 12.3.1 в [1]). Определение оператора `static_cast<>` фактически означает, что в классе *T* должен быть задан конструктор инициализации с аргументом, тип которого совпадает

¹ Полиморфными называются классы с виртуальными функциями (см. главу 9).

с типом выражения `e`: выражение преобразуется конструктором в объект типа `T`. Например, если мы еще раз взглянем на конструктор инициализации в классе `TMoney`, то увидим, что наш конструктор фактически выполняет неявное преобразование типа:

```
long double -> TMoney
```

Поэтому мы можем упростить интерфейс нашего класса `TMoney` за счет сокращения количества дружественных функций — нам нет необходимости писать три варианта операции сложения, достаточно написать один:

```
friend TMoney operator+(const TMoney &a, const TMoney &b);
```

Тем не менее правильно работают все три варианта сложения:

```
TMonet t = 2, s = 0;
s = t+t; cout << s << endl;
s = t+2; cout << s << endl;
s = 2+t; cout << s << endl;
```

Если мы опять включим отладочный вывод в наши конструкторы, то увидим, что сначала вызывается конструктор инициализации, аргументом которого является заданная константа, и только после этого срабатывает операция сложения. Точнее, в данном случае конструктор вызывается для создания временного анонимного объекта. С одной стороны, это позволяет нам не писать лишние функции, полагаясь на преобразования, с другой стороны, лишние вызовы конструктора, очевидно, снижают эффективность выполнения программы. Если же мы хотим повысить эффективность программы и избавиться от лишних вызовов конструкторов, то надо реализовать перегруженную операцию со всеми возможными сочетаниями параметров. Тогда в процессе работы просто будет вызываться подходящий вариант, и не будет выполняться никакой «закулисной» работы по преобразованию аргументов.

То же самое касается и методов: нам нет необходимости иметь два метода сложения с присваиванием. Достаточно оставить единственный — тот, параметром в котором является денежная сумма. При сложении денег с числом оно будет преобразовано в деньги конструктором. Таким образом, и операции сравнения нам достаточно реализовать в единственном экземпляре с двумя аргументами типа `TMoney`, как показано в листинге 3.13.

Листинг 3.13. Реализация функций друзей сравнения

```
bool operator==(const TMoney &a, const TMoney &b)
{ return (a.Summa == b.Summa);          // сравниваются суммы
}
bool operator!=(const TMoney &a, const TMoney &b)
{ return !(a == b);                      // сравниваются деньги
}
bool operator<(const TMoney &a, const TMoney &b)
{ return (a.Summa < b.Summa);            // сравниваются суммы
}
bool operator>=(const TMoney &a, const TMoney &b)
```

Листинг 3.13 (продолжение)

```

{ return !(a < b);                                // сравниваются деньги
}
bool operator>(const TMoney &a, const TMoney &b)
{ return !((a < b)&&(a == b));                    // сравниваются деньги
}
bool operator<=(const TMoney &a, const TMoney &b)
{ return (a < b)|| (a == b);                      // сравниваются деньги
}

```

Конструктор вызывается каждый раз, когда в операции с денежной суммой участвует константа-число.

Функции-операции преобразования

При наличии преобразования в одну сторону естественно иметь возможность выполнять и обратное преобразование:

TMoney -> long double

C++ позволяет это сделать: мы должны определить в классе функцию-метод преобразования, как показано в листинге 3.14.

Листинг 3.14. Функция преобразования

```

operator long double() const
{ return Summa/100; }

```

В данном случае мы осуществили перегрузку функции-операции (см. п. п. 12.3.2 в [1]). Тогда становятся возможными присвоения объектов типа TMoney переменным встроенного типа long double, например:

```

int main()
{   TMoney    t = 2.67,
    s = TMoney(1234.56);
    long double ww = s;                // присвоили деньги
    cout << ww << endl;
    ww = t;                            // присвоили деньги
    cout << ww << endl;
    return 0;
}

```

Аналогичные неявные преобразования выполняются при передаче параметров в функцию и при возврате результата. Пусть у нас определены функции:

```

TMoney f1(const TMoney &a)
{ return a; }
long double f2(const TMoney &a)
{ return a; }

```

Тогда допускаются вызовы:

```

TMoney t = f1(2.67);
long double ww = f2(t);

```

В первом случае осуществляется неявное преобразование целого числа 2.67 в тип TMoney конструктором, и переменная t получает значение 2 рубля 67 копеек.

А во втором случае неявное преобразование в дробное число выполняется при возврате, и переменной `w` присваивается значение 2.67.

Общий вид функции преобразования следующий:

```
operator type();
```

Здесь `type` может быть встроенным типом, типом класса или именем `typedef`. На месте `type` не может стоять тип массива или функции. Функция преобразования *обязательно* должна быть методом. В объявлении не должны задаваться ни тип возвращаемого значения, ни список параметров — это будет ошибкой.

Неявное преобразование можно реализовать для любых типов. Пусть у нас определен некоторый класс `X`. Тогда в определении класса `Y` мы можем определить неявное преобразование типов:

```
X -> Y
Y -> X
```

Это показано в листинге 3.15.

Листинг 3.15. Неявное преобразование типов произвольного вида

```
class Y {
    Y(X a){ /* ... */ }           // X -> Y
    operator X() { /* ... */ };    // Y -> X
};
```

Первое преобразование делает конструктор инициализации, а второе — функция-операция приведения типа. В результате объекты типа `Y` можно инициализировать объектами типа `X`, разрешается смешивать без указания явных преобразований объекты типов `X` и `Y` в одном выражении.

Опытные программисты знают, что определение неявных преобразований часто приводит к неприятным сюрпризам. И это хорошо, если неприятности выявляются при трансляции. Например, для нашего класса `TMoney` с определенным ранее преобразованием в `long double` вдруг выясняется, что выражения, в которых участвуют денежные суммы и дробные константы, являются неоднозначными. Например:

```
long double ww = s+1;           // ошибка трансляции
```

Этот оператор в системе Visual C++.NET вызывает ошибку трансляции:

```
error C2666: 'TMoney::operator+' : 2 overloads have similar conversions
```

Это сообщение означает, что компилятор не может выбрать, какое из преобразований применять при сложении: то ли привести константу 1 к типу `TMoney`, то ли `s` привести к типу `long double`. Заметьте, что операции сложения денег и констант не определены, тем не менее ошибка возникает.

СОВЕТ

Объявляйте функции преобразования только тогда, когда без них совершенно невозможно обойтись. Иначе рискуете нарваться на очень «хитрые» ошибки.

Запрет неявных преобразований

Таким образом, с определением преобразований по умолчанию надо быть осторожным. Лучше даже запретить их совсем, чтобы заставить клиента явно обозначать свои намерения при преобразованиях типов. Запретить следующее преобразование легко — достаточно просто не определять функцию-операцию преобразования:

```
TMoney -> long double
```

Но C++ позволяет запретить неявные преобразования и конструктором. Для этого в языке существует специальное ключевое слово `explicit` (см. п. п. 7.1.2/6 в [1]). Заголовок конструктора инициализации должен быть таким:

```
explicit TMoney(const long double &t=0.0);           // конструктор
```

Тогда допускается только явный вызов конструктора как при объявлении с инициализацией, так и при передаче параметра:

```
TMoney t = f1(TMoney(2.67));
TMoney s = TMoney(12.34);
```

При попытках не писать справа от знака присваивания тип `TMoney` возникают ошибки трансляции:

```
TMoney p = 2.50;                                // ошибка трансляции
```

Однако следующая запись не является запрещенной, так как представляет собой явный вызов конструктора:

```
TMoney p(TMoney(2.50));
```

Классы-оболочки встроенных типов

Преобразование типов позволяет нам обойти запрет на определение новых операций для встроенных типов. Для этого мы должны определить класс-оболочку для встроенного типа и реализовать в этом классе преобразование типов и требуемые операции. Давайте реализуем операцию возведения в целую степень для типа `double`. Класс-оболочка представлен в листинге 3.16.

Листинг 3.16. Класс-оболочка `Real` для `double`

```
class Real
{
    double t;
public:
    Real(const double &a=0.0){ t = a;}           // double -> Real
    operator double () { return t; }             // Real -> double
    // "новая" операция возведения в целую степень
    Real operator^(int d)
    {
        double r = t;
        if (d==0) return 1;                       // t^0 = 1
        else if (r==0) return 0;                   // 0^d = 0
        else if (d==1) return *this;               // t^1 = t
        else if (d<0) { r=1/r; d=-d; }             // t^(-d)=(1/t)^d
    }
}
```

```
double p = 1;
for(int i=1; !(i>d); i++) p* = r;    // вычисляем степень
return p;                          // неявное преобразование
}
};
```

Проверка показывает, что все варианты работают правильно:

```
int main()
{
    Real t = 2;
    cout << (t^0) << endl;          // 2^0 = 1
    cout << (t^1) << endl;          // 2^1 = 2
    t = 1;
    cout << (t^12) << endl;         // 1^12 = 1
    cout << (t^-2) << endl;         // 1^-2 = 1
    t = 0;
    cout << (t^0) << endl;          // 0^0 = 1
    cout << (t^3) << endl;          // 0^3 = 0
    t = 2;
    cout << (t^2) << endl;          // 2^2 = 4
    cout << (t^-2) << endl;         // 2^-2 = 0.25
    return 0;
}
```

Но надо помнить, что эта операция работает только для переменных типа `Real` — константы типа `double` таким способом возводить в степень не получится.

Таким образом, мы можем легко ввести «новые» арифметические операции для любого встроенного типа данных. Например, для `int` можно написать класс `Integer`, который инкапсулирует все, касающееся целых чисел. Собственно, мы действовали аналогично разработчикам языка программирования Java, реализовав тип-оболочку для встроенного типа данных.

Резюме

Язык C++ поддерживает перегрузку встроенных операций для нового типа данных. Новый тип данных определяется в C++ с помощью конструкций `enum`, `class`, `struct` и `union`. Операции можно перегружать либо как внешние глобальные функции, либо как методы класса. Если операция перегружена методом класса, то количество явно определяемых параметров у нее на один меньше: у бинарных операций параметр один, а у унарных параметр отсутствует. Операции инкремента и декремента разрешается перегружать в префиксной и постфиксной формах. Постфиксная форма отличается наличием фиктивного параметра целого типа. Операции с присваиванием должны возвращать ссылку на объект своего класса. Выражение, которое может стоять слева от знака операции присваивания, называется l-значением.

Не все операции подходят для перегрузки методами класса — такие операции лучше перегружать как дружественные функции. Перегрузка операций имеет ограничения. Ряд особо важных операций перегружать запрещено. Некоторые операции разрешается перегружать только методами класса. Унарную операцию

нельзя сделать бинарной, а бинарную — унарной, однако отдельные операции можно перегружать и как бинарные, и как унарные. При перегрузке операций запрещается задавать параметры по умолчанию.

Для явного преобразования типа в C++ дополнительно включено 4 операции преобразования, которые применяются в различных ситуациях.

Конструкторы инициализации обеспечивают преобразование типов параметров в тип класса. Обратное преобразование можно выполнить, если перегрузить в классе функцию преобразования. Функция преобразования может быть только методом класса. Сочетание конструкторов и функции преобразования обеспечивает преобразование типов по умолчанию. Это позволяет писать классы-оболочки для встроенных типов. Неявный вызов конструктора инициализации можно запретить, если объявить конструктор с ключевым словом `explicit`.

Контрольные вопросы

1. Какие операции нельзя перегружать? Как вы думаете, почему?
2. Можно ли перегружать операции для встроенных типов данных?
3. Можно ли при перегрузке изменить приоритет операции?
4. Можно ли определить новую операцию?
5. Можно ли перегрузить операцию «запятая»?
6. Чем отличается функциональная форма вызова бинарной операции от infixной формы?
7. Перечислите особенности перегрузки операций как методы класса. Чем отличается перегрузка внешним образом от перегрузки как метод класса?
8. Какой результат должны возвращать операции с присваиванием?
9. Объясните, что такое l-значение.
10. Как различаются перегруженная префиксная и постфиксная операции инкремента и декремента?
11. Что означает выражение `*this`? В каких случаях оно используется?
12. Какие операции не рекомендуется перегружать как методы класса? Почему?
13. Объясните, почему при перегрузке операций нельзя задавать параметры по умолчанию?
14. Какие операции разрешается перегружать только как методы класса?
15. Перечислите все возможные способы задания явного преобразования типов.
16. Дайте определение дружественной функции. Как объявляется дружественная функция? А как определяется?
17. Объясните, какую операцию выполняет конструкция `static_cast<>`?
18. Какой вид конструктора фактически является конструктором преобразования типов?

19. Чем различаются операции `reinterpret_cast<>` и `static_cast<>`?
20. Для чего нужны функции преобразования? Как объявить такую функцию в классе?
21. Объясните назначение операции `const_cast<>`.
22. Как запретить неявное преобразование типа, выполняемое конструктором инициализации?
23. Какие проблемы могут возникнуть при определении функций преобразования?
24. Для чего служит ключевое слово `explicit`?
25. Как с помощью функции преобразования и конструктора инициализации определить «новые» операции для встроенных типов данных.

Упражнения

1. Определить перечислимый тип `WeekDay`. Определить для этого типа операцию перехода к следующему дню как операцию инкремента `operator++` в префиксной и постфиксной формах, а также операцию перехода к предыдущему дню как операцию `operator--` в обеих формах. Реализовать операцию вывода как перегруженную операцию `operator<<`, которая должна выводить на экран английские названия дней недели.
2. Определить структуру `Pair` с двумя полями, `first` и `second`, типа `float`. Реализовать в структуре конструкторы инициализации с присвоением параметрам значений по умолчанию. Перегрузить операцию с присваиванием `operator+=` как метод. Определить дружественные функции сравнения и операцию вывода.
3. Реализовать класс `Time` из упражнения 5 в главе 2. Метод вычисления разности двух моментов времени реализовать как перегруженную операцию `operator-`, а метод, вычисляющий время через заданное количество минут, — как перегруженную операцию сложения `operator+`. Для вывода значения времени на экран реализовать дружественную функцию `operator<<`.
4. Разработать класс `Date` для работы с датами. В классе должно быть три поля: год, месяц и день. Для представления месяцев определить в классе перечислимый тип `month`. Перегрузить операцию `operator+` для вычисления даты через заданное количество дней. Перегрузить операции инкремента `operator++` и декремента `operator--` для вычисления следующей и предыдущей дат.
5. Реализовать класс `Fraction` из упражнения 7 в главе 2. Указанные там операции реализовать как дружественные функции, перегрузив подходящие операции. Реализовать дружественную операцию вывода, перегрузив операцию `operator<<`.
6. Создать класс `Goods` (товар). В классе должны быть представлены поля наименования товара, даты оформления, цены товара, количества единиц товара и номера накладной, по которой товар поступил на склад. Для представления

даты использовать класс `Date` из упражнения 4. Реализовать операции изменения цены товара, изменения количества товара (увеличения и уменьшения), вычисления стоимости товара. Метод `toString()` должен выдавать стоимость товара в виде строки. Операции ввода-вывода реализовать с использованием метода `toString()`.

7. Реализовать класс `Fraction`, используя структуру `Pair`. Структура включает в себя два поля подходящего типа: `first` и `second`. Обеспечить реализацию всех арифметических операций и операций сравнения.
8. Разработать полнофункциональный класс `Double`, имитирующий встроенный тип `double`. Реализовать все операции, которые допустимы с типом `double`, самостоятельно выбрав, какие из них будут методами, а какие — дружественными функциями. Определить операцию возведения в произвольную степень как операцию `operator^` (см. листинг 3.16). Обеспечить преобразование во встроенный тип `double`. Реализовать в качестве методов все элементарные функции.
9. Реализовать класс `Money`, но денежная сумма должна быть представлена строкой `string` длиной 100 элементов. Младшая цифра имеет младший индекс.
10. Реализовать полнофункциональный класс `LongInteger` (см. упражнение 3 в главе 2). Класс должен обеспечивать инициализацию числами любого целого типа и строкой типа `string`. Должны быть реализованы все арифметические операции, работающие в C++ с целыми числами, и все операции сравнения. Реализовать операции ввода-вывода как дружественные функции.
11. Создать класс `BitString` для работы с битовыми строками длиной 64 бита (см. упражнение 9 в главе 2). Реализовать перегруженные операции как методы класса.

Глава 4

Массивы и классы

Массив представляет собой группу элементов одного типа, каждый из которых имеет номер. Как мы видели в главе 1, реализовав некоторый класс, мы получаем возможность объявлять массивы объектов данного типа (см. листинг 1.19). Нужно только помнить, что для создания каждого элемента массива вызывается конструктор без аргументов. При наличии в классе конструктора инициализации можно инициализировать элементы массива обычным образом: для каждого элемента будет вызван конструктор инициализации. Использовать такой массив разрешается точно так же, как и массив объектов встроенных типов.

Однако языку C++ массивы достались в наследство от C; это — очень простые, даже примитивные конструкции. При использовании массивов обычно требуется писать довольно много кода, который к тому же подвержен ошибкам — любой программист вам это подтвердит. Такое положение связано в первую очередь с тем, что при обращении к элементам массива по индексу величина индекса по умолчанию никак не проверяется. Поэтому очень часто возникает ошибка выхода за границу массива. Во-вторых, при передаче массива в качестве параметра и при возврате в качестве результата функция на самом деле оперирует не массивом, а адресом его первого элемента. Поэтому надежность программ, оперирующих массивами, оставляет желать лучшего.

Кардинально изменить положение позволяют классы. Рассмотрим «взаимоотношения» массивов и классов подробнее.

Поля-массивы в классе

До сих пор в качестве полей мы использовали только скалярные переменные. Но в C++ нет запрета на объявление в классе поля-массива. Естественно, размер класса с полем-массивом увеличивается на размер массива. Однако использование массива в классе недостаточно хорошо отражено в стандарте. Это вызывает

многочисленные вопросы, особенно связанные с инициализацией¹. Мы начнем разбираться с этими вопросами на самом простом примере, в котором объявляются и инициализируются несколько полей-массивов (листинг 4.1).

Листинг 4.1. Поля-массивы в классе

```
class Arrays
{
    int m0[10];
    static const unsigned int k = 10;
    enum { n = 10 };
    int m1[k];
    int m2[n];
public:
    Arrays() // конструктор
    { for (int i = 0; i < k; ++i)
        m0[i]=m1[i]=m2[i]=0;
    }
};
```

В классе `Arrays` объявлено три поля-массива: `m0`, `m1` и `m2`. Как видим, задать количество элементов массива можно либо явной константой, либо константным выражением со статическими и (или) перечислимыми константами, причем константы должны быть определены раньше массива. Задавать количество элементов поля-массива обязательно.

Нельзя задать количество элементов как значение другого поля, заполняемого конструктором, даже если это поле константное. Например, пусть в классе объявлены следующие поля:

```
const int k;
int m[k];
```

В этом случае система Visual C++.NET 2003 выдает ошибку компиляции C2327, которая сигнализирует о том, что `k` в данном случае не является ни статической константой, ни константой перечислимого типа.

Конструктор без аргументов `Array()` обнуляет массивы, выполняя цикл в теле. Такой способ инициализации поля-массива — в теле конструктора — является наиболее простым и позволяет присваивать элементам массива любые значения. Однако наиболее часто выполняется обнуление, поэтому для массивов разрешается применять инициализацию нулем. Тогда наш конструктор выглядит значительно проще:

```
Arrays(): m0(), m1(), m2() {}
```

Для массива объектов некоторого типа такая запись означает вызов конструктора по умолчанию (без аргументов), и не забудьте, что этот вызов выполняется для каждого элемента массива.

¹ Удивительно, но мне не удалось обнаружить в стандарте явного определения способов инициализации полей-массивов. И в известной мне литературе по этому вопросу тоже практически ничего не написано.

К сожалению, применение списков инициализации для полей-массивов этим и ограничивается — в скобках нельзя ничего указывать. Например, при написании следующей конструкции система Visual C++.NET 2003 выдаст сообщение об ошибке C2536, сигнализируя о том, что явная инициализация массивов запрещена:

```
Arrays(): m0(1,2,3,4,5), m1(), m2() {}
```

Попытки написать в скобках некоторое одиночное выражение (например, вызов функции, заполняющей массив) компилятор отвергает по той же причине.

Явная инициализация не проходит даже для символьных массивов. Например, объявим в классе `Arrays` символьный массив

```
char s[4];
```

После этого попытаемся в списке инициализации конструктора присвоить этому массиву символьную константу:

```
Arrays(): m0(), m1(), m2(), s("abc") {}
```

Тогда в системе Visual C++.NET 2003 мы получим сообщение об ошибке:

```
error C2536: 'Arrays::Arrays::s' :  
cannot specify explicit initializer for arrays
```

Это сообщение говорит о том, что нельзя определять явный инициализатор для массивов.

Тем не менее можно задать в классе указатель на символ:

```
char *s;
```

Затем можно инициализировать указатель символьной константой в списке инициализации конструктора, как было показано ранее.

Как это ни странно, но большие проблемы возникают при попытках объявить в классе константный массив встроенного типа! Как мы уже выяснили, константы нельзя инициализировать в теле конструктора, значения им присваиваются только в списке инициализации конструктора. Однако для константного массива встроенного типа не работает даже инициализация нулем.

ПРИМЕЧАНИЕ

Этот вопрос практически не отражен в стандарте, поэтому компиляторы ведут себя по-разному. В системе Visual C++.NET 2003 выдается ошибка компиляции C2439, а Borland C++ Builder 6 выдает только предупреждение W8038 о том, что массив не инициализируется.

Не проходит и отмена константности. Например, зададим массив `m0` как константный, а в теле конструктора определим инициализацию в цикле:

```
for (int i = 0; i < 10; ++i)  
    const_cast<int>(m0[i]) = 0;
```

Однако и Visual C++.NET 2003, и Borland C++ Builder 6 отказываются компилировать такой цикл.

Удивительно, но для константного массива из объектов невстроенного типа задавать инициализацию нулем разрешается. Для этого в классе должен быть определен конструктор без аргументов, который вызывается для инициализации каждого элемента константного поля-массива. Например, вполне можно инициализировать константный массив денег:

```
const TMoney ss[10];
```

Для этого достаточно задать в списке инициализации конструктора инициализацию нулем `ss()`. Как реально инициализируется такой массив, конечно, зависит от реализации конструктора по умолчанию.

Реализация простого класса строк

Символьный массив — самый распространенный тип массива в C++. Объясняется это тем, что встроенного строкового типа в C++ нет. Хотя средства обработки строк в C++ достаточно развиты (см. приложение), для изучения механизма использования массивов как полей классов реализуем простой класс строк, похожих на строки Turbo Pascal [44], а заодно разберемся и с перегрузкой операции индексирования `operator[]`. Отметим, что для добавления строк в язык программирования в системе Turbo Pascal потребовалось реализовать их в трансляторе, а мы сделаем это сами средствами стандартного языка C++.

Назовем наш класс строк `TString`, чтобы не иметь проблем со стандартным классом `string`, который реализован в составе стандартной библиотеки C++ (см. п. 21 в [1]). В Turbo Pascal строка представляет собой массив байтов длиной 256 элементов. Первый байт содержит длину строки, остальные байты — информационные, они содержат символы строки. Индексы символов изменяются от 1 до 255. В системе реализованы операции сравнения, присваивания, сцепления строк и индексирования отдельного символа. Со строками в системе Turbo Pascal оперируют шесть функций:

- `Length()` возвращает длину строки;
- `Pos()` выполняет поиск одной строки в другой слева направо; результатом является номер первого символа найденной строки или нуль, если строка не найдена;
- `Delete()` удаляет из строки заданное количество символов, начиная с указанного;
- `Insert()` выполняет вставку одной строки в другую после указанного символа;
- `Copy()` возвращает подстроку исходной строки;
- `Concat()` выполняет сцепление произвольного количества строк.

Кроме того, со строками работают стандартные функции ввода-вывода системы Turbo Pascal.

Очевидно, что набор операций слишком ограничен, поэтому в нашем классе `TString` существенно расширим множество методов и операций, по возможности максимально приблизив интерфейс класса к интерфейсу стандартного класса `string`.

Обязательный набор операций включает в себя все операции сравнения, а также операции сцепления + и индексирования []. Писать операцию присваивания, очевидно, нет необходимости, так как она создается по умолчанию. Помимо перечисленных, мы можем реализовать еще несколько полезных операций. Первым кандидатом является операция сцепления с присваиванием +=. Операции удаления подстроки тоже можно разнообразить, реализовав, например, операции -= и -. Перегруженные операции дают возможность оперировать только всей строкой. Для манипулирования подстроками реализуем множество дополнительных методов:

- присваивания с общим именем assign;
- сравнения с общим именем compare;
- поиска подстроки слева и справа с общими именами find и rfind соответственно;
- добавления, удаления, вставки и замены подстрок с общими именами append, erase, insert и replace;
- преобразования строки в верхний и нижний регистры.

Полезной также будет функция substr(), возвращающая подстроку текущей строки. Так как преобразование строки в верхний и нижний регистры представляет собой фактически унарную операцию, преобразование в верхний регистр можно реализовать, перегрузив операцию инкремента ++, а преобразование в нижний регистр — перегрузив операцию декремента --.

Нам потребуются конструктор без аргументов и несколько конструкторов инициализации: символьной и строковой константой, подстрокой уже объявленной строки и символьного массива. Конструктор копирования, как обычно, можно не определять. Интерфейс класса TString представлен в листинге 4.2.

Листинг 4.2. Интерфейс класса TString

```
class TString
{
public:
    typedef unsigned char byte;           // для сокращения записи
    // конструкторы
    TString():size(),s() { };              // конструктор без аргументов
    TString(const char ch, byte count = 1);
    TString(const char S[]);
    TString(const char *First, const char *Last);
    TString(const char S[], byte ind, byte count=-1);
    TString(const TString &S, byte ind, byte count=-1);
    // методы присваивания
    TString& assign(const char ch, byte count = 1);
    TString& assign(const char rhs[], byte ind, byte count=-1);
    TString& assign(const char *First, const char *Last);
    TString& assign(const TString &rhs, byte ind, byte count=-1);
    // индексирование
    char& operator[](const byte &ind);
    const char& operator[](const byte &ind) const;
    byte Length() const { return size; }
```

Листинг 4.2 (продолжение)

```

// сравнение
int compare(byte pos, byte count, const TString &rhs) const;
int compare(byte pos, byte count,
            const TString &rhs, byte pos_rhs, byte count_rhs) const;
int compare(byte pos, byte count, const char rhs[]) const;
int compare(const char *First, const char *Last) const;
int compare(byte pos, byte count,
            const char rhs[], byte pos_rhs, byte count_rhs) const;
// добавление в конец
TString& operator+=(const TString &rhs);
TString& append(const char ch, byte count = 1);
TString& append(const char rhs[], byte ind, byte count=-1);
TString& append(const char *First, const char *Last);
TString& append(const TString &S, byte ind, byte count=-1);
// вставка в строку
TString& insert(byte pos, const char ch, byte count = 1);
TString& insert(byte pos, const char rhs[], byte count = -1);
TString& insert(byte pos, const char *First, const char *Last);
TString& insert(byte pos, const TString &rhs);
TString& insert(byte pos,
                const TString &rhs, byte ind, byte count = -1);
// замена подстроки
TString& replace(byte pos, byte n, const char ch, byte count = 1);
TString& replace(byte pos, byte n, const char rhs[], byte count = -1);
TString& replace(byte pos, byte n, const char *First, const char *Last);
TString& replace(byte pos, byte n, const TString &rhs);
TString& replace(byte pos, byte n,
                const TString &rhs, byte ind, byte count = -1);
// удаление подстроки
TString& erase(byte pos, byte count = -1);
TString& operator-=(const TString &rhs);
// получение подстроки
TString substr(byte pos, byte count = -1) const;
// поиск подстрок
// поиск слева направо
byte find(const TString &rhs, byte pos = 0) const;
byte find(byte index, const TString &rhs, byte pos = 0) const;
byte find(const char rhs[], byte pos = 0) const;
byte find(byte index, const char rhs[], byte pos = 0) const;
byte find(const char rhs) const;
byte find(byte index, const char rhs) const;
// поиск справа налево
byte rfind(const TString &rhs, byte pos = 0) const;
byte rfind(byte index, const TString &rhs, byte pos = 0) const;
byte rfind(const char rhs[], byte pos = 0) const;
byte rfind(byte index, const char rhs[], byte pos = 0) const;
byte rfind(const char rhs) const;
byte rfind(byte index, const char rhs) const;
// смена регистра
TString operator++(); // toUpper
TString operator--(); // toLower
// дружественные функции
friend TString operator+(const TString &lhs, const TString &rhs);
friend TString operator-(const TString &lhs, const TString &rhs);

```

```

friend ostream& operator<<(ostream& t, const TString &s);
friend istream& operator>>(istream& t, TString &s);
friend bool operator==(const TString &lhs, const TString &rhs);
friend bool operator!=(const TString &lhs, const TString &rhs);
friend bool operator<(const TString &lhs, const TString &rhs);
friend bool operator>(const TString &lhs, const TString &rhs);
friend bool operator<=(const TString &lhs, const TString &rhs);
friend bool operator>=(const TString &lhs, const TString &rhs);
private:
    byte size;
    char s[256];
};

```

Поля данных в нашем классе, естественно, закрыты. Поле `size` содержит длину строки. Строки Turbo Pascal не завершаются нулем и имеют максимальную длину 255 символов, но мы для надежности зарезервировали массив на 256 байт, чтобы в последнем байте всегда был нуль — это гарантирует нам отсутствие неожиданностей при обработке массива функциями библиотеки `<cstring>`¹. Индексирование символов, как принято в C++, будем начинать с нуля. Индекс последнего символа равен 254, а максимальная длина строки — 255 символам.

Конструкторы

В первую очередь уделим внимание реализации конструкторов. У нас шесть объявленных конструкторов, седьмой — конструктор копирования — система создаст по умолчанию. Первый — конструктор без аргументов — реализован непосредственно в классе. Этот конструктор просто обнуляет поля с помощью списка инициализации. В остальных конструкторах для обнуления и заполнения массива используется стандартная функция `memset()`, прототип которой задан в библиотеке `<cstring>`.

Второй конструктор, показанный в листинге 4.3, позволяет объявить строку, присвоив ей последовательность одинаковых символов, например:

```
TString d ('-', 60);
```

Листинг 4.3. Конструктор инициализации последовательностью символов

```

TString::TString(const char ch, byte count)
{
    TString t;                // t - обнуляется
    t.size = count;           // количество символов
    memset(t.s, ch, count);    // заполнение массива
    *this = t;                // копирование в текущий объект
}

```

В конструкторе объявлен локальный объект, который и заполняется параметрами. Параметр-количество задан по умолчанию в прототипе, показанном в интерфейсе класса, поэтому этот же конструктор позволяет инициализировать строку единственным символом, например:

```

TString t1 = '-';
TString t2 ('+');

```

¹ Описание основных функций библиотеки `<cstring>` приведено в приложении.

При задании аргументов обязательно выполнение условия ($\text{First} < \text{Last}$), то есть элемент `*Last` не включается в создаваемую строку. Именно такой способ задания параметров встречается в стандартной библиотеке шаблонов (Standard Template Library, STL), только в библиотеке обычно используются не указатели, а *итераторы*¹.

Такой конструктор позволяет нам объявлять наши строки следующим образом:

```
char *v = "Строка-инициализатор";
TString vss(v, v+6);           // vss = "Строка"
TString wss(v+7, v+strlen(v)); // wss = "инициализация"
TString tss(v+4, v+9);         // tss = "ка-ин"
```

В качестве параметров в этом случае можно задавать любые разрешенные формы указателей. Например, можно не объявлять константу-строку, а задать ее непосредственно в качестве параметра при объявлении:

```
TString rr("0123456789"+3,"0123456789"+6);
```

Переменная `rr` получит значение «345».

Пятый конструктор позволяет нам тоже инициализировать нашу строку подстрокой символьного массива, при этом подстрока задается индексом первого символа и количеством символов. Если последний параметр используется по умолчанию, то с помощью этого конструктора мы можем задать подстроку от заданного символа до конца строки инициализации. Текст конструктора представлен в листинге 4.6.

Листинг 4.6. Конструктор инициализации подстрокой символьного массива

```
TString::TString(const char S[], byte index, byte count)
{ TString t(S+index, S+index+count);           // все проверки
  *this = t;
}
```

В этом конструкторе параметры проверяются при создании локального объекта. Если параметры заданы с ошибками, то, как всегда, возвращается пустая строка.

Шестой конструктор позволяет нам инициализировать новую строку подстрокой из ранее объявленной строки (листинг 4.7).

Листинг 4.7. Конструктор инициализации из ранее объявленной строки

```
TString::TString(const TString &S, byte index, byte count)
{ TString t;
  if ( (count > S.size)||
        (index+count > S.size))
    t.size = S.size-index;           // неправильное количество
  else t.size = count;
  if (t.size>0)
    memcpy(t.s, (S.s+index), t.size); // копируем строку
  *this = t;                         // заменяем текущий объект
}
```

¹ Итераторы описываются в главе 6.

Необходимо отметить, что в конструкторах довольно серьезно проверяются параметры. Однако сообщить программе-клиенту об ошибке параметров конструктор не в состоянии — отсутствует возвращаемое значение. А если мы определим лишний параметр, то у нас не будет возможности объявить объект без инициализации, и мы тем самым грубо нарушим один из принципов синтаксиса: объявление объектов новых типов не должно отличаться от объявлений встроенных типов. Решить эту проблему можно только с помощью *механизма обработки исключений* (см. главу 7).

Реализация методов и операций

Реализация методов `assign` чрезвычайно проста: всю работу фактически делает соответствующий конструктор, что и показано в листинге 4.8 на примере двух методов. Проверку параметров в явном виде нам делать не требуется, вся проверка уже «упрятана» в конструкторе.

Листинг 4.8. Реализация методов `assign`

```
TString& TString::assign(const char ch, byte count)
{ TString t(ch, count);          // локальный объект - проверка параметров
  *this = t;                     // изменение текущего объекта
  return *this;
}
typedef TString::byte byte;      // для краткости
TString& TString::assign(const char rhs[], byte ind, byte count)
{ TString t(rhs, ind, count); *this = t;  return *this;
}
```

Остальные методы реализуются совершенно аналогично. Единственный вопрос заключается только в том, зачем нам понадобились эти методы. Разве нельзя обойтись только операцией присваивания? Дело в том, что операция присваивания — бинарная, поэтому параметр может быть только один — выражение, стоящее справа от знака операции. Для нашего класса строк — это присваиваемая строка того же типа `TString`. Прототип операции присваивания для нашего класса `TString` такой (см. п. п. 12.8/10 в [1]):

```
TString& operator=(const TString &rhs);
```

Методы позволяют нам сделать присваивание значительно более гибкой операцией — мы можем оперировать любой подстрокой.

Операция индексирования

Операция индексирования обязана возвращать ссылку, так как выражение `имя[индекс]` может стоять как справа, так и слева от знака присваивания:

```
TString t = "Привет от RR строки!";
t[12] = t[6];
```

Если выражение `имя[индекс]` стоит слева от знака присваивания, то изменяется текущий объект (так же, как и в операциях с присваиванием).

Кроме того, обычно реализуют два метода: константный и неконстантный. Неконстантный метод работает, когда выражение `имя[индекс]` стоит слева от знака присваивания. Константный метод вызывается, когда выражение `имя[индекс]` используется для доступа к символам константы-строки (объявленной явно или передаваемой как параметр по константной ссылке).

Наша операция, естественно, должна проверять правильность задания индекса. Так как тип индекса — `byte`, являющийся переопределенным типом `unsigned char`, то фактически надо проверять, не равен ли индекс 255, так как ни меньше нуля, ни больше 255 значение быть не может. Правда неясно, что делать, если заданный индекс все-таки равен 255. Наилучшим решением было бы прекратить выполнение операции и как-то сообщить об этом в вызывающую программу. Однако как и для конструкторов, это можно сделать только с помощью *механизма обработки исключений* — другим способом нам это сделать не удастся. Например, не получится возвращать код завершения, сигнализирующий об ошибке, так как операция должна возвращать ссылку, а не значение. Лишний параметр, в котором можно было бы передать код ошибки, мы тоже использовать не можем, поскольку никаких лишних параметров не допускается — операция индексирования является бинарной и должна иметь всего два аргумента. Один из них — текущий объект, второй — индексное выражение в квадратных скобках.

Поэтому до изучения механизма исключений придется реализовывать константный и неконстантный методы по-разному. Так как константный метод не изменяет содержимое полей класса, то в константном методе можно вернуть ссылку на последний нулевой элемент массива символов — это и будет служить признаком ошибки. В неконстантном методе так поступать нельзя: если операция применяется слева от знака присваивания, то, возвращая ссылку на завершающий элемент, мы разрешаем его изменять. Таким образом, мы можем «лишиться» завершающего нуля, и работа многих методов будет нарушена. Поэтому пока будем завершать этот метод аварийно.

ВНИМАНИЕ

Это — не лучшее решение. В профессиональном программировании аварийное завершение программы выполняется только в безвыходной ситуации. Мы пересмотрим это решение позднее после изучения механизма исключений.

С учетом этих соображений реализация выглядит так, как показано в листинге 4.9.

Листинг 4.9. Реализация операции индексирования

```
char& operator[](const byte &index)
{ if (index<255) return s[index];           // нормальная работа
  else abort();                             // аварийное завершение
}

const char& operator[](const byte &index) const
{ if (index<255) return s[index];           // правильный индекс
  else return s[255];                       // неправильный индекс
}
```


Функции-операции достаточно просты, поэтому мы реализовали их как подставляемые непосредственно в классе.

Другие операции

И наконец, займемся реализацией операций со строками. В первую очередь, рассмотрим операции сцепления, и начнем с операции сцепления с присваиванием (листинг 4.10).

Листинг 4.10. Реализация операции сцепления с присваиванием

```
TString& TString::operator+=(const TString &rhs)
{ memcpy(s+size, rhs.s, 255-size);    // дописываем строку
  size = strlen(s);                  // новая длина
  return *this;
}
```

С помощью функции `memcpy()` (см. приложение) мы просто «дописываем» в «хвост» текущей строки начало строки-аргумента, причем ровно столько символов, сколько имеется «свободных» нулевых байтов в текущей строке. В число переписываемых попадают и нулевые байты прицепляемой строки. Затем устанавливается новая длина текущей строки и выполняется возврат ссылки на текущий объект, чтобы операцию, как и стандартную, можно было применять многократно.

Нам достаточно иметь реализацию только с аргументом типа `TString`, так как преобразование параметра обеспечивают конструкторы инициализации.

С помощью этой операции и конструкторов мы легко реализуем другие операции сцепления, что можно наблюдать в листинге 4.11. Все методы `append()`, как и методы `assign()`, реализованы однотипно: с помощью подходящего конструктора создается локальный объект-строка, которая прицепляется к текущей строке.

Листинг 4.11. Реализация методов сцепления

```
TString& TString::append(const char ch, byte count)
{   TString t(ch, count);           // локальная строка - проверка параметров
    *this+=t;                        // прицепили к текущей строке
    return *this;
}

TString& TString::append(const TString &S, byte ind, byte count)
{   TString t(S, ind, count); *this+=t; return *this; }
// дружественная функция
TString operator+(const TString &lhs, const TString &rhs)
{   TString t = lhs; t+=rhs; return t; }
```

Операция сцепления двух строк объявлена и реализована как дружественная, чтобы не иметь проблем с типом левого аргумента. И опять обратите внимание на то, как мы здорово упростили себе работу, используя конструкторы и уже реализованную операцию сцепления с присваиванием. Такие функции даже проверять не надо¹ — они сразу работают.

¹ Забудьте! Будем считать, что я этого не говорил. Проверять написанный код надо всегда.

Покажем теперь примеры реализации функций вставки, замены и удаления. Начнем с функции удаления. Метод должен обеспечивать разные варианты удаления символов: в начале, в середине и в конце строки. Последний вариант удаления удобно задавать с одним параметром-индексом, показывающим, начиная с какого символа нужно выполнять удаление. Поэтому количество можно задать по умолчанию равным `-1`, как в других методах. Текст метода приведен в листинге 4.12.

Листинг 4.12. Метод удаления подстроки

```
TString& TString::erase(byte pos, byte count)
{
    TString t1(*this, 0, pos);           // начало строки
    if (count != static_cast<byte>(-1))   // преобразование необходимо
    {
        TString t2(*this, pos+count);     // хвост строки
        t1+=t2;                           // получили результат
    }
    *this = t1;                           // присвоили текущему
    return *this;
}
```

Метод устроен очень просто — опять основную работу выполняют конструкторы, причем локальные объекты конструируются из текущей строки. Сначала формируется строка, содержащая все символы от начала до символа с номером `pos`, причем если номер равен нулю, то формируется пустая строка — это обеспечивает конструктор. Если требуется удалить из середины, значит, надо сформировать и строку-«хвост». Результат получается как сцепление начала и «хвоста». Единственная «хитрость» — преобразование типа в условии оператора `if`. На первый взгляд кажется, что достаточно написать условие без явного преобразования:

```
if (count != -1)
```

Ведь в прототипе метода значение параметра как раз равно `-1`:

```
TString& erase(byte pos, byte count = -1)
```

Но в прототипе выполняется преобразование из `int` в `byte`, так как таков тип параметра. В условии же оператора `if` по умолчанию выполняется преобразование из `byte` в `int`, так как по стандарту менее «объемный» тип приводится к более «объемному». В данном случае `-1` — это целая константа, поэтому значение `count` (равное 255) приводится к целому, и условие не работает! То есть требуется явное преобразование.

Реализация других операций удаления тоже особых трудностей не представляет, так как в них используются уже реализованные конструкторы и методы. Листинг 4.13 иллюстрирует перегрузку операций вычитания для удаления подстроки.

Первая операция «вырезает» аргумент из текущего объекта, вторая удаляет из левого аргумента правый. В операции с присваиванием используется метод поиска `find()`. Если заданная подстрока найдена в текущей строке, то возвращается индекс первого символа. Если же подстрока отсутствует, то возвращается 255. Реализация метода `find()` будет показана далее.

Листинг 4.13. «Вычитание» подстроки из строки

```

TString& TString::operator-=(const TString &rhs)
{ byte i = this->find(rhs);           // ищем подстроку в текущей
  if (i!=static_cast<byte>(-1))       // если нашли
    this->erase(i, rhs.size);         // удаляем
  return *this;
}
// дружественная функция
TString operator-(const TString &lhs, const TString &rhs)
{   TString t = lhs;                 // формируем левый аргумент
    t-=rhs;                          // удаляем правый аргумент
    return t;                        // возвращаем результат
}

```

Пусть переменная-строка `b` имеет значение `"0123456789"`. Описываемые операции позволяют нам писать такие выражения:

```

b-="67";
TString t = b - "01";

```

Реализация операций вставки и замены основана на тех же принципах: сначала конструируются составные части результирующей строки, которые затем сцепляются в окончательный результат. Методы `replace()` обеспечивают замену не равных по длине подстрок — заменяющая строка может быть как длиннее, так и короче заменяемой подстроки. В качестве примера в листинге 4.14 приведено по одному методу из обеих групп.

Листинг 4.14. Пример реализации методов вставки и замены

```

TString& TString::insert(byte pos, const TString &rhs, byte ind, byte count)
{ TString t(rhs, ind, count);         // вставляемая строка
  TString t1(*this, 0, pos);          // левая половина текущей строки
  TString t2(*this, pos+1);           // правая половина
  t1+=t; *this = t1+t2;               // получаем результат
  return *this;
}
TString& TString::replace(byte pos, byte n,
                          const TString &rhs, byte ind, byte count)
{ (*this).erase(pos,n);               // удаляем заменяемую строку
  this->insert(pos, rhs, ind, count);   // вставляем заменяющую
  return *this;
}

```

Реализация метода `substr()` тривиальна (листинг 4.15).

Листинг 4.15. Реализация метода `substr()`

```

TString TString::substr(byte pos, byte count) const
{ TString t(s, pos, count);
  return t;
}

```

Реализация операций и методов сравнения достаточно проста, поэтому останавливаться на ней не будем.

Нам осталось совсем немного: операции поиска, операции перевода строки в верхний и нижний регистр и операции ввода-вывода. Поиск слева особых сложностей не представляет, мы используем стандартную функцию `strstr()` из библиотеки `<csrtng>` (см. приложение). В листинге 4.16 показана реализация только двух методов, остальные реализуются совершенно аналогично.

Листинг 4.16. Поиск слева

```
byte TString::find(const TString &rhs, byte pos) const
{
    if (pos > rhs.Length()) pos = 0;           // защита
    const char *sp = strstr(s, rhs.s+pos);     // поиск
    if (sp)                                     // если найдена, sp != null
        return(sp-s);                         // вычисление номера байта
    else return -1;                            // не найдена = 255
}

byte TString::find(byte index, const TString &rhs, byte pos) const
{
    if (index > size) index = 0;
    if (pos > rhs.Length()) pos = 0;
    const char *sp = strstr(s+index, rhs.s+pos);
    if (sp) return(sp-s); else return -1;
}
```

Первый вариант метода выполняет поиск в текущей строке заданной подстроки, причем можно задать не всю подстроку, а только ее «хвост», прописав ненулевое значение параметра `pos`. Функция проверяет параметр и выполняет поиск. Если строка найдена, то возвращается номер первого символа. При отрицательном результате поиска возвращается значение `-1` (которое по умолчанию переводится в 255). Второй метод, помимо этого, позволяет выполнять поиск в «хвосте» текущей строки.

Удивительно, но поиск справа в стандартной библиотеке C представлен только одной функцией `strrchr()` — поиск первого символа справа (см. приложение). С ее помощью мы легко реализуем только поиск символа справа. В других методах эту функцию использовать трудно, поэтому мы реализуем один метод `rfind()` практически «вручную», а затем задействуем его при реализации остальных методов. В листинге 4.17 это и продемонстрировано.

Листинг 4.17. Методы поиска справа налево

```
TString::byte TString::rfind(byte index,           // откуда искать
                             const TString &rhs,  // что искать
                             byte pos) const      // позиция поиска
{
    if (index > size) index = 0;                   // защита от ошибки
    if (pos > rhs.Length()) pos = 0;               // защита от ошибки
    TString lhs(*this, index);                     // где искать
    TString What(rhs, pos, rhs.Length()-pos);      // что искать
    for(int i = lhs.Length() - What.Length(); i>=0; i--) // справа налево
        if (strncmp(lhs.s+i, What.s, What.Length())==0)
            return i+index;                         // нашли
    return -1;                                      // не нашли
}
```

Листинг 4.17 (продолжение)

```

byte TString::rfind(const TString &rhs, byte pos) const
{
    return this->rfind(0, rhs, pos); // вызов предыдущего метода
}
byte TString::rfind(byte index, const char *rhs, byte pos) const
{
    TString What(rhs, pos, strlen(rhs)-pos); // что искать
    return this->rfind(index, What);         // ищем
}

```

Остальные функции реализуются аналогично. Некоторых пояснений требует только реализованный «вручную» метод. После проверки параметров с помощью конструкторов формируется строка `lhs`, в которой происходит поиск, и строка `What`, которую надо найти. Далее от «хвоста» к началу выполняется поиск с помощью функции `strncmp()` из библиотеки `<cstring>`.

Операции ввода-вывода, как обычно, перегружаются в качестве дружественных функций, текст которых приведен в листинге 4.18.

Листинг 4.18. Перегрузка операций ввода-вывода

```

std::ostream& operator<<(std::ostream& t, const TString &S)
{ return (t << S.s); }
std::istream& operator>>(std::istream& t, TString &S)
{ char ss[256] = {0}; // локальный массив - обнулили
  t.getline(ss, 255); // ввод с пробелами
  S.size = strlen(ss); // установка полей класса
  memcpy(S.s, ss, S.size); // копирование массива в поле
  return t;
}

```

Операция вывода интереса не представляет в силу своей тривиальности, а вот операцию ввода нужно рассмотреть чуть подробнее. Дело в том, что стандартная операция ввода `operator>>` для строк работает с ограничениями — до первого пробела. Нам же желательно осуществлять ввод всех символов без пропусков до нажатия клавиши `Enter`. Поэтому при перегрузке операции ввода строки мы должны использовать либо функцию `getline()` класса `string`, либо одноименный метод потока, что и сделано в данном случае¹.

Статические поля-массивы

Наши функции перевода регистра должны обеспечивать перевод и русских, и английских букв. Простейшая реализация методов преобразования в верхний и нижний регистры может быть такой, как в листинге 4.19. Поскольку перевод регистра — разовое действие, то пусть обе операции возвращают значение.

Листинг 4.19. Простейшая реализация методов изменения регистра строки

```

TString TString::operator++() // toUpper
{ const char lower[59]
    = "abcdefghijklmnopqrstuvwxyzабвгдезийклмнопрстуфхцчшщъыьэя";

```

¹ Подробности см. в главе¹⁴.

```

const char upper[59]
    = "ABCDEFGHIJKLMNOPQRSTUVWXYZБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЬЬЪЭЮЯ";
for(int i = 0; i<this->Length(); i++)
{ for (int j = 0; j < 59; j++) if (s[i] == lower[j]) s[i]=upper[j]; }
return *this;
}
TString TString::operator--()    // toLower
{ const char lower[59]
    = "abcdefghijklmnopqrstuvwxyzбвгдежзийклмнопрстуфхцчшщъьэюя";
  const char upper[59]
    = "ABCDEFGHIJKLMNOPQRSTUVWXYZБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЬЬЪЭЮЯ";
  for(int i = 0; i<this->Length(); i++)
  { for (int j = 0; j < 59; j++) if (s[i] == upper[j]) s[i]=lower[j]; }
  return *this;
}

```

Методы прекрасно работают, однако в таком виде они нас, конечно, не устраивают — любой программист тут же скажет, что использование локальных массивов очень неэффективно. Массивы заново создаются при каждом входе в функцию — на это тратится и время и место при выполнении программы.

Первое побуждение — вынести массивы из методов и объявить два поля-массива. Но тут нас ожидают некоторые сложности. Во-первых, поля нельзя объявить константными, хотя по сути своей они таковыми и являются. Дело в том, что тогда мы не сможем эти поля инициализировать — константные массивы встроенного типа никаким образом не инициализируются (см. ранее). Но если массивы не константные, то их придется заполнять в каждом конструкторе, написав для этого приватную функцию.

Во-вторых, и это гораздо важнее, неконстантные поля-массивы существенно увеличивают размер класса — на 118 байт. Пока объектов немного, это не доставляет беспокойства. Но представьте, что нам надо объявить массив размером в 1000 элементов типа `TString`. Ситуация отнюдь не надуманная, так как подобный массив, например, в качестве буфера может использовать простой текстовый редактор. Но такой массив имеет уже 118 000 лишних байтов, которые еще и заполняются во время создания — ведь *конструктор вызывается для каждого элемента массива*. Ситуация совершенно неприемлемая как с точки зрения расхода памяти, так и с точки зрения эффективного выполнения программы.

Так как массивы у нас символьные, то можно было бы обойтись указателями на символ — тем более, что их можно инициализировать символьной константой. Ситуация с памятью несколько улучшается: два указателя занимают в классе всего 8 байт. Однако указатели тоже будут инициализироваться для каждого создаваемого объекта.

Массивы нужно все-таки иметь в единственном экземпляре; они, очевидно, должны быть объявлены константными и проинициализированы один раз при объявлении. Мы можем это сделать, если объявим их глобальными. Однако такое решение совершенно неприемлемо с точки зрения инкапсуляции: глобальные переменные излишне доступны и их использование способствует появлению трудноуловимых ошибок. Массивы по сути своей являются (и должны быть) частью класса `TString`.

В языке C++ есть средство, позволяющее сделать именно то, что нам требуется: объявить массивы в единственном экземпляре, сделать их константными, проинициализировать при объявлении и тем не менее локализовать их в классе. Такие константные поля-массивы надо объявить в классе статическими (см. п. п. 9.4.2 в [1]):

```
static const char lower[59];           // маленькие буквы
static const char upper[59];          // БОЛЬШИЕ БУКВЫ
```

Инициализация (определение) статических полей выполняется вне класса:

```
const char TString::lower[59] =
"abcdefghijklmnopqrstuvwxyzабвгдежзийклмнопрстуфхцчщъыьэюя";
const char TString::upper[59] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZABVGDEJZIKLMNOPRSTUFHCXCSHBYEYU";
```

Этот оператор присваивания является *определением* статического члена класса (а в классе — только объявление). Такой оператор присваивания — единственная форма инициализации константных статических массивов (и любых других константных статических полей встроенного нецелочисленного типа). Обратите внимание, что слово `static` в операторе инициализации отсутствует — его там писать нельзя. Статические массивы, как и статические константы (и вообще любые статические поля), являются частью класса, создаются в единственном экземпляре при запуске программы (до создания любых объектов класса), поэтому их нельзя инициализировать в конструкторе.

ПРИМЕЧАНИЕ

Помимо статических полей, в классе можно объявлять и статические методы. Такие методы являются методами класса и могут вызываться до создания первого объекта этого класса.

Как мы помним, статические константы места в классе не занимают. Точно так же не увеличивают размер класса и любые статические поля — они хранятся вне класса. Но принцип инкапсуляции выполняется — доступ к таким полям имеют только методы и «друзья» класса. Наши методы перевода регистра упрощаются (листинг 4.20).

Листинг 4.20. Реализация методов перевода регистра

```
TString TString::operator++()           // toUpper
{ for(int i = 0; i < this->Length(); i++)
  { for (int j = 0; j < 59; j++) if (s[i] == lower[j]) s[i]=upper[j]; }
  return *this;
}
TString TString::operator--()           // toLower
{ for(int i = 0; i < this->Length(); i++)
  { for (int j = 0; j < 59; j++) if (s[i] == upper[j]) s[i]=lower[j]; }
  return *this;
}
```

Проверим работу методов.

```
TString t ("Мама мыла Милу");    std::cout << t << std::endl;
++t;    std::cout << t << std::endl;
--t;    std::cout << t << std::endl;
TString b = "abCDefGH";    std::cout << b << std::endl;
++b;    std::cout << b << std::endl;
--b;    std::cout << b << std::endl;
```

При выполнении этого фрагмента на экране появятся следующие строки:

```
Мама мыла Милу
МАМА МЫЛА МИЛУ
мама мыла милу
abCDefGH
ABCDEF GH
abcdefgh
```

Как видим, все работает правильно. Таким образом, мы создали законченный, вполне работоспособный класс строк.

Статические элементы класса

Помимо констант и массивов, статическим элементом класса может быть как поле, так и метод (см. п. 9.4 в [1]). Естественно, статические методы предназначены, в первую очередь, для манипуляции статическими полями. Обозначаются поля и методы словом `static`.

Страуструп Б. на с. 274 в [2] определяет статическое поле как поле, которое является частью класса, но не является частью объекта этого класса. Это означает, что инкапсуляция работает как и для обычных полей, а вот память выделяется не в объекте. Статические поля класса размещаются в статической памяти¹ (как глобальные и статические переменные) и создаются в единственном экземпляре, независимо от количества определяемых в программе объектов. Все объекты (даже созданные динамически) разделяют единственную копию статических полей (см. п. п. 9.4.2 в [1]). Более того, все классы-наследники тоже разделяют эту единственную копию (см. главу 8). При этом в класс включается только объявление, а определение статического поля выполняется отдельно за пределами класса (как мы видели ранее на примере статического константного массива).

ПРИМЕЧАНИЕ

Единственным исключением из этого правила являются целочисленные статические константы, которые разрешается инициализировать (определять) непосредственно в классе (см. листинг 2.5).

Естественно, определение должно быть единственным в программе, иначе компоновщик (не компилятор) сообщит о повторном определении. Если определение отсутствует, то мы тоже получим ошибку компоновки.

¹ См. следующую главу.

При определении поля выполняется и его инициализация. Если статическое поле является константным, то инициализация обязательна. Инициализация статических полей выполняется точно так же, как инициализация глобальных и статических переменных (см. п. п. 9.4.2/7 в [1]). Для неконстантных статических полей элементарных типов при отсутствии явной инициализации выполняется обнуление. Для статических полей неэлементарных типов, естественно, вызываются конструкторы: либо явным образом конструктор инициализации, либо конструктор по умолчанию (при отсутствии явной инициализации).

Продemonстрируем все это на элементарном примере (листинг 4.21).

Листинг 4.21. Статические поля

```
#include <iostream>
#include <string>
using namespace std;
class Person // демонстрационный класс
{ string fio; // неэлементарный тип
  int year;
public:
  Person() // конструктор по умолчанию
  : fio("Lippman"), year(1953)
  {}
  Person(string fio, int y) // конструктор инициализации
  { fio = fio; year = y; }
  void print() const; // вывод полей на экран
};
class StaticField
{ static const int m01 = 1; // инициализированная константа
  static const int m02; // неинициализированное константное поле
  static const double m03; // константное поле нецелого типа
  static int m04; // неконстантное поле целого типа
  static double m05; // неконстантное поле нецелого типа
  static const Person p; // константное поле неэлементарного типа
  static Person t; // неконстантное поле неэлементарного типа
public:
  static void print(); // статический метод
};
// определение статических полей
const int StaticField::m02 = 2; // обязательная инициализация
const double StaticField::m03 = 3.1; // обязательная инициализация
int StaticField::m04; // инициализация по умолчанию (ноль)
double StaticField::m05; // инициализация по умолчанию (ноль)
const Person StaticField::p("Kupaev", 1999); // конструктор инициализации
Person StaticField::t; // конструктор по умолчанию
// определение статического метода
void StaticField::print()
{ cout << m01 << endl;
  cout << m02 << endl;
  cout << m03 << endl;
  cout << m04 << endl;
  cout << m05 << endl;
```

```

    p.print();
    t.print();
}
void Person::print() const           // константный нестатический метод
{ cout << Fio << ', ' << year << endl; }
int main()
{   StaticField::print();           // вывод значений статических полей
    return 0;
}

```

В этой программе класс `Person` написан только для демонстрации статических полей неэлементарного типа. В нем определено два конструктора — по умолчанию и инициализации, — чтобы продемонстрировать вызовы при инициализации статических полей типа `Person`.

В классе `StaticField` объявляется ряд статических полей, константных и неконстантных. Для вывода значений на экран объявляется статический метод `print()`. После определения класса задано определение статических полей. Для константных статических полей (`m02` и `m03`) инициализацию писать обязательно. Неконстантные статические поля элементарных типов (`m04` и `m05`) инициализируются по умолчанию — выполняется обнуление. И наконец, определение статических полей типа `Person` демонстрирует вызов конструкторов. Поле `p` должно быть обязательно проинициализировано, так как является константным. Именно для его инициализации в классе `Person` реализован конструктор инициализации. Поле `t` не является константным, поэтому определяется без явной инициализации. Однако для него вызывается конструктор по умолчанию, что можно наблюдать при вызове статического метода `print()` в главной программе. Программа выведет на экран следующее:

```

1           // m01 - константа, инициализируется в классе
2           // m02 - константа, инициализируется явно
3.1         // m03 - константа, инициализируется явно
0           // m04 - не константа, инициализируется по умолчанию (0)
0           // m05 - не константа, инициализируется по умолчанию (0)
Купаев,1999 // p   - константа, инициализируется явно
Lipman,1953  // t   - не константа, инициализируется неявно

```

Обратите внимание: статический метод работает, хотя не определен ни один объект типа `StaticField`. Именно поэтому вызов метода пишется с префиксом — именем класса:

```
StaticField::print();
```

В отличие от этого метода, метод `print()` класса `Person` вызывается для конкретных объектов `p` и `t`.

Определения статических полей на первый взгляд противоречат всем принципам инкапсуляции: поля объявлены приватными, но значения им присваиваются «в открытую». Однако, во-первых, такая инициализация разрешена только в определении, которое обязан предоставить создатель класса, иначе программа не пройдет компоновку. Во-вторых, определение-то у нас единственное. Таким

образом, механизм защиты данных работает по-прежнему — присвоить значение приватной статической переменной «в открытую» в другом месте не получится. Теперь разберемся со статическими методами, которые существенно отличаются от обычных. Статические методы называют методами класса. Статический метод можно вызвать для объекта:

```
object.staticmethod()
```

Кроме того, статический метод можно вызывать для класса независимо от определения объектов класса:

```
class::staticmethod()
```

Так как статический метод не «приписан» к объекту, он не получает указателя `this` в качестве параметра. Ни конструкторы, ни деструктор, ни операция присваивания не могут быть статическими. Статические методы не могут быть константными. Статические методы не могут быть виртуальными (см. главу 9).

Статические методы позволяют решить одну небольшую проблему, которая иногда возникает при реализации конструкторов. Яркий пример — класс комплексных чисел. Комплексные числа имеют две формы записи: в декартовых и полярных координатах. Обе формы записи задаются парой действительных чисел, только в первом варианте эти числа являются абсциссой и ординатой, а во втором — радиусом и углом. Прототипы обоих конструкторов, очевидно, одинаковы:

```
Complex(const double &a, const double &b);
```

Таким образом, мы не можем написать в классе два конструктора. Для решения этой проблемы надо написать две статические функции с разными именами, которые вызывают закрытый конструктор. Это могло бы выглядеть так, как показано в листинге 4.22.

Листинг 4.22. Именованные «конструкторы»

```
class Complex
{ public:
    static Complex Decart(const double &re, const double &im);
    static Complex Polar (const double &r, const double &alpha);
// остальные члены класса
private:
    double Re, Im;
    Complex(const double &x, const double &y):Re(x), Im(y) {};
};
Complex Complex::Decart(const double &re, const double &im)
{ return Complex(re, im); }
Complex Complex::Polar (const double &r, const double &alpha)
{ return Complex(r*cos(alpha), r*sin(alpha)); }
```

Используются такие функции вполне традиционно:

```
const double pi = 3.1415926;
Complex a = Complex::Decart(1.0, 0.0);
Complex a = Complex::Polar (1.0, PI/2.0);
```

Это правильно работает именно потому, что статические функции могут быть вызваны *до* создания объекта, то есть до вызова конструктора — поэтому они этот конструктор и вызывают, создавая и возвращая объект.

Однако самое простое и, пожалуй, уже классическое применение статических полей — подсчет объектов. Для этого в классе объявляется статическое поле целого типа, которое увеличивается в конструкторе, а уменьшается в деструкторе. Это как раз тот случай, когда деструктор требуется определить явным образом, поскольку при уничтожении объекта нужно выполнять специфические действия. Таким способом можно собирать статистику интенсивности создания и уничтожения объектов некоторого класса (листинг 4.23).

Листинг 4.23. Подсчет объектов класса

```
#include <iostream>
class Object
{ static unsigned int count;          // статическое поле - счетчик
public:
    Object();                        // конструктор
    ~Object();                       // деструктор
    static unsigned int Count();      // выдача счетчика
};
unsigned int Object::count = 0;      // определение и инициализация счетчика
Object::Object() { ++count; }        // увеличение счетчика при создании
Object::~Object() { --count; }       // уменьшение счетчика при разрушении
unsigned int Object::Count()         // получение значения
{ return count; }
int main()
{ Object b;                          // объект только один
  cout << "Количество=" << Object::Count() << endl;
  { Object a[10];                    // создается 10 объектов
    cout << "Количество=" << Object::Count() << endl;
  }                                  // массив уничтожается
                                    // объект опять только один
  cout << "Количество=" << Object::Count() << endl;
  return 0;
}
```

В классе `Object` задано статическое поле-счетчик, которое увеличивается в конструкторе, то есть при каждом создании объекта. При вызове деструктора счетчик уменьшается. Метод `Count()` сделан статическим, так как по своей сути этот метод является методом всего класса, а не отдельного объекта.

Программа демонстрирует изменения счетчика при «рождении» и «смерти» объектов — на экран выводится следующее:

```
Количество=1          // создание объекта b
Количество=11         // создание массива a[10]
Количество=1          // вышли из блока - массив уничтожен
```

Сначала создается только один объект `b`; при входе во внутренний блок создается массив `a`, в котором 10 объектов, и для каждого конструктор увеличивает счетчик. При выходе из блока элементы массива уничтожаются, и деструктор при каждом вызове уменьшает счетчик.

Резюме

В качестве поля в классе может быть задан массив. Количество элементов массива должно определяться константным выражением, причем выражение должно быть определено до объявления поля-массива. Инициализировать поле-массив можно в теле конструктора. Инициализация массива с помощью списка инициализации конструктора ограничена: разрешается только инициализация нулем. Константный массив встроенного типа невозможно инициализировать никаким способом. При инициализации константного массива невстроенного типа для каждого элемента массива вызывается конструктор по умолчанию.

Обычно при наличии в классе полей-массивов перегружается операция индексирования оператор `[]`, которая возвращает ссылку на объект того же класса. Операция перегружается в двух вариантах: как константная и неконстантная.

Класс может содержать статические элементы: поля и методы, которые являются элементами класса, а не объекта. Статические поля в классе только объявляются, их требуется еще отдельно определить вне класса. При определении статические поля могут быть проинициализированы. Если явной инициализации нет, то статические поля класса по умолчанию инициализируются нулями, как и глобальные переменные. Статические поля не занимают место в классе. Статические методы не получают указателя `this` в качестве параметра, они не могут быть ни константными, ни виртуальными.

Контрольные вопросы

1. Можно ли объявить в классе поле-массив? Каким образом задается размер поля-массива?
2. Каким образом инициализируется поле-массив?
3. Какой результат должна возвращать перегруженная операция индексирования? Почему?
4. Почему операцию индексирования перегружают в двух вариантах?
5. Объясните проблемы, возникающие в случае аварийного завершения работы конструктора.
6. Как определяются статические поля класса?
7. Какие элементы класса можно объявлять статическими?
8. Можно ли объявить в классе статическую константу? А константный статический массив?
9. Какие статические поля можно инициализировать непосредственно в классе?
10. Как определяются статические поля? В какой момент работы программы выполняется инициализация статических полей?
11. Сколько места в классе занимают статические поля?
12. Чем отличается статический метод от обычного?

13. Какие методы класса не могут быть статическими?
14. Можно ли статический метод вызвать для объекта?
15. Какие варианты применения статических полей вы можете привести? А каким образом применяются статические методы?

Упражнения

1. Реализовать класс `Money` для работы с денежными суммами. Денежная сумма должна быть представлена массивом, каждый элемент которого представляет собой некоторый номинал и содержит количество купюр данного номинала. Номинал может принимать значения (в рублях): 0.01, 0.1, 0.5, 1, 2, 5, 10, 50, 100, 500, 1000. Обеспечить инициализацию несколькими способами: двумя целыми (рубли и копейки), дробным числом (общая сумма денег), массивом номиналов. Реализовать все арифметические операции, представленные в листинге 1.15, и соответствующие операции с присваиванием. Обеспечить вывод общей денежной суммы на экран и преобразование в дробное число.
2. Определить в классе `Money` из предыдущего упражнения перечислимый тип `nominal`, значения которого должны быть индексами в массиве с количеством купюр.
3. Создать класс `BitString` для работы с битовыми строками. Битовая строка должна быть представлена массивом типа `char` длиной 256 элементов. Один символ массива представляет собой один бит и принимает значение 0 или 1. Младший бит имеет младший индекс. Реализовать традиционные операции для работы с битами (`and`, `or`, `xor`, `not`), перегрузив подходящие операции. Обеспечить доступ к отдельному биту. Реализовать конструктор инициализации строкой, конструктор инициализации целым числом, начав с произвольного бита. Операция вывода должна выводить битовую строку в шестнадцатеричном виде.
4. Создать класс `Money` для работы с денежными суммами. Сумма должна быть представлена массивом, каждый элемент которого — десятичная цифра. Максимальная длина массива — 100 цифр, реальная длина задается в конструкторе. Младший индекс соответствует младшей цифре денежной суммы. Младшие две цифры — копейки.
5. Создать класс `Polinom` для работы с многочленами до 100-й степени. Коэффициенты должны быть представлены массивом из 100 элементов-коэффициентов. Младшая степень имеет меньший индекс (нулевая степень — нулевой индекс). Размер массива задается как аргумент конструктора инициализации. Реализовать арифметические операции и операции сравнения, вычисление значения полинома для заданного значения, дифференцирование, интегрирование.
6. Реализовать класс `Data` (см. упражнение 4 в главе 2), используя для представления месяцев массив структур. Структура имеет два поля: название месяца (строка), количество дней в месяце. Индексом в массиве месяцев является перечислимый тип `month`. Реализовать два варианта класса: с обычным и статическим массивами месяцев.

7. Создать класс `Fraction` для работы с дробными десятичными числами. Число должно быть представлено двумя массивами типа `unsigned char`: целая и дробная части, каждый элемент — десятичная цифра. Для целой части младшая цифра имеет меньший индекс, для дробной части старшая цифра имеет меньший индекс (десятые — в нулевом элементе, сотые — в первом, и т. д.). Реализовать арифметические операции сложения, вычитания и умножения, а также операции сравнения.
8. Реализовать класс `Rational` (см. упражнение 10 в главе 2) с перегрузкой операций и подсчетом объектов (см. листинг 4.23).
9. Добавьте код подсчета объектов в класс `Money` из упражнения 4.
10. Персональная карточка содержит фамилию и дату рождения. Реализовать класс `ListPerson` для работы с картотекой персоналий. Класс должен содержать массив персональных карточек. Реализовать методы добавления и удаления карточек, а также метод доступа к карточке по фамилии. Фамилии в массиве должны быть уникальными. Реализовать операцию объединения двух картотек (если карточка есть в обеих картотеках, включать только одну). Реализовать метод, выдающий по фамилии знак зодиака. Для этого в классе должен быть объявлен статический массив структур `Zodiac` с полями названия знака зодиака, а также дат начала и окончания периода. Индексом в массиве должен быть перечислимый тип `zodiac`. Для представления дат использовать упрощенный вариант класса `Date` (см. упражнение 6).

Глава 5

Динамическая память в C++

Было бы прекрасно, если бы программист мог писать программы, совсем не заботясь о памяти. Более того, в идеале программист и не должен ничего знать о том, где размещаются переменные программы. Но, к сожалению (или к счастью), мы живем в реальном мире и работаем на реальных компьютерах, объем памяти которых ограничен. И хотя некоторые языки программирования¹ пытаются скрыть от программиста факт наличия памяти, имитируя неограниченный ее объем, на практике обычно оказывается, что не учитывать конечность памяти просто нельзя — это плохо сказывается на «здоровье» программы.

Память и объекты

C++ по отношению к памяти «поступает» значительно «честнее»: уже на странице 4 стандарта описывается «модель памяти C++ (см. п. 1.7 в [1]). Согласно этой модели, основной единицей памяти в C++ является байт. Память, доступная программе, состоит из последовательности байтов. Каждый байт имеет уникальный адрес и состоит из последовательности битов. Однако стандарт не утверждает, что байт обязательно содержит 8 бит — напротив, количество битов в байте является особенностью реализации. Тем не менее одно требование к байту прописано явно: байт должен быть такого размера, чтобы в нем был способен поместиться любой символ из базового набора символов.

Соответственно (см. п. п. 5.3.3 в [1]), операция `sizeof()` выдает размер объекта или типа в байтах, а результат выполнения операций `sizeof(char)`, `sizeof(signed char)` и `sizeof(unsigned char)` равен 1. Размеры остальных встроенных типов зависят от реализации.

Каждый объект C++ является «областью памяти» (см. п. 1.8 в [1]). Объект обладает рядом свойств, например имеет имя и тип. Объекту приписывается

¹ Например, C#, Java, Lisp, Prolog.

(см. п. 3.7 в [1]) *период хранения в памяти* (storage duration). Это свойство определяет и время жизни объекта, и тип памяти, в которой хранится объект (см. п. 3.8 в [1]). Различают три периода хранения и в соответствии с этим — три класса памяти, в которой размещаются объекты:

- статическая память;
- автоматическая память;
- динамическая память.

В статической памяти размещаются все глобальные объекты, объявленные вне каких-либо функций и классов, а также любые статически переменные, в том числе объявленные в функциях и классах. Память для этих объектов выделяется в момент запуска программы, а уничтожаются такие объекты только при завершении программы.

В автоматической памяти размещаются локальные нестатические объекты, которые объявляются в любом блоке, в том числе в теле любой функции. В частности, переменные, объявленные в теле главной функции, тоже являются локальными. Память таким объектам выделяется в момент объявления, а уничтожаются они при выходе из области видимости. Естественно, наиболее подходящей реализацией является размещение локальных объектов в стековой памяти (хотя в стандарте об этом нет ни слова).

В стандарте (см. п. 12.5 в [1]) динамическая память называется «свободной». Динамическая память выделяется объектам во время выполнения программы с помощью специальных операций `new` и `new[]` (см. п. п. 5.3.4 в [1]). Возврат динамической памяти (и уничтожение объектов при этом) тоже выполняется специальными операциями `delete` и `delete[]` (см. п. п. 5.3.5 в [1]). В стандарте работа с динамической памятью описывается в разных местах (см. п. п. 3.7.3, 5.3.4, 5.3.5, 12.5, 17.4.3.4; 18.4, 20.4 в [1]), поэтому составить целостную картину непросто.

Управление динамической памятью

Мы уже неоднократно использовали операцию `new` — настало время разобраться с некоторыми подробностями. Начнем с самого простого — с создания динамических объектов встроенных типов:

```
int *p1 = new int;
```

Данный оператор выделяет `sizeof(int)` байт свободной памяти и записывает адрес этой памяти в переменную-указатель `p1`. Память никак не инициализируется, даже не обнуляется.

Однако `new` обладает гораздо более широкими возможностями. Чтобы присвоить значение динамической переменной, необходимо задать его после типа в круглых скобках, например:

```
int *p2 = new int(100);
```

В данном случае значение `*p2` равно 100. Вообще в скобках может стоять любое выражение, приводимое в данном случае к типу `int`. Если нужно просто обнулить выделяемую память, можно использовать конструкцию инициализации нулем:

```
int *p3 = new int();
```

С помощью операции `new` можно создать даже константный динамический объект, например:

```
const int *pp = new const int(3);
```

Конечно, константный объект обязан быть проинициализирован, однако ни Visual C++.NET 2003, ни C++ Builder 6 не выдают никаких сообщений об ошибке и даже предупреждений, если константный объект создается без инициализации.

```
const int *pp = new const int; // явная ошибка не диагностируется
```

Случайное изменение (например, `*pp=0`;) динамического константного объекта невозможно — компилятор следит за этим. Не проходит даже явная отмена константности:

```
const_cast<int>(*pp) = 1;
```

Компилятор сообщает, что не может преобразовать `const int` в «чистый» тип `int`. Его, конечно, можно «обмануть», подменив указатель, например, так:

```
int *p = const_cast<int *>(pp); *p = 0;
```

Однако на программистском сленге такой прием получил название «грязный хак» (*hack*), поэтому без крайней нужды его применять не следует. Да и зачем тогда создавать динамический объект как константный?

Уничтожение объектов и возврат памяти выполняется операцией `delete`, например:

```
delete p1;
```

Естественно, указатель должен быть именно тот, который использовался при создании динамического объекта операцией `new`. Константный динамический объект уничтожается точно так же:

```
delete pp;
```

Сама операция `delete` не обнуляет указатель, однако она «правильно» работает в том случае, если указатель уже нулевой — она просто ничего не делает. Поэтому проверять указатель на нуль перед уничтожением объекта нет необходимости.

С помощью операции `new[]` в C++ можно создавать динамические массивы, например:

```
int *p = new int[100];
```

При этом выделяется *непрерывная область* памяти, достаточная для размещения 100 целых чисел типа `int`. Инициализировать такой массив при создании, к сожалению, невозможно. Отсюда следует, что создавать константный динамический

массив встроенного типа смысла не имеет — мы не сможем его проинициализировать «легальными» способами¹:

```
const int *pp = new const int[100];
```

Хотя ни C++ Builder 6, ни Visual C++.NET 2003 никак не реагируют на создание такого массива, тем не менее попытки присвоить значение элементу массива «пресекают на корню». Вероятно, имеет смысл создавать динамический массив с константой-указателем, чтобы именно указатель случайно нельзя было изменить, например:

```
int * const pp = new int[100];
```

Основное преимущество динамического массива состоит в том, что количество элементов можно задавать выражением, вычисляемым во время выполнения программы. Более того, стандарт разрешает создавать динамический массив даже нулевого размера (см. п. п. 5.3.4/7 в [1]), например:

```
int t = 0; int *pt = new int[t];
```

Указатель `pt` получает корректный адрес, отличающийся от любого другого адреса. Таким образом, нет никакой необходимости проверять значение выражения на ноль перед созданием массива.

Уничтожение динамического массива осуществляется операцией `delete[]`. Например, удалить динамический массив `pt` следует так:

```
delete []pt;
```

Пустые квадратные скобки необходимы. Они говорят компилятору, что указатель адресует массив, а не единичный элемент. Отсутствие скобок не является синтаксической ошибкой (компилятор не выдает никаких сообщений), однако правильность выполнения программы не гарантируется.

Двухмерный динамический массив

Создание двухмерного массива также не представляет особой сложности, например:

```
int (*p2)[100] = new int[5][100];
```

Указатель `p2` содержит адрес начала массива из пяти элементов, каждый из которых является массивом из 100 целых. Размер элементов-массивов слева и справа, естественно, должен совпадать — иначе программа не транслируется из-за невозможности конвертирования типов.

Было бы здорово, если бы можно было создать двухмерный динамический массив произвольных размеров, например:

```
float *r = new float[m][n];
```

¹ Хотя в этом случае как раз стоит вспомнить об упомянутом «грязном хаке» и проинициализировать константный массив, который в дальнейшем можно использовать только как константный.

Однако такая запись является ошибкой: вычисляемым может быть только первое, самое левое измерение, остальные должны быть заданы константами. Обычно двумерные динамические массивы применяются при реализации матричной арифметики. И такое ограничение существенно затрудняет написание универсальных функций для работы с матрицами произвольных размеров. Эту проблему решают по-разному. Наиболее современное решение — это шаблоны, в которых размеры матриц задаются как параметры шаблона. Однако в этом случае объект-матрица будет иметь большой размер. Поэтому лучше все-таки использовать динамическую память. Сначала создается динамический массив указателей на массивы (строки матрицы), а затем каждый указатель инициализируется динамическим массивом для чисел (рис. 5.1). Таким образом, наш «главный» указатель является указателем на указатели:

```
int n; // количество строк матрицы
int m; // количество столбцов матрицы
float **M = new float *[n]; // массив указателей на строки матрицы
for(int i = 0; i < n; ++i)
    M[i] = new float[m]; // строки матрицы
```

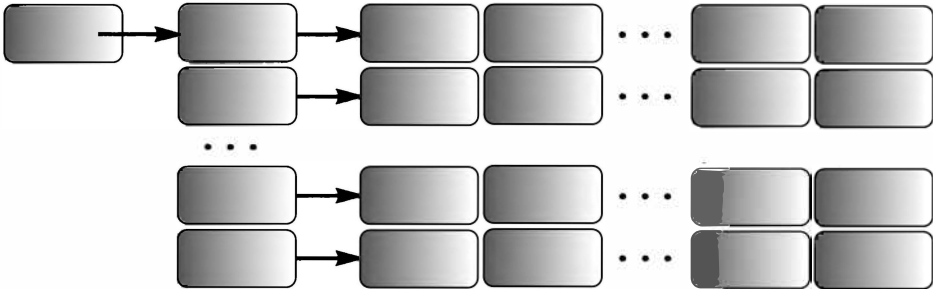


Рис. 5.1. Динамический двумерный массив

Работу по резервированию памяти может выполнять конструктор класса-матрицы. Тогда деструктор должен память возвращать. И это как раз тот случай, когда деструктор нужно реализовать явно.

Возврат памяти тоже осуществляется «постепенно»: сначала возвращаются массивы для чисел, а затем — массив указателей:

```
for (i=0; i<n; i++) delete[] M[i];
delete[] M;
```

POD-типы

Пока мы рассматривали только встроенные типы данных. Время жизни динамических объектов встроенных типов начинается с момента выделения памяти и заканчивается в момент ее возврата. Стандарт определяет (см. п. 3.9 в [1]), что объекты встроенных типов относятся к так называемым POD-объектам (Plain Old Data). Помимо встроенных, к POD-объектам относятся перечисления, указатели, массивы, структуры и объединения. Как видим, все типы, изначально

определенные в C, относятся к POD-типам. Что касается классов, то POD-классами являются только классы с *тривиальным конструктором*. Тривиальный конструктор — это конструктор, создаваемый автоматически, и класс при этом должен удовлетворять следующим условиям:

- класс не должен содержать виртуальных функций (см. главу 9);
- все поля в классе должны быть POD-объектами;
- если класс является наследником, то базовый класс должен иметь тривиальный конструктор, и наследование не должно быть виртуальным (см. главу 10).

На практике это означает, что простой класс, не являющийся наследником и не имеющий конструкторов и виртуальных функций, является POD-классом. В частности, пустой класс является POD-классом. Время жизни POD-объектов определяется так же, как и время жизни объектов встроженных типов.

Допустим, в классе не определены конструкторы:

```
struct Type
{ int t; };
```

Тогда возможны три способа создания одиночного динамического POD-объекта:

```
Type *p1 = new Type;           // по умолчанию
Type *p2 = new Type();         // по умолчанию - инициализация нулем
Type *p3 = new Type (*p2);     // копирование
```

В первом случае работает конструктор без аргументов, создаваемый по умолчанию; второй вариант тоже сопровождается вызовом конструктора без аргументов, однако дополнительно выполняется инициализация нулем; в третьем варианте вызывается автоматически создаваемый конструктор копирования.

NonPOD-типы

Любой класс, не удовлетворяющий перечисленным в предыдущем разделе условиям «POD-типизации», является nonPOD-типом. Стандарт определяет, что создание динамического nonPOD-объекта операцией `new` выполняется в два этапа:

1. Для объекта выделяется необходимый объем свободной памяти.
2. Вызывается конструктор для инициализации выделенной памяти.

Время жизни динамических nonPOD-объектов начинается только с момента окончания работы конструктора. Первый шаг выполняется функциями выделения памяти: для одиночного объекта — функцией `operator new()`, а для массива объектов — функцией `operator new[]()`. В стандарте (см. п. п. 3.7.3 и 18.4 в [1]) эти функции определены так:

```
void *operator new(std::size_t size) throw(std::bad_alloc);
void *operator new[](std::size_t size) throw(std::bad_alloc);
```

Следующая конструкция называется *спецификацией исключений* (см. главу 7):

```
throw(std::bad_alloc)
```

Эта конструкция означает, что в случае нехватки памяти генерируется стандартное исключение типа `bad_alloc`.

Отметим, что при невозможности выделить память конструктор не вызывается. Более того (см. п. п. 5.3.4/17 в [1]), в случае возникновения исключения во время инициализации объекта сначала выполняется возврат выделенной памяти, а затем начинается обработка исключения.

Уничтожение динамического объекта тоже выполняется в два шага: сначала вызывается деструктор, а затем — функция возврата памяти. Время жизни динамического объекта заканчивается, когда начинается выполнение кода деструктора. Функции возврата памяти определены в стандарте там же; где и функции выделения памяти, и имеют следующие прототипы:

```
void operator delete (void *) throw();  
void operator delete [] (void *) throw();
```

Пустая спецификация исключений `throw()` показывает, что функция не генерирует никаких исключений.

Как и в случае создания динамических POD-объектов, динамический объект произвольного класса можно создать без инициализации:

```
Type *p1 = new Type;  
Type *p2 = new Type();
```

При наличии в классе определенного конструктора без аргументов после выделения памяти в данном случае вызывается именно он. Кроме того, мы можем инициализировать динамические объекты (аналогично объектам встроенных типов), если в классе определен конструктор инициализации.

При создании динамического массива объектов конструктор без аргументов вызывается для каждого элемента массива. Подчеркнем еще раз, что вызывается именно конструктор по умолчанию, а не конструктор инициализации.

Создание динамических объектов-констант ничем не отличается от создания динамических констант встроенных типов. Конечно, даже динамически создаваемую константу требуется инициализировать (для этого в классе должен быть определен конструктор инициализации), однако и Visual C++.NET 2003, и C++ Builder 6 «пропускают» отсутствие инициализации. Например, без всяких сообщений транслируется такое объявление:

```
const Type *p1 = new const Type;
```

Естественно, в этом случае во время выполнения вызывается конструктор без аргументов.

Операция `delete` осуществляет уничтожение динамических объектов: сначала вызывается деструктор, а затем возвращается память. Возврат памяти выполняется функцией `operator delete()` для одиночного объекта и функцией `operator delete[]()` для массива. Эти функции не генерируют исключений — это гарантирует, что операция уничтожения объекта не преподнесет никаких сюрпризов — ведь деструктор тоже не должен генерировать исключений. При уничтожении массива деструктор вызывается для каждого элемента удаляемого массива.

Еще одна форма операции new

В стандарте определена еще одна форма операции new, позволяющая обойтись без исключений. Герб Саттер в [22] называет эту форму операции new: «обычная старая new». Работает она так же, как new в языке C: в случае невозможности выделить память возвращается нулевой указатель. Чтобы этот вид операции new работал, необходимо подключить заголовок:

```
#include <new>
```

В стандарте прототипы функций определены так (см. п. п. 18.4 в [1]):

```
void *operator new(std::size_t size, const std::nothrow_t &) throw();
void *operator new[](std::size_t size, const std::nothrow_t &) throw();
```

Спецификация исключений throw() как раз и говорит о том, что данная форма new не генерирует никаких исключений. Вызов такой операции new несколько необычен:

```
int *pn = new (std::nothrow) int[10000000];
```

Имя nothrow — это обычно имя пустого класса, определенного в стандартном пространстве имен std (см. главу 13). При такой форме обращения указатель pn получит значение NULL — из-за отсутствия памяти исключение генерироваться не будет. Естественно, это нужно проверять:

```
if(pn != NULL) // if (!pn) - можно и так
{ // действия при успешном выделении памяти
}
```

Возвращают такую память обычным образом — с помощью соответствующей формы операции delete:

```
void operator delete(void *ptr, const std::nothrow_t &) throw();
void operator delete[](void *ptr, const std::nothrow_t &) throw();
```

Несмотря на «страшный» прототип, использование этой формы операции delete ничем не отличается от показанной ранее. Заметим, что даже при отсутствии выделенной памяти, когда указатель равен нулю, вызов операции delete совершенно безопасен.

Размеры динамических объектов

В главе 1 мы узнали, сколько памяти выделяют системы программирования для объектов классов и структур. Не меньший интерес представляет вопрос об объеме выделяемой динамической памяти. К сожалению, стандартных средств, позволяющих выяснить размеры выделяемой памяти, практически нет. Операция sizeof() позволяет выяснить только размеры памяти, выделяемой одиночному объекту, и не может вычислить размер динамического массива. Пусть в программе определена пустая структура:

```
struct Empty {};
```

Тогда следующий фрагмент программы (Visual C++.NET 2003) выдаст на экран 1 и 1, хотя во втором случае выделено не менее 101 байта памяти:

```
Empty *p3 = new Empty();
cout << sizeof(*p3) << endl;           // вывели размер
delete p3;
Empty *p5 = new Empty[101];
cout << sizeof(*p5) << endl;
delete[] p5;
```

Обычно в библиотеках любой интегрированной среды реализовано множество различных функций для работы с динамической памятью, которые не являются стандартными в C++. Мы их рассматривать не будем, за исключением одной функции — `_msize()`, которая позволяет узнать реальный размер выделенного блока динамической памяти. Ее прототип:

```
size_t _msize (void *block);
```

Она реализована и в Visual C++.NET 2003, и в C++ Builder 6. Используем ее для вывода размера выделяемой динамической памяти (листинг 5.1).

Листинг 5.1. Размеры динамической памяти

```
#include <iostream>
#include <malloc.h>                               // для _msize()
using namespace std;
//#pragma pack(1)
struct Empty {};
struct Type { int t; char ch; };
// главная функция
int main()
{
    Type *p0 = new Type();
    cout << "Type =" << _msize(p0) << endl;
    cout << "Type =" << sizeof(*p0) << endl;;
    delete p0;
    p0 = new Type[13];
    cout << "Type[13]=" << _msize(p0) << endl;
    cout << "Type[13]=" << sizeof(*p0) << endl;;
    delete p0;
    //-----
    Empty *p3 = new Empty();
    cout << "Empty=" << _msize(p3) << endl;
    cout << "Empty=" << sizeof(*p3) << endl;
    delete p3;
    Empty *p5 = new Empty[13];
    cout << "Empty[13]=" << _msize(p5) << endl;
    cout << "Empty[13]=" << sizeof(*p5) << endl;
    delete[] p5;
    Empty m[13]; cout << sizeof(m) << endl;
    getch();
    return 0;
}
```


В системе Visual C++.NET 2003 эта программа в режиме выравнивания по умолчанию выдает следующие цифры:

```
Type =8
Type =8
Type[13]=104
Type[13]=8
Empty=1
Empty=1
Empty[13]=13
Empty[13]=1
13
```

Те же величины выдаются и в режиме выравнивания по границе байта. А вот в системе C++Builder 6 выдаются совсем другие числа. Без выравнивания по границе байта:

```
Type =12
Type =8
Type[13]=104
Type[13]=8
Empty=12
Empty=8
Empty[13]=104
Empty[13]=8
104
```

В режиме выравнивания по байту (`#pragma pack(1)`):

```
Type =12
Type =5
Type[13]=68
Type[13]=5
Empty=12
Empty=1
Empty[13]=16
Empty[13]=1
13
```

Хотя мы не можем с уверенностью сказать, что функция `_msize()` показывает реально выделенный объем динамической памяти (обычно помимо памяти непосредственно для данных выделяется еще память для управляющей информации), однако даже поверхностное сравнение — в пользу Visual C++. NET 2003.

«Умный» массив

Теперь, вооружившись знаниями о динамическом выделении памяти, реализуем с помощью класса более «умный» массив числового типа. Встроенные массивы слишком просты и ненадежны. Наш массив должен быть «поумнее» — нужно обеспечить проверку индекса при обращении к элементу массива. И конечно, такой «массив» можно будет передавать в функции любым из возможных способов (по значению, по ссылке, по указателю) и возвращать в качестве результата.

Операции, которые со встроенными массивами должны выполняться в цикле, можно «упрятать» внутрь соответствующих методов и перегруженных операций. Например, практически обязательно реализовать операцию присваивания, чтобы можно было писать так:

```
a = b;
```

Запись совершенно естественная и понятная любому программисту: массив *a* становится копией массива *b*. Для числового массива естественно реализовать также арифметические операции. Например, умножение каждого элемента массива на некоторый коэффициент может выглядеть так:

```
a *= 1.34;
```

Здесь, очевидно, *a* является «умным» массивом, каждый элемент которого умножается на число 1.34. Ту же операцию можно реализовать с двумя массивами, например:

```
a *= b;
```

Это выражение представляет собой умножение элементов массива *a* на соответствующие элементы массива *b*. Если размер массива вычисляется методом `size()`, то операция реализуется с помощью простого цикла:

```
if (a.size() == b.size())
    for (int i = 0; i < a.size(); ++i)
        a[i] = a[i] * b[i];
```

Очевидно, «упрятывание» операций существенно упростит работу с массивами и значительно повысит надежность программ. Инкапсуляция (и полиморфизм!) — в действии!

Набор операций с «умным» массивом может быть очень широким — все зависит от требований задачи и фантазии разработчика. Например, сортировку можно реализовать, перегрузив подходящую унарную операцию (например, `~` или `!`). Единственным аргументом такой операции, очевидно, является текущий объект — «умный» массив.

Реализуем массив с элементами типа `double`, а тип самого массива пусть называется `TArray`. Возможный интерфейс класса приведен в листинге 5.2. Как обычно, пусть операции с присваиванием являются методами класса, а соответствующие операции без присваивания — дружественными функциями. Операция индексирования указана дважды, как и в классе `TString`.

Листинг 5.2. Интерфейс «умного» массива

```
class TArray
{ public:
    typedef unsigned int UINT;                // для краткости
    // конструкторы
    TArray(UINT size, double k=0.0);
    TArray(const double *begin, const double *end);
    TArray(const TArray &a);                  // конструктор копирования
    TArray(const TArray &a, UINT begin, UINT k);
    ~TArray();                                // деструктор
```

продолжение 

Листинг 5.2 (продолжение)

```

// индексирование
double& operator[](UINT index);
const double& operator[](UINT index) const;
// присваивание
TArray& operator=(const TArray &rhs);
TArray& assign(const TArray &a, UINT begin, UINT k);
TArray& assign(const double *begin, const double *end);
// Методы-операции
TArray& operator+=(const double &a);           // аргумент - выражение
TArray& operator-=(const double &a);
TArray& operator*=(const double &a);
TArray& operator/=(const double &a);
TArray& operator%=(const double &a);
TArray& operator+=(const TArray &rhs);        // аргумент - массив
TArray& operator-=(const TArray &rhs);
TArray& operator*=(const TArray &rhs);
TArray& operator/=(const TArray &rhs);
TArray& operator%=(const TArray &rhs);
UINT size() const;                           // количество элементов
double& max_value();                          // "левая" функция
double& min_value();                          // "левая" функция
double summa();                               // сумма всех элементов
double summa(UINT l, UINT r);                 // сумма элементов
double product();                             // произведение всех элементов
double product(UINT l, UINT r);               // произведение элементов
void operator!();                             // сортировка
double* find(const double &a);                // поиск - "левая" функция
// скалярное произведение массивов
friend double product_dot(const TArray &a, const TArray &b);
friend TArray operator+(const TArray&a, const TArray &b);
friend TArray operator*(const TArray&a, const TArray &b);
friend TArray operator-(const TArray&a, const TArray &b);
friend TArray operator/(const TArray&a, const TArray &b);
friend TArray operator%(const TArray&a, const TArray &b);
friend ostream& operator <<(ostream& to, const TArray &a);
friend istream& operator >>(istream& to, TArray &a);
private:
    double *data;                             // динамический массив
    UINT size_array;                           // количество элементов
};

```

Конструкторы

Начнем с конструкторов. Очевидно, конструктор без аргументов нам ни к чему — лучше всегда задавать количество аргументов явно, тем самым объявляя массив фиксированной длины. Вторым аргументом этого конструктора может быть значение, которым инициализируются элементы массива. По умолчанию это значение равно нулю.

Реализация конструктора не представляет труда, однако неясным остается важный вопрос: что делать, когда клиент задает неправильный размер, например

нулевой¹ или даже отрицательный? Конечно, ни один программист «в здравом уме и ясной памяти» не напишет отрицательную константу в качестве размера массива, однако длина нашего массива может задаваться с помощью переменной или даже выражения, которое вычисляется во время выполнения и приводится к типу параметра-размера. Поэтому иметь защиту от такой ошибки просто необходимо. Очевидно, что создавать массив в этом случае нельзя, поэтому просто аварийно завершим программу, выдав соответствующее диагностическое сообщение. Аналогично поступим и при реализации других конструкторов. Конечно, это — не лучшее решение, поэтому в главе 7 мы перепишем конструкторы с использованием аппарата обработки исключений. Текст конструктора представлен в листинге 5.3.

Листинг 5.3. Самый простой конструктор

```
TArray::TArray(UINT size, double k)
{
    if (size > 0)                                // защита "от дурака"
    {
        size_array = size;                       // размер массива
        data = new double[size_array];           // создали массив
        for(UINT i = 0; i < size_array; ++i)      // заполняем массив
            data[i] = k;                         // значение по умолчанию
    }
    else                                          // неправильная длина
    { cout << "Not size > 0!"; abort(); }
}
```

Значение по умолчанию прописано в прототипе конструктора, показанном в интерфейсе класса TArray.

Второй конструктор инициализации позволит нам конструировать наш массив из встроенного. Этот конструктор имеет два аргумента-указателя — это обеспечит нам использование любой последовательности элементов исходного встроенного массива для конструирования нашего «умного» (листинг 5.4). Вспомним аналогичный конструктор в классе TString (см. листинг 4.5).

Листинг 5.4. Конструктор инициализации из встроенного массива

```
TArray::TArray(const double *begin, const double *end)
{
    if (begin < end)                             // защита "от дурака"
    {
        size_array = (end - begin);              // количество элементов
        data = new double[size_array];           // создаем массив
        for(UINT i = 0; i < size_array; ++i)      // заполняем массив
            data[i] = *(begin+i);               // копируем из массива
    }
    else                                          // неправильные параметры
    { cout << "Not (begin < end) !"; abort(); }
}
```

Нужно отметить, что в этом конструкторе *обязательно* должно выполняться условие (*begin < end*) — последний задаваемый элемент не включается. Это позволяет задавать весь массив парой (*begin*, *begin* плюс количество). Почему

¹ Хотя стандарт разрешает задавать нулевой размер динамического массива, в данном случае это, очевидно, явная ошибка.

мы именно так трактуем аргументы, поясним на простом примере. Пусть объявлен массив:

```
double t[10] = {0,1,2,3,4,5,6,7,8,9};
```

Тогда весь массив `t` задается парой `(t, t+10)`, а последовательность элементов с `i`-го до `j`-го, исключая `j`-й, — парой `(t+i, t+j)`. Такое объявление является естественным в C++ и избавляет нас от возможных ошибок типа «плюс-минус единичка».

Этот конструктор дает возможность объявлять наши «умные» массивы, инициализируя их значениями встроенных массивов, например:

```
double one[5] = {5,4,3,2,1};
TArray One (one, one+5); // весь массив
double two[] = {1,2,3,4,5,6};
TArray Two (two, two + sizeof(two)/sizeof(double)); // весь массив
TArray Three (one+1, one+2); // один элемент
TArray Four (&two[1], &two[2]); // один элемент
```

Массивы `One` и `Two` становятся «копиями» массивов `one` и `two` соответственно. А вот массив `Three` содержит только один элемент массива `one`, равный четырем. Так как параметры конструктора — указатели, то при создании массива можно задавать любые адреса, удовлетворяющие условию (`begin < end`). Например, массив `Four` содержит один элемент массива `two`, равный двум.

Еще один конструктор позволит нам создавать и инициализировать новый «умный» массив из последовательности элементов уже существующего, что обеспечивается индексом первого элемента последовательности и количеством элементов (листинг 5.5).

Листинг 5.5. Конструктор инициализации из другого «умного» массива

```
TArray::TArray(const TArray &a, UINT begin, UINT k)
{
    if ((begin < a.size() &&
        (k <= a.size() - begin)) // защита от ошибок
    {
        size_array = k; // количество элементов
        data = new double[size_array]; // создаем массив
        for(UINT i = 0; i < size_array; ++i) // заполняем массив
            data[i] = a.data[i + begin]; // начиная с элемента begin
    }
    else // неправильные параметры
    {
        cout << "Not ((0 <= begin < size) or (0 < k < size)) !"; abort();
    }
}
```

Как обычно, вначале проверяются параметры. Если с ними все в порядке, то создается и заполняется динамический массив.

Конструктор копирования описан далее в разделе «Копирование и присваивание». При наличии конструкторов можно объявлять «умные» массивы разнообразными способами:

```
double a[] = {9,1,2,3,22,2,4,7,5};
TArray A(a, a+(sizeof(a)/sizeof(double))); // из массива
TArray B(A); // конструктор копирования
TArray D = B; // конструктор копирования
```

```

TArray G (B,2,5);           // вырезка из B
TArray H(15);               // 15 элементов = 0
TArray F(25, -1);           // 25 элементов = -1

```

Массив *A* инициализируется элементами встроенного массива *a*. Массивы *B* и *D* создаются конструктором копирования. Массив *G* — это «вырезка» из массива *B*, в *G* прописываются 5 элементов *B*, начиная с *B*[2]. Пятнадцать элементов массива *H* обнуляются, а 25 элементов массива *F* равны -1 .

Как уже отмечалось, параметры конструкторов «умного» массива могут быть переменными, например:

```

TArray::UINT size = (sizeof(a)/sizeof(double))*2; // size = 18
TArray H(size); // 18 элементов = 0
TArray F(size-8, a[4]); // 10 элементов = 22
TArray G(H, 2, size-10); // 8 элементов H

```

Массив *G* состоит из восьми элементов массива *H*, начиная с *H*[2]. Здесь необходимо обратить внимание только на то, что тип *UINT*, определенный в классе *TArray*, необходимо писать с префиксом класса.

Еще раз нужно подчеркнуть, что в случае неправильных параметров в конструкторах программа заканчивается аварийно, так как обработать ошибки в конструкторах без использования механизма исключений невозможно. Более того, если выделение памяти закончится аварийно, генерируется *стандартное* исключение типа *bad_alloc* (см. п. п. 15 и 18.4 в [1]), которое мы тоже не обрабатываем. В этом случае утечки памяти не происходит: память просто не выделяется, и объект не создается.

Деструктор

Каждый из конструкторов создает динамический массив и заполняет его значениями. Доступ к динамическому массиву выполняется по закрытому указателю *data*. Однако выделенную память надо возвращать, чтобы в нашем классе не возникала утечка памяти. Клиент этого делать не может, так как доступа к указателю у него нет. Да и нельзя этот доступ клиенту давать — этим мы грубо нарушим принцип инкапсуляции. Возвращать память должен сам наш класс.

Как раз для этих целей и предназначен деструктор (см. главу 2). Напомним: конструктор создает объект, деструктор его уничтожает (см. п. 12.4 в [1]). Деструктор вызывается автоматически *при каждом уничтожении объекта*. Для локальных объектов это делается при выходе из блока, а для динамических, которые клиент создает посредством операции *new* (или *new[]*), — при каждом вызове операции *delete* (или *delete[]*). Текст деструктора приведен в листинге 5.6.

Листинг 5.6. Деструктор

```

TArray::~TArray()
{ delete[]data; // освобождаем память
  data = 0;
}

```

Деструктор очень короткий, поэтому его можно реализовать непосредственно в классе. Единственное его назначение (и действие) — возвращать память, которую запрашивает конструктор при создании массива. Обнуление указателя, как написано у нас, делать не обязательно.

Отметим, что мы не писали деструкторы ни для класса `TMoney` (см. листинг 1.15), ни для класса `TString` (см. листинг 4.2). Если деструктор не реализован, создается стандартный деструктор по умолчанию (см. п. п. 12.4/3 в [1]), который и вызывается при уничтожении объекта. Такой деструктор «ничего не делает» [2]. Для нашего «умного» динамического массива автоматический деструктор не подходит, так как он не выполняет операцию `delete[]` — мы это должны делать сами явным образом.

Вообще деструкторы необходимо определять для каждого класса, конструкторы которого запрашивают какие-нибудь ресурсы, подлежащие возврату. Чаще всего — это память. Но мы увидим далее, что деструкторы полезны и в некоторых других случаях.

Операция индексирования

При реализации операции индексирования мы тоже не можем обработать неправильный параметр таким образом, чтобы сообщить клиентской программе об ошибке. В данном случае, в отличие от ситуации с классом `TString` (см. главу 4), у нас нет специального значения, которое можно было бы возвращать в качестве результата. Единственный вариант обработки ошибки в данном случае — завершать программу аварийно, что и показано в листинге 5.7.

Листинг 5.7. Перегрузка операции индексирования с аварийным завершением

```
double& TArray::operator[](UINT index)
{ if (index<size_array) return data[index];
  else { cout << "Index out of range!"; abort(); }
}
const double& TArray::operator[](UINT index) const
{ if (index<size_array) return data[index];
  else { cout << "Index out of range!"; abort(); }
}
```

Нам опять требуются две функции, так как при отсутствии второй мы не сможем работать с константными параметрами. Например:

```
TArray operator*(const TArray &a, const TArray &b);
```

Попытаемся написать в этой дружественной функции умножения массивов выражение поэлементного умножения:

```
result[i] = a[i] * b[i];
```

В этом случае в системе Visual C++.NET дважды выдается ошибка трансляции C2678:

```
binary '[' : no operator found which takes a left-hand operand of type
'const TArray' (or there is no acceptable conversion)
```

Система жалуется именно на выражение `a[i]*b[i]`, так как `a` и `b` указаны в заголовке функции как константные массивы.

Вообще-то говоря, для индексирования мы можем использовать любую бинарную операцию, например `|` (битовая операция ИЛИ), однако этого не следует делать. Программист, который попытается использовать наш умный массив, будет часто ошибаться, так как общепринятый смысл и реализация в C++ этой операции — совсем не индексирование. В C++ есть другая операция, которая очень хорошо подходит для реализации доступа к элементам — это операция вызова функции `()`. Фактически в листинге 5.7 надо только заменить одни скобки другими (листинг 5.8).

Листинг 5.8. Перегрузка операции `()` для индексирования

```
double& TArray::operator()(UINT index)
{ if (index < size_array) return data[index];
  else { cout << "Index out of range!"; abort(); }
}
const double& TArray::operator()(UINT index) const
{ if (index < size_array) return data[index];
  else { cout << "Index out of range!"; abort(); }
}
```

Тогда обращение к элементам массива будет выглядеть так:

```
result(i) = a(i) * b(i);
```

Для C++ это несколько необычная запись, однако в языках Фортран и Бейсик, например, индексы пишутся именно в круглых скобках.

Нужно еще сказать, что применение операции `()` для индексирования одномерного массива не даёт никаких преимуществ по сравнению с обычной операцией `[]`. А вот для индексирования многомерных массивов как раз удобнее использовать круглые скобки. Дело в том, что операция `[]` — бинарная, поэтому в скобках может быть только один операнд-индекс. Это создает некоторые чисто технические неудобства при индексировании многомерных массивов. Поэтому проще задействовать операцию `()`, так как количество ее аргументов может быть произвольным. Мы еще вернемся к вопросам перегрузки операции `()`.

Копирование и присваивание

Теперь разберемся с копированием наших массивов. Нам необходимо явно реализовать копирующий конструктор, так как стандартный делает совсем не то, что требуется. В стандарте C++ (см. п. п. 12.8/8 в [1]) прямо сказано, что конструктор копирования, создаваемый по умолчанию, выполняет поэлементное копирование полей класса. Такое копирование называется *поверхностным* (shallow copying). На языке C++ реализация конструктора копирования для нашего класса `TArray` по умолчанию является такой, как показано в листинге 5.9.

Листинг 5.9. Конструктор копирования по умолчанию для класса `TArray`

```
TArray::TArray(const TArray &t)
{ this->size_array = t.size_array;
  this->data = t.data;
}
```

Тогда следующее объявление приводит к ситуации, показанной на рис. 5.2:

```
TArray A (B);
```


Такое поведение конструктора приводит к ошибкам типа «висячая ссылка» при уничтожении объекта. При уничтожении локальных объектов деструкторы выполняются в порядке, обратном порядку вызова конструкторов. Поэтому сначала уничтожается более «молодой» объект А. Деструктор уничтожаемого объекта возвращает память, а вторая ссылка «провисает». При уничтожении второго объекта — массива В — поведение программы непредсказуемо. Наименьшим злом будет аварийное завершение программы.

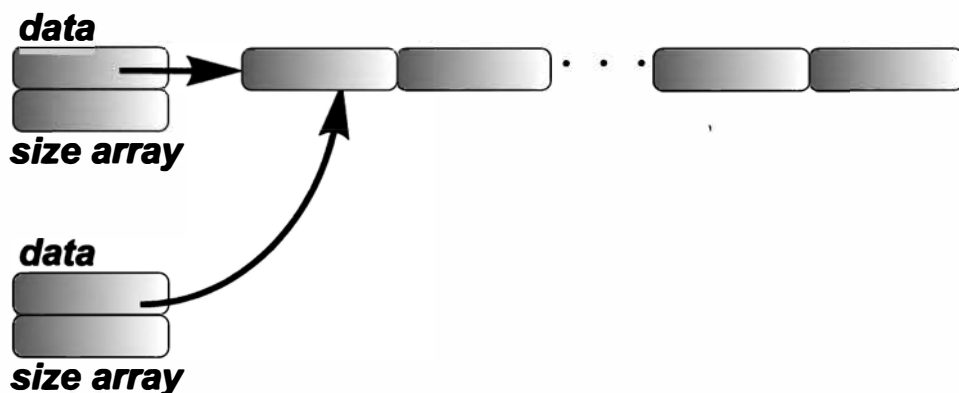


Рис. 5.2. Результат работы стандартного конструктора копирования

Нам нужно, чтобы конструктор делал совсем другое: сначала создал динамический массив такого же размера, что и копируемый, а потом скопировал в новый массив элементы инициализирующего массива. Такое копирование называется *глубоким* (deep copying). Текст конструктора показан в листинге 5.10.

Листинг 5.10. Конструктор копирования для класса TArray

```
TArray::TArray(const TArray &t)
{
    this->size_array = t.size_array;           // размер
    this->data = new double[size_array];       // новый массив
    for(UINT i = 0; i < size_array; ++i)      // копируем
        data[i] = t.data[i];
}
```

Эта ситуация принципиально отличается от предыдущей — у нас имеется отдельная копия динамического массива для каждого объекта (рис. 5.3), поэтому деструкторы сработают корректно.

С учетом того, что копируемый массив имеет заведомо корректные значения полей, для инициализации полей нового массива можно воспользоваться списком инициализации.

```
TArray::TArray(const TArray &t)
: size_array(t.size_array), data(new double[size_array])
{
    for(UINT i = 0; i < size_array; ++i)
        data[i] = t.data[i];
}
```

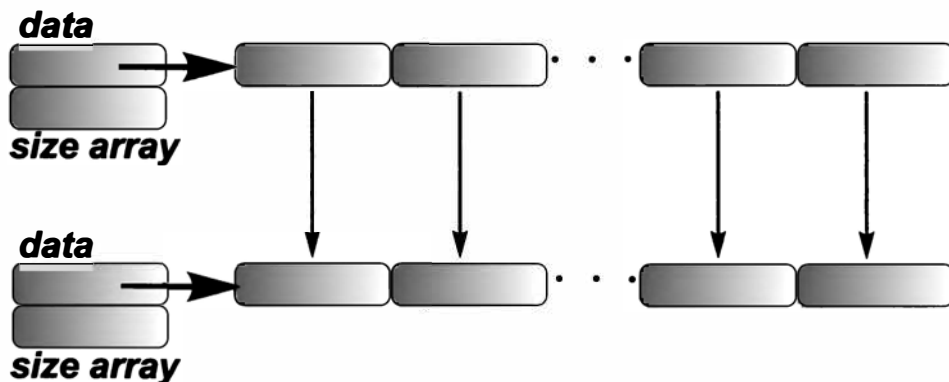


Рис. 5.3. Правильная работа конструктора копирования

До сих пор мы, не задумываясь, пользовались и стандартной операцией присваивания, которая выполняется при отсутствии явного определения операции. Прототип этой операции для некоторого типа *T* выглядит так (см. п. п. 12.8/10 в [1]):

```
T& operator=(const T& );
```

Такая операция, как и копирующий конструктор, выполняет поэлементное копирование правого операнда в левый (см. п. п. 12.8 и 13.5.3 в [1]). Пока мы не использовали в классах динамическую память, все прекрасно работало. Однако стандартная операция присваивания для класса *TArray* работает неправильно, как и стандартный конструктор копирования. Но если стандартный конструктор провоцировал только ошибки типа «висячая ссылка», то стандартная операция присваивания создает еще и ситуацию утечки памяти — ее иллюстрирует рис. 5.4. Поэтому нам надо реализовать собственную операцию, чтобы ее работа соответствовала рис. 5.3.

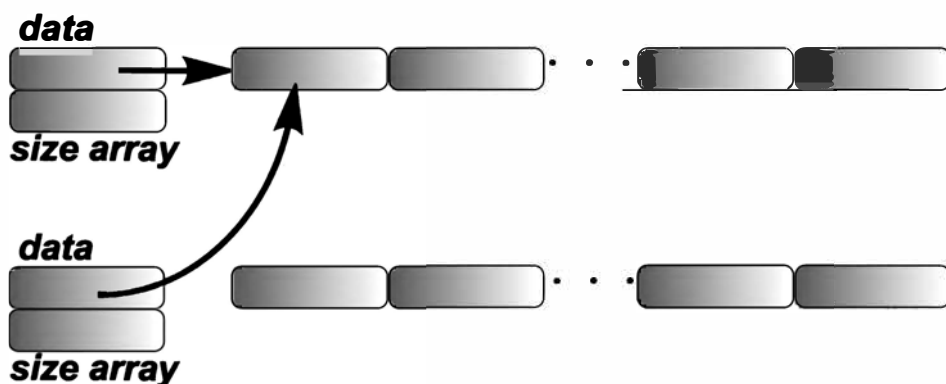


Рис. 5.4. Неправильная работа стандартной операции присваивания

Наша операция должна создать новый динамический массив, скопировать туда элементы присваиваемого массива и *возвратить* системе память прежнего динамического массива. Именно этим операция присваивания отличается от инициализации копированием: при инициализации нет необходимости возвращать память.

Операция присваивания, очевидно, должна возвращать ссылку на TArray, так как левый аргумент (текущий объект) стоит слева от операции и изменяется. Ссылка обеспечивает нам возможность многократного присваивания. Ставший уже классическим текст перегруженной операции присваивания для динамического класса представлен в листинге 5.11.

Листинг 5.11. «Каноническая» реализация операции присваивания

```
TArray& TArray::operator=(const TArray &t)
{ if (this != &t) // отслеживаем самоприсваивание
  { size_array = t.size_array; // определили размер
    double *new_data = new double[size_array];
    for(UINT i = 0; i < size_array; ++i)
      new_data[i] = t.data[i]; // копируем
    delete[] data; // вернули предыдущий массив
    data = new_data; // вступили во владение
  }
  return *this;
}
```

В этой функции надо обратить внимание на проверку операции самоприсваивания: `this != &t`

Это выражение позволяет нам не выполнять никакой работы, если программист напишет присваивание массива самому себе:

```
a = a;
```

Очевидно, в этом случае не требуется создавать новый динамический массив и копировать в него элементы.

Приведенный вариант реализации операции присваивания прекрасно работает и делает именно то, что нам нужно, — создает копию массива. Однако существует другое, гораздо более элегантное решение, которое показал Герб Саттер в [21]. Нам необходимо реализовать функцию обмена полей класса TArray (листинг 5.12). Аргумент у такой функции только один, так как вторым является текущий объект.

Листинг 5.12. Функция обмена с текущим объектом

```
void Swap(TArray &other)
{ std::swap(data, other.data); // стандартная функция обмена
  std::swap(size_array, other.size_array);
}
```

Можно было бы написать обмен полей и явным образом:

```
double *d = data; data = other.data; other.data = d;
UINT s = size_array; size_array = other.size_array; other.size_array = s;
```

Однако мы использовали функцию `swap()`, входящую в стандартную библиотеку¹, о чем сигнализирует префикс `std::`.

СОВЕТ

Всегда, когда возможно, используйте средства стандартной библиотеки.

¹ Надо подключить стандартную библиотеку алгоритмов директивой `#include <algorithm>`.

Функцию `Swap` можно поместить в приватную часть класса `TArray`. С ее помощью операция присваивания реализуется очень просто (листинг 5.13).

Листинг 5.13. Реализация операции присваивания «по Саттеру»

```
TArray& TArray::operator=(const TArray &t)
{ TArray temp(t);           // временный локальный объект temp = t
  Swap(temp);               // обмен полями с текущим объектом
  return *this;              // возврат текущего объекта
}                             // объект temp уничтожен
```

В этом варианте всю работу делают конструктор копирования и деструктор (которые, как обычно, уже отлажены и проверены). Сначала создается локальный объект `temp`, который инициализируется массивом-аргументом, затем выполняется обмен полей текущего объекта с полями объекта `temp`. Результат обмена представлен на рис. 5.5.

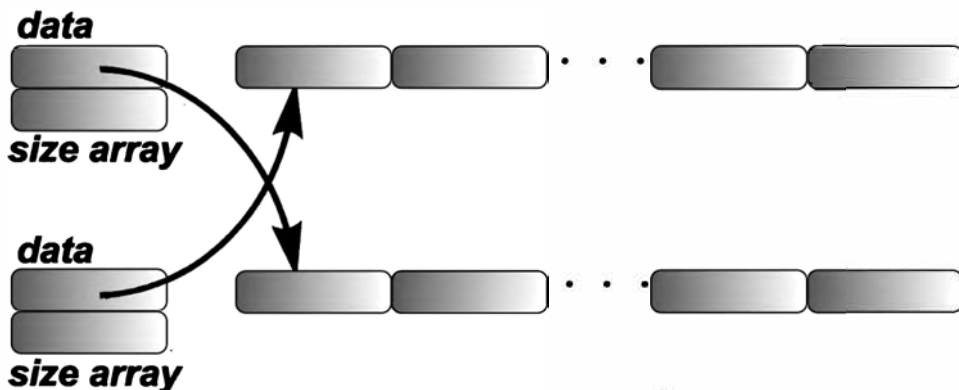


Рис. 5.5. Результат обмена полей

При завершении работы локальный объект уничтожается, вызывается деструктор, и память возвращается системе.

С помощью функции обмена легко реализуются методы присваивания `assign()` (листинг 5.14).

Листинг 5.14. Методы присваивания `assign`

```
TArray& TArray::assign(const TArray &t, UINT l, UINT r)
{ TArray temp(t,l,r);       // локальный объект-массив
  Swap(temp);               // обмен полями
  return *this;              // возврат текущего объекта
}                             // локальный объект уничтожен
TArray& TArray::assign(const double *begin, const double *end)
{ TArray temp(begin, end);
  Swap(temp);
  return *this;
}
```

Как видим, основная работа выполняется конструктором.

Реализация методов

Реализация операций с присваиванием и соответствующих бинарных операций сложности не представляет¹, поэтому покажем в качестве примера реализацию трех операций умножения (листинг 5.15). Остальные операции реализуются точно так же — надо только заменить знак операции сложения.

Листинг 5.15. Реализация операций сложения

```
TArray& TArray::operator+=(const double &rhs)
{ for(UINT i = 0; i<size_array; ++i)
    data[i] *= rhs;                // складываем с текущим объектом
  return *this;
}

TArray& TArray::operator+=(const TArray &rhs)
{ if (size_array == rhs.size_array) // защита
  { for(UINT i = 0; i<size_array; ++i)
      data[i] *= rhs.data[i];      // изменяем текущий объект
  }
  return *this;
}

TArray operator+(const TArray &a, const TArray &b)
{ TArray temp = a;                // левый аргумент
  if (a.size() == b.size())        // защита
    temp += b;                    // добавляем - наша операция
  return temp;
}
```

При умножении массивов необходимо проверять равенство размеров. Если количество элементов не совпадает, то возвращается левый аргумент операции. Во втором методе это — текущий объект. В последнем случае создается локальный объект, которому и присваивается левый массив. Если размеры левого и правого массивов совпадают, то правый массив добавляется к локальному массиву. В противном случае он остается без изменения и возвращается в качестве результата.

В дополнение к этим функциям можно реализовать еще два варианта дружественных функций сложения массива с числом:

```
friend TArray operator+(const TArray &a, const double &b);
friend TArray operator+(const double &a, const TArray &b);
```

Два варианта необходимо, чтобы не было проблем с коммутативностью.

Для примера покажем еще реализацию операций, вычисляющих различные произведения элементов массивов (листинг 5.16). Аналогично реализуются и методы суммирования.

Во втором методе `product()` используется реализация первого метода, причем сначала, как обычно, создается локальный объект.

Последняя функция вычисляет скалярное произведение двух массивов. Если размеры массивов не совпадают, то умножения не происходит и возвращается сумма элементов левого аргумента.

¹ Мы пока не обращаем внимания на эффективность реализации.

Листинг 5.16. Методы и функции произведений

```
double TArray::product()
{ double p = 1.0;
  for(UINT i = 0; i<size_array; ++i)
    p*=data[i];           // работаем с текущим массивом
  return p;
}
double TArray::product(UINT begin, UINT end)
{ TArray temp(*this, begin, end); // создаем локальный массив из текущего
  return temp.product();         // работает предыдущий метод
}
double product_dot(const TArray &a, const TArray &b)
{ TArray temp(a);              // локальный объект из левого аргумента
  if (a.size() == b.size())     // защита
    temp *= b;                  // поэлементное умножение
  return temp.summa();          // вычисление суммы локального массива
}
```

Для реализации метода сортировки используем стандартный алгоритм `sort()`, входящий в состав стандартной библиотеки, — посмотрите, как просто это делается (листинг 5.17).

Листинг 5.17. Сортировка массива

```
void TArray::operator!()
{ std::sort(data, data+size_array); }
```

Обратите внимание: в качестве параметров сортировки мы задали два указателя именно таким способом, какой использовали в конструкторе, показанном в листинге 5.4.

ПРИМЕЧАНИЕ

Поскольку мы использовали в реализации класса `TArray` функции стандартной библиотеки, мы должны подключить их директивой `#include <algorithm>`.

Реализация дружественных функций ввода-вывода вполне традиционна — никаких подсказок в функциях не выводится, это должна делать программа-клиент (листинг 5.18).

Листинг 5.18. Реализация дружественных функций ввода-вывода

```
ostream& operator <<(ostream& to, const TArray &a)
{   for(TArray::UINT i = 0; i<a.size(); ++i)
    to << a.data[i] << ' ';           // выводим через пробел
  return to;
}
istream& operator >>(istream& to, TArray &a)
{   for(TArray::UINT i = 0; i<a.size(); ++i)
    to >> a.data[i];
  return to;
}
```

При выводе элементы массива отделяются пробелом. При вводе можно разделять элементы массива пробелом или завершать ввод каждого числа нажатием клавиши `Enter`.

Использование «умного» массива

Разнообразные конструкторы, набор операций и методов позволяют нам управлять с «умными» массивами так же, как и со встроенными типами: объявлять, вводить, выводить, вычислять и присваивать новые значения элементам и массиву в целом. Например, мы можем объявить константный массив:

```
double a[5] = {1,2,3,4,5};  
const TArray U(a,a+5);
```

Ни элементы этого массива, ни сам массив `U` нельзя будет изменить никаким образом — все попытки пресекаются компилятором.

Интересно, что наш «умный» массив может быть элементом обычного массива C++. Однако ограничением является то, что объявляемый массив C++ обязательно требуется инициализировать; например, такое объявление без инициализации не транслируется:

```
TArray R[4];
```

Причиной является отсутствие в классе `TArray` конструктора без аргументов. Но и при инициализации возникают аналогичные проблемы — попробуем явно указать количество элементов массива C++:

```
TArray R[4] = { 5,6,3,4 }; // R — это обычный массив!!!
```

Тогда количество инициализирующих выражений должно ему соответствовать. Если инициализирующих выражений будет меньше, то опять возникает ошибка из-за отсутствия конструктора без аргументов. А если же инициализирующих выражений больше, то выдается «обычная» ошибка «слишком много инициализаторов». Таким образом, массив лучше объявлять без явного указания количества элементов, например:

```
TArray R[] = { 5,6,3,4 };
```

Элементами массива `R` будут 4 «умных» массива, однако не совсем ясно, какой при этом вызывается конструктор. Если мы выполним программу в отладочном режиме по шагам с заходом внутрь вызываемых функций¹, то увидим, что во всех случаях вызывается один и тот же конструктор, аргументом которого является длина «умного» массива (см. листинг 5.3). Поэтому массив `R` будет содержать 4 разных массива типа `TArray`: массив `R[0]` состоит из пяти элементов, массив `R[1]` включает 6 элементов, а массивы `R[2]` и `R[3]` — 3 и 4 элемента соответственно. Причем все элементы всех «умных» массивов по умолчанию равны нулю.

Если мы хотим инициализировать элементы «умных» массивов другими значениями, нам придется явно вызывать конструкторы, например:

```
double d[5] = {1,2,3,4,5};  
TArray R[] = { TArray (5,1), TArray (d, d+5), TArray (R[1]) };
```

¹ В системе Visual C++.NET 2003.

В данном случае массив R включает 3 «умных» массива, каждый из которых содержит 5 элементов. Проинициализированы они явно и разными конструкторами, причем для инициализации последнего используется предыдущий элемент — «умный» массив.

Доступ к элементам такого двумерного массива выполняется обычным образом, например:

```
R[0][0] = 11.11;
```

Нулевой элемент «умного» массива R[0] получает значение 11.11. Следующее присваивание тоже допустимо, так как в данном случае работает перегруженная операция присваивания класса TArray:

```
R[0] = U;
```

Вывод массива R на экран можно выполнять с помощью обычного двукратного вложенного цикла, например:

```
for(int i=0; i<3; ++i)
for(int j=0; j<R[i].size(); ++j)
    cout << R[i][j] << endl;
```

Все 15 чисел выведутся в столбик. Однако, вспомнив о том, что операция вывода << перегружена для «умного» массива, мы можем записать вывод проще:

```
for(int i=0; i<3; ++i) cout << R[i] << endl;
```

Тогда на экране появятся 3 строки по 5 элементов.

Еще раз нужно отметить, что массив R *не является* «умным»: он «не знает» своей длины, с ним нельзя выполнить ни одну из операций, которые «понимает» наш «умный» массив. Однако каждый элемент R — это «умный» массив.

«Умный» массив можно передать в функцию в качестве параметра любым способом и вернуть в качестве результата. Давайте напишем простую функцию-фильтр, которая получает массив и диапазон значений, а возвращает новый массив, состоящий из элементов, значения которых входят в диапазон. Прототип функции, очевидно, должен быть таким:

```
TArray array(TArray source, double L, double R)
```

Мы передаем массив по значению. Нужно помнить, что при передаче параметра по значению, тип которого не является встроенным, вызывается конструктор копирования. При передаче по ссылке или по указателю такого не происходит. Однако и в данном случае передача по значению отнюдь не означает, что в стек функции копируется весь массив. Если мы выясним размер класса TArray с помощью операции sizeof(), то окажется, что класс занимает всего 8 байт¹ — именно столько памяти отводится под два поля (беззнаковое целое и указатель) нашего класса. Поэтому в данном случае можно не заботиться об экономии памяти и времени, передавая массив по ссылке. Реализация функции представлена в листинге 5.19.

¹ В системе Visual C++.NET 2003 с атрибутами выравнивания по умолчанию.

Вспомним, однако, наш класс `TString` (см. листинг 4.2). Так как в качестве поля в классе прописан символьный массив, то при передаче параметра по значению как раз выполняется копирование всего массива в стек.

ВНИМАНИЕ

Для обоснованного выбора способа передачи контейнера в качестве параметра необходимо знать внутреннее устройство контейнера.

Поэтому общепринятым способом является передача контейнера по константной ссылке — это в любом случае гарантирует отсутствие накладных расходов на память и время.

Листинг 5.19. Функция-фильтр «умного» массива

```
typedef TArray::UINT uint;           // для краткости
TArray array(TArray source, double L, double R)
{ uint count = 0;                   // количество элементов
  for(uint i = 0; i < source.size(); ++i)
    if ((L<=source[i])&&(source[i]<=R)) ++count; // считаем
  TArray result(count);             // массив-результат
  for(uint i = 0, j = 0; i < source.size(); ++i)
    if ((L<=source[i])&&(source[i]<=R))
      { result[j]=source[i]; ++j; } // копируем
  return result;                    // возвращаем массив
}
```

Функция сначала считает количество элементов исходного массива, попадающих в заданный диапазон. Затем создается локальный массив требуемого размера и заполняется нужными значениями. Полученный массив возвращается в качестве результата.

Наш массив реализован только для элементов типа `double`, что является довольно жестким ограничением. В следующей главе мы поговорим о том, как создать более универсальную динамическую структуру.

Резюме

Память в C++ измеряется байтами. Операция `sizeof(char)` выдает результат, равный 1. Объекты C++ размещаются в трех видах памяти: статической, автоматической и динамической. Вид памяти фактически определяет и время жизни объекта. Для размещения одиночных объектов в динамической памяти используется операция `new`, для уничтожения — операция `delete`. Динамические массивы создаются операцией `new[]`, уничтожаются операцией `delete[]`. При создании динамического массива количество элементов можно задать выражением, вычисляемым во время выполнения. Однако для многомерных динамических массивов разрешается задавать вычисляемое выражение только для самого левого измерения — остальные должны быть заданы константами. Размеры выделяемой динамической памяти зависят от реализации.

Все типы в C++ делятся на POD-типы и nonPOD-типы. Для последних при создании динамических объектов сначала выделяется память, а потом вызываются конструкторы (без аргументов или инициализации), а при уничтожении — сначала вызывается деструктор, а потом возвращается память. При создании массива nonPOD-типа конструктор и деструктор вызываются для каждого элемента массива.

В стандарте определено несколько разных форм операций `new` и `delete`. Наиболее надежная форма операции `new` при отсутствии памяти генерирует исключение `bad_alloc`. Реализована и форма без генерации исключений, возвращающая при отсутствии памяти нулевой указатель.

Класс, запрашивающий для своих нужд динамическую память, обычно должен делать это в конструкторах. Тогда деструктор должен память возвращать. При реализации динамического класса практически обязательна реализация конструктора копирования и операции присваивания, так как конструктор копирования и операция присваивания, генерируемые по умолчанию, осуществляют «поверхностное» копирование, что приводит к висячим и потерянным ссылкам.

При реализации динамического массива обязательна перегрузка операции индексирования, которая в этом случае может проверять значение задаваемого индекса. Вместо операции `operator[]` можно использовать операцию вызова функции `operator()`, которая прекрасно подходит для индексирования многомерного массива. Для динамических массивов можно реализовать набор полезных операций, спрятав циклы внутрь методов. Кроме того, такие массивы можно передавать в качестве параметров и возвращать как результат.

Контрольные вопросы

1. Что является единицей памяти в C++? Какие требования к размеру единицы памяти прописаны в стандарте C++?
2. В каких единицах выдает результат операция `sizeof`? Какие типы данных имеют размер 1?
3. Какие три вида памяти входят в модель памяти C++?
4. Какие формы операций `new` и `delete` вы знаете? В чем их различие?
5. Какие типы являются POD-типами? В чем различие работы механизма выделения и освобождения памяти с помощью операций `new` и `delete` при работе с POD- и nonPOD-объектами?
6. С помощью каких операций в C++ выделяют память?
7. Чем отличается операция `new` от операции `new[]`?
8. Почему для динамических классов-контейнеров деструктор надо писать явным образом?
9. С помощью каких операций в C++ осуществляется возврат динамической памяти?

10. Что такое «глубокое копирование» и когда в нем возникает необходимость? Какое копирование осуществляет стандартный конструктор копирования?
11. Чем отличается копирование от присваивания?
12. Объясните, почему в реализации операции присваивания требуется проверка присваивания самому себе.
13. Сколько аргументов у операции присваивания?
14. Можно ли в качестве операции индексирования использовать операцию вызова функции ()? В чем ее преимущества перед операцией []?
15. Почему необходимо писать два определения операции индексирования? Чем они различаются?

Упражнения

Во всех заданиях обязательно должны быть реализованы инициализирующий конструктор, конструктор копирования, деструктор, ввод-вывод. Подходящие операции реализуются как методы класса, а остальные — как внешние дружественные функции.

1. Взяв за образец класс динамического массива `TArray` (см. листинг 5.2), создать класс `DoubleArray` — одномерный массив действительных чисел с задаваемыми границами индексов. Должны допускаться и отрицательные индексы. Определить методы и операции: количество элементов, получение элемента по индексу, присвоение элемента, присвоение массива, умножение на скаляр, максимальный и минимальный элементы, сумма элементов, среднее арифметическое.
2. Добавить к классу `DoubleArray` из упражнения 1 операции поэлементного сложения, вычитания, умножения и деления. Операции должны проверять совпадение длины массивов.
3. Используя как образец класс `TArray` (см. листинг 5.2), модифицировать класс `TString` из главы 4 (см. листинг 4.2), сделав его динамическим. Операции вставки, удаления и сцепления должны резервировать новый динамический массив и переписывать в него результат операции.
4. Создать класс `BitString` (см. упражнение 3 в главе 4), используя динамический массив типа `char`. Операции с битовыми строками должны «уметь» работать со строками разной длины.
5. Модифицировать класс `Decimal` из упражнения 6 в главе 1, реализовав его с использованием динамического массива типа `char`.
6. Реализовать класс `Money` (см. упражнение 4 в главе 4) с помощью динамического массива. Операции должны «уметь» работать с разными размерами динамического массива.
7. Реализовать класс `Set` (множество) типа `int`. Множество должно обеспечивать включение элемента в множество, исключение элемента из множества,

объединение, пересечение и вычитание множеств, вычисление мощности множества, проверку присутствия элемента в множестве, проверку включения одного множества в другое. Операции, которые изменяют количество элементов в множестве, должны создавать новый динамический массив и записывать в него результат.

8. Используя как образец класс `TAarray` (см. листинг 5.2), реализовать класс динамического массива с элементами типа `double` без арифметических операций. Написать функцию, принимающую в качестве аргументов два указателя на обычный массив типа `double` и возвращающую динамический массив как результат. Исходный массив заполнить случайными числами в диапазоне от -30 до $+70$. Умножить каждое число на минимальное. Добавить в массив-результат сумму и среднее арифметическое по абсолютной величине.
9. Написать функцию, получающую в качестве аргументов указатели на массив типа `int` и возвращающую объект-множество (см. упражнение 7) в качестве результата.
10. Реализовать класс `ListPerson` (см. упражнение 10 в главе 4) с использованием динамического массива.

Глава 6

Контейнеры

В реальных задачах обычно требуется обрабатывать группы данных довольно большого объема. Например, посчитать зарплату для всех сотрудников университета или транслировать программу, состоящую из нескольких тысяч строк исходного текста. Поэтому в любом языке программирования, в том числе и в C++, существуют средства объединения данных в группы — обычно это массивы. Однако в C++ массивы, как было отмечено в предыдущей главе, являются слишком простыми, а потому очень ненадежными конструкциями, и для преодоления их ограничений мы разработали «умный» массив. Однако одних массивов как средств объединения однородных данных явно недостаточно. В процессе развития языков программирования, и C++ в частности, программистское сообщество выработало более общую конструкцию объединения однородных данных в группу — *контейнер*. Несколько лет назад употреблялся термин «коллекция», однако с момента принятия стандарта C++ термин «контейнер» стал общепринятым — именно так называются наборы данных в стандартной библиотеке шаблонов (STL).

Характеристики контейнеров

В языке C++ (да и в любом другом) контейнер — это набор однотипных элементов. Я не случайно употребил слово «набор», а не «множество», так как множество — это тоже контейнер. По этому определению массив — это контейнер. Каталог файлов на диске — тоже контейнер.

Каждый контейнер характеризуется, в первую очередь, своим именем и типом входящих в него элементов. Имя контейнера — это имя переменной в программе, которое подчиняется правилам видимости C++. Как объект, контейнер должен обладать временем жизни в зависимости от места и времени создания, причем время жизни контейнера в общем случае не зависит от времени жизни его элементов.

Тип контейнера складывается из типа самого контейнера и типа входящих в него элементов. Тип контейнера — это не тип его элементов. Как правило, тип контейнера определяет способ доступа к элементам. Тип элементов может быть либо встроенным, либо реализованным. В том числе элементами контейнера могут быть контейнеры. Например, элементами некоторого каталога могут быть каталоги. Список строк тоже может служить примером контейнера контейнеров, так как отдельную строку можно считать контейнером символов.

И наконец, надо сказать о размере контейнера. Размер контейнера может быть либо определен при объявлении, либо не задан. В первом случае получаем контейнер фиксированной длины. Именно таким контейнером является «умный» массив (см. листинг 5.2), который не изменяет количество своих элементов за время жизни. Однако в общем случае количество элементов контейнера с заданной фиксированной длиной может изменяться от нуля до объявленного количества. Пример нашего класса строк `TString` (см. листинг 4.2) показывает, что размер контейнера (255 элементов) и количество элементов в нем (определяемое методом `Length()`) — это разные вещи.

Если же размер контейнера не задается, то, естественно, количество элементов контейнера изменяется во время работы программы. Элементы добавляются в контейнер и удаляются из него. Такой контейнер является контейнером переменного размера.

Доступ к элементам контейнера

Одной из важнейших характеристик контейнера является *доступ* к его элементам. Обычно различают *прямой*, *последовательный* и *ассоциативный* доступ. Прямой доступ к элементу — это доступ по номеру (или, еще говорят, по индексу) элемента. Именно таким образом мы обращаемся к элементам массива, например:

```
v[7]
```

Это выражение означает, что мы хотим оперировать элементом контейнера `v`, имеющим номер (индекс) 7. Нумерация элементов может начинаться, вообще говоря, с любого числа, однако в C++ принято нумерацию начинать с нуля, так как для встроенных массивов (которые являются частным случаем контейнера) принята именно такая нумерация.

Последовательный доступ отличается тем, что мы не имеем в распоряжении индексов элементов, зато можем перемещаться последовательно от элемента к элементу. Можно считать, что существует невидимая «стрелка»-индикатор, которую перемещают по элементам контейнера с помощью некоторого множества операций. Тот элемент, на который в данный момент «стрелка» показывает, называется текущим.

Обычно набор операций для последовательного доступа включает операции:

- перехода к первому элементу;
- перехода к последнему элементу;
- перехода к следующему элементу;

- перехода к предыдущему элементу;
- перехода на n элементов вперед (от первого в сторону последнего элемента контейнера);
- перехода на n элементов назад (от конца к началу контейнера);
- получения (изменения) значения текущего элемента.

Эти операции могут быть представлены в функциональной форме, например:

```
next(v);           // перейти к следующему
prev(v);           // перейти к предыдущему
first(v);          // перейти к первому
last(v);           // перейти к последнему
current(v);        // получить текущий
forward(v, n);     // перейти на n элементов вперед
back(v, n);        // перейти на n элементов назад
```

Операция изменения текущего элемента — это, естественно, операция присваивания, например:

```
current(v) = value;
```

В этом случае функция `current()` должна возвращать ссылку на элемент контейнера.

Те же операции, реализованные как методы класса (контейнера), можно представить следующим образом:

```
v.next();          // перейти к следующему
v.prev();          // перейти к предыдущему
v.first();         // перейти к первому
v.last();          // перейти к последнему
v.current();       // получить текущий
v.skip(n);         // перейти на n элементов вперед
v.skip(-n);        // перейти на n элементов назад
```

Однако в C++ «стрелку»-индикатор удобнее представить в виде некоторого объекта, связанного с контейнером. Если этот объект имеет имя `iv`, то те же операции могут быть реализованы и так:

```
iv = v.begin();    // перейти к первому
iv = v.end();      // перейти к последнему
++iv;             // перейти к следующему
--iv;             // перейти к предыдущему
iv+=n;            // перейти на n элементов вперед
iv-=n;            // перейти на n элементов назад
*iv               // получить значение текущего элемента
```

Не правда ли, очень похоже на указатель?! Однако это не указатель¹ — в практике объектно-ориентированного программирования такой объект называется итератором. *Итератор* — это объект, обеспечивающий последовательный доступ к элементам контейнера. Так же как контейнер представляет собой более общую

¹ Обычным указателем такой объект будет только для контейнера-массива.

концепцию, чем массив, так и итератор является более общей концепцией, чем указатель. В [17] итератор описан как один из шаблонов (*паттернов*) программирования — *Iterator*.

Ассоциативный доступ похож на прямой, однако основан не на номерах элементов, а на содержимом элементов контейнера. Например, в банковской системе контейнер может содержать записи о счетах клиентов. Обязательным элементом записи является поле, содержащее фамилию клиента, например:

```
class TAccount
{
    string Family;           // фамилия
    unsigned long Count;     // номер счета
    Date Open;              // дата открытия счета
public:
    // ...
};
```

Если контейнер *v* содержит такие объекты, то выражение представляет собой счет, открытый на имя Стенли Липпмана:

```
v["Lippman"]
```

Такое выражение в C++ является вполне корректным, так как операция индексирования может быть перегружена для аргумента любого типа.

Поле, с содержимым которого *ассоциируется* элемент контейнера, называется *ключом* (полем доступа). Элемент контейнера, соответствующий некоторому значению ключа, обычно так и называется значением. Ассоциативный контейнер, таким образом, состоит из множества пар «ключ-значение». Как правило, ассоциативный контейнер упорядочен некоторым образом по ключу. В данном случае контейнер, содержащий счета клиентов, отсортирован по полю *Family*, поэтому элементом, предшествующим записи с фамилией *Lippman*, может быть запись с фамилией *Кираев*, а следующим — запись с фамилией *Martin*.

Методы доступа к контейнеру — настолько важная характеристика, что в стандартной библиотеке (см. п. 17 в [1]) различают контейнеры *последовательные* и *ассоциативные*. Последовательными контейнерами, которые обеспечивают и прямой, и последовательный варианты доступа, являются контейнеры *vector* и *deque*, а ассоциативным — контейнер *map*. Контейнер, в котором доступ только последовательный, — это *list*.

Операции с контейнером

Все операции с контейнером можно разделить на несколько групп:

- операции доступа к элементам, включая операцию замены значений элементов;
- операции добавления и удаления отдельных элементов или групп элементов;
- операции поиска элементов и групп элементов;
- операции с контейнером как объектом; в частности, важными являются операции объединения контейнеров;
- прочие (специальные) операции, зависящие от вида контейнера.

Операции доступа мы уже рассмотрели. Операции добавления и удаления элементов работают только для контейнера переменного размера. Очевидно, что эти операции с элементами можно выполнять самыми разными способами:

- добавлять и удалять элементы в начале контейнера;
- то же самое делать в «хвосте» контейнера;
- вставлять элементы перед текущим элементом или после него, удалять текущий элемент;
- делать вставки в соответствии с некоторым порядком сортировки элементов контейнера, в этом случае обязательно выполняется операция поиска;
- удалять элемент, содержимое которого равно заданному, в этом случае «за кадром» тоже работает операция поиска.

Первые три операции обычно применяются к последовательным контейнерам. Способ вставки и удаления определяет вид последовательного контейнера. Если вставка и удаление выполняются только на одном конце контейнера, то такой контейнер называется *стеком*, а работает он в соответствии с дисциплиной обслуживания LIFO (Last In First Out — последним вошел, первым вышел). Говорят, что текущий элемент находится на вершине стека. Если же элементы добавляются на одном конце, а удаляются из другого, контейнер называется *очередью*. Очередь работает в соответствии с дисциплиной обслуживания FIFO (First In First Out — первым пришел, первым ушел). Можно выполнять и вставку, и удаление на обоих концах контейнера — такой контейнер называется *деком* (от английского термина *deque*¹, который является аббревиатурой от «double ended queue», то есть «очередь с двумя концами»). Таким образом, дек представляет собой обобщение очереди и стека.

Если же контейнер ассоциативный, то он упорядочен по полю доступа, поэтому операции вставки и удаления всегда выполняются в последних вариантах. Но и последовательный контейнер может быть отсортирован, поэтому операция вставки тоже может вставлять «по порядку». Примером отсортированного последовательного контейнера является приоритетная очередь `priority_queue` — один из последовательных контейнеров стандартной библиотеки.

Все операции, которые мы рассматривали до сих пор, — это операции с отдельными элементами контейнера. Но и с самим контейнером или его частью можно выполнять некоторые операции — вспомните класс `TString` (см. листинг 4.2). Обычно строки можно инициализировать другими строками и присваивать. Строки можно объединять разными способами, можно выполнять много разных операций с подстроками. Аналогично — и с контейнерами. Наиболее часто используется операция объединения двух контейнеров с получением нового контейнера, которая может быть реализована в различных вариантах:

- простое *сцепление* двух контейнеров, в новый контейнер попадают все элементы и первого, и второго контейнеров; операция не коммутативна;

¹ Этот термин, ставший ныне общепринятым, впервые использовал Дональд Кнут.

- ❑ объединение упорядоченных контейнеров, называемое *слиянием*, в новый контейнер попадают все элементы первого и второго контейнеров; объединенный контейнер упорядочен; операция коммутативна;
- ❑ объединение двух контейнеров как объединение множеств, в новый контейнер попадают только те элементы, которые есть хотя бы в одном контейнере; операция коммутативна;
- ❑ объединение двух контейнеров как пересечение множеств, в новый контейнер попадают только те элементы, которые есть в обоих контейнерах; операция коммутативна.

Одной из операций с контейнером является извлечение из него части элементов и создание из них нового контейнера. Часто эту операцию выполняет конструктор, а требуемая часть контейнера задается двумя итераторами.

Отдельно необходимо сказать о контейнерах-множествах. *Множество* — это контейнер, в котором каждый элемент единственный. Помимо операций объединения и пересечения, для контейнеров-множеств реализуется операция вычитания множеств: в контейнер-результат попадают только те элементы первого контейнера, которых нет во втором; операция не коммутативна. Очень часто с множествами выполняется операция проверки включения, которая фактически является операцией поиска (как отдельного элемента, так и подмножества элементов).

Тип элементов оказывает существенное влияние на то, какие операции могут выполняться с контейнером. Например, для строк операция сортировки обычно не нужна, а для числовых контейнеров или для списка счетов в банке такая операция может быть не только полезной, но и необходимой. Для числовых контейнеров, очевидно, часто необходимы операции поиска минимума и максимума, суммы и произведения элементов контейнера — как в нашем «умном» массиве (см. листинг 5.2).

Реализация контейнеров

Контейнеры, как правило, реализуются с помощью указателей и динамической памяти (см. п. п. 3.7.3 в [1]). Использование указателей и динамических переменных в классах в сочетании с перегрузкой операций представляет собой удивительно мощный механизм создания новых типов данных — контейнеры стандартной библиотеки тому наглядный пример.

При реализации контейнеров нам предстоит решить несколько важных вопросов. Во-первых, каким образом выделяется память?

Выделение памяти операцией `new[]` обеспечивает выделение непрерывной области памяти. Количество элементов обычно задается выражением, вычисляемым во время работы программы. Такая форма практически всегда используется для реализации динамических массивов — именно так был реализован «умный» массив (см. листинг 5.2). Способ выделения памяти настолько важен, что даже в стандарте указано, что контейнер `vector` (см. п. п. 23.2.4 в [1]), входящий в стандартную библиотеку C++, должен быть реализован с помощью операции `new[]`.

Второй способ распределения памяти — выделение одиночного элемента операцией `new`. Выделение памяти для одиночного элемента обычно требуется для реализации контейнеров с переменным количеством элементов. В этом случае не только память для элемента выделяется динамически, но и в состав самого элемента входят один или несколько (чаще всего два) указателей для связи элементов друг с другом. Обычно такие контейнеры либо последовательные, либо ассоциативные.

Как правило, выделением памяти занимается конструктор контейнера. Если память выделяется, то надо ее возвращать — иначе в программе возникает утечка памяти. Решение этой проблемы и составляет ответ на второй вопрос: каким образом вернуть память системе. Обычно эту работу «возлагают» на деструктор. Как мы знаем, операции возврата памяти являются «парными» для операций выделения памяти. Если память выделялась операцией `new`, то возвращать память нужно операцией `delete`. Если же память выделялась массивом (операцией `new[]`), то и возвращать ее нужно соответствующей операцией `delete[]`.

И наконец, третий вопрос связан с копированием и присваиванием. Для динамических классов, в которых используются указатели, предлагаемые по умолчанию операции совершенно не подходят, требуется явная реализация — как для «умного» массива (см. листинги 5.10, 5.11 и 5.13).

Последовательный контейнер

В предыдущей главе мы реализовали функцию-фильтр для «умного» массива (см. листинг 5.19). Наша функция-фильтр дважды просматривает массив — это не слишком хорошо. Однако подобный режим работы является следствием того, что «умный» массив «не умеет» изменять свой размер. Если бы в нашем распоряжении был контейнер, к которому можно было бы присоединять элементы по мере необходимости, то функция выглядела бы так:

```
TArray array(TArray source, double L, double R)
{
    TArray result; // массив-результат
    for(uint i = 0; i < source.size(); ++i)
        if ((L<=source[i])&&(source[i]<=R)) // если в диапазоне
            result+=source[i]; // прицепляем элемент
    return result; // возвращаем массив
}
```

Как видите, «способность» изменять размеры по мере необходимости может существенно повлиять на быстродействие — очевидно, что при большом исходном массиве эта функция будет работать в два раза быстрее, чем предыдущая.

Кроме того, наш «умный» массив представляет собой контейнер фиксированной длины, доступ к элементам которого осуществляется по индексу. Последовательный доступ реализуется в цикле последовательным изменением индекса. Чтобы разобраться, каким образом выполняется последовательный доступ с помощью итераторов, реализуем контейнер-дек с помощью двусвязного списка. Типом контейнера по сложившейся уже традиции пусть будет `TDeque`. Не задумываясь пока об общности, опять задействуем дробные числа типа `double`.

Вложенные классы

Простейшая структура элемента¹ последовательного контейнера очевидна:

```
struct Elem
{ double item;           // информационная часть
  // связующая часть
  Elem *next;            // следующий элемент
  Elem *prev;            // предыдущий элемент
};
```

Чтобы обеспечить инициализацию создаваемого элемента, добавим конструктор инициализации:

```
Elem(const double &a):item(a){}
```

Отметим, что программа-клиент, которая собирается использовать дек, ничего не должна знать о внутренней структуре узла. Чтобы это обеспечить, инкапсулируем узел в классе-контейнере и реализуем его в приватной части класса `TDeque` в качестве *вложенного* (см. п. 9.7 в [1]).

Вложенным называется класс, объявленный внутри другого класса. Он является членом объемлющего класса, а его определение может находиться в любой из секций `public` или `private`. Уровень вложенности стандартом не ограничивается.

Вложенный класс по умолчанию не имеет доступа к приватным элементам объемлющего класса, так же как объемлющий класс — к приватным элементам вложенного. Чтобы это было возможно, нужно использовать механизм друзей, например:

```
class Outer
{ //...
  friend class Inner;           // для Inner доступна приватная часть Outer
  class Inner                   // вложенный класс
  { friend class Outer;         // для Outer доступна приватная часть Inner
    //...
  };
  //...
};
```

Ни вложенный, ни объемлющий классы не имеют возможности пользоваться методами друг друга непосредственно — как и для обычных невложенных классов необходимо объявить объект (или указатель), для которого вызвать нужный метод. Например, объект объемлющего класса может передаваться методу вложенного класса как параметр:

```
void Outer::Inner::MethodInner(const Outer &t)
{ //...
  memInner = t.MethodOuter(); // вызов метода объемлющего класса
  //...
}
```

Здесь метод `MethodInner()` вложенного класса получает ссылку на объект объемлющего класса и обычным способом вызывает метод `MethodOuter()` для инициализации своего поля `memInner`.

¹ Обычно элемент списка называют узлом (*node*).

Внутри методов вложенного класса указатель `this` является указателем на текущий объект вложенного класса.

Если вложенный класс объявлен как `public`, то его можно использовать в качестве типа во всей программе. Но поскольку класс вложенный, то его имя нужно писать с префиксом — именем объемлющего класса, например:

```
Outer::Inner *pointer;
```

Если вложенный класс объявлен в приватной части объемлющего класса, то он доступен только членам объемлющего класса и его друзьям. В этом случае компоненты вложенного класса обычно делают открытыми — тогда нет нужды объявлять объемлющий класс другом, например:

```
class Outer
{ //...
    friend class Inner;           // для Inner доступна приватная часть Outer
    structure Inner              // все элементы доступны в Outer
    { //...
    };
    //...
};
```

Естественно, имя вложенного класса должно быть уникально в объемлющем классе, но может совпадать с другими именами вне класса, например:

```
class A { /* ... */ };           // внешний класс
class B
{ //...
    class A { /* ... */ };       // вложенный класс
    //...
};
```

В этом определении внешний класс `A` не конфликтует с вложенным классом `A`. Методы вложенного класса можно реализовать непосредственно внутри класса. Если же методы вложенного класса определяются вне класса, то определение необходимо писать вне самого внешнего из объемлющих классов — в глобальной области видимости. Естественно, имя метода в таком случае должно иметь два префикса, образованные из имен вложенного и объемлющего классов. Вообще, количество префиксов должно быть равно уровню вложенности классов: первый префикс является именем самого объемлющего класса, второй — именем вложенного класса первого уровня, третий — именем класса, вложенного во вложенный класс, и т. д. Например, конструктор без аргументов класса `Inner` может быть определен таким образом:

```
Outer::Inner::Inner()
{ //...
}
```

Первое имя `Inner` — это имя класса, второе — имя самого конструктора.

В глобальной области видимости вне объемлющего класса можно определить и сам вложенный класс. Подобное определение выглядит отчасти парадоксально,

но C++ разрешает это делать — если в объемлющем классе задать объявление класса. Например:

```
class A
{
    //...
    class B;                // объявление вложенного класса
    //...
};
class A::B                  // внешнее определение вложенного класса
{
    //...
};
```

Доступность определенного таким образом класса зависит от того, в какой части объемлющего класса задано объявление: если объявление приватное, то и определение является приватным в объемлющем классе.

Зачем определять вложенный класс внешним образом? Это позволяет написать определение в отдельном файле¹ и тем самым обеспечивает повышенный уровень инкапсуляции.

Вернемся к разработке дека.

Итератор для последовательного контейнера

Для дека нам потребуются конструкторы и деструктор. Минимально необходимое множество операций включает в себя операции добавления элемента в начало и конец дека, операции удаления первого и последнего элементов. Еще необходимы функция проверки, есть ли в контейнере элементы, и функция, выдающая количество элементов в контейнере. Обычно в состав методов включают и методы получения значений первого и последнего элементов контейнера. Рассмотрим более подробно операцию доступа к элементам контейнера.

Так как внутренняя структура элемента скрыта, поскольку определена в приватной части класса-дека, программа-клиент не сможет работать с элементами контейнера посредством указателей. Для перебора элементов контейнер должен обеспечить последовательный доступ к ним по-другому. Это можно сделать одним из двух способов:

- реализовать методы доступа непосредственно как методы контейнера;
- инкапсулировать все операции доступа в отдельный класс-итератор.

Второе решение предпочтительней хотя бы потому, что позволяет отделить *интерфейс доступа* от интерфейса контейнера. Это позволит в дальнейшем иметь один и тот же *универсальный интерфейс доступа* для разных типов контейнеров. Таким образом, итераторы играют очень важную роль при инкапсуляции информации контейнеров.

ВНИМАНИЕ

Именно таким образом реализованы итераторы в стандартной библиотеке шаблонов (STL). Это — одно из решений, оказавших существенное влияние на ее состав и структуру.

¹ О раздельной трансляции см. главу 13.

Класс-итератор можно реализовать как отдельный независимый класс. Но так как итератор должен иметь доступ к внутренней структуре элемента контейнера, он должен с этим классом «дружить». Очевидно, что итератор очень тесно связан с внутренней организацией контейнера, поэтому лучше его реализовать в качестве вложенного в класс-контейнер класса (см. п. 9.7 в [1]), как и класс-узел. Поскольку программе-клиенту потребуется создавать объекты-итераторы, этот класс должен быть определен в открытой части класса контейнера.

Назовем вложенный класс-итератор именем `iterator`. При создании итератора программа-клиент обязана будет указать префикс — имя объемлющего класса, например:

```
TDeque::iterator it;
```

Значение итератора представляет собой позицию в контейнере. Набор операций с итераторами фактически уже описан нами (см. ранее раздел «Доступ к элементам контейнера»), однако полезно еще раз на этом остановиться. Итак, класс-итератор должен обеспечивать следующий минимальный набор операций:

- получение элемента в текущей позиции итератора (*);
- присваивание итератора (=);
- проверка совпадения позиций, представленных двумя итераторами (== и !=);
- перемещение итератора к следующему элементу контейнера (++).

Итератор с таким набором операций в стандартной библиотеке называется *прямым*. Если мы добавим операцию перемещения к предыдущей позиции (декремент --), то получим итератор, который в стандартной библиотеке называется *двунаправленным*. Отметим, что набор операций двунаправленного итератора соответствует множеству операций с указателями при переборе элементов массива. Это позволяет одинаковым образом обращаться и с массивами, и с контейнерами. Однако надо отметить, что встроенные указатели, по выражению Д. Эджера [25], являются «глупыми», тогда как итератор мы можем сделать настолько «умным», насколько пожелаем.

Для того чтобы начать перебирать элементы контейнера, итератору надо присвоить первоначальное значение, соответствующее первому элементу контейнера. Обычно для этого в контейнер включают метод `begin()`, который в качестве результата возвращает итератор, установленный в начало последовательности элементов контейнера.

Метод `end()` возвращает итератор, установленный в конец последовательности элементов контейнера. Что считать концом последовательности, составляет важный вопрос реализации. По примеру STL (STL — прекрасный пример для подражания!) будем считать, что концом последовательности является позиция *за последним элементом* последовательности. Таким образом, пара методов, `begin()`, `end()`, определяет *полуоткрытый интервал*, который содержит первый элемент, но выходит за пределы последнего элемента (рис. 6.1). Это в точности соответствует ситуации, описанной нами при реализации конструктора класса `TAggay` (см. листинг 5.4).



Рис. 6.1. Методы begin() и end()

Полуоткрытый интервал обладает двумя достоинствами:

- ❑ не нужно специально обрабатывать пустой интервал, так как в пустом интервале значения `begin()` и `end()` равны;
- ❑ упрощается проверка завершения перебора элементов контейнера — цикл продолжается до тех пор, пока итератор не достигнет позиции `end()`.

Для реализации полуоткрытого интервала в контейнер обычно добавляют «пустой» фиктивный элемент, не содержащий данных (рис. 6.2).

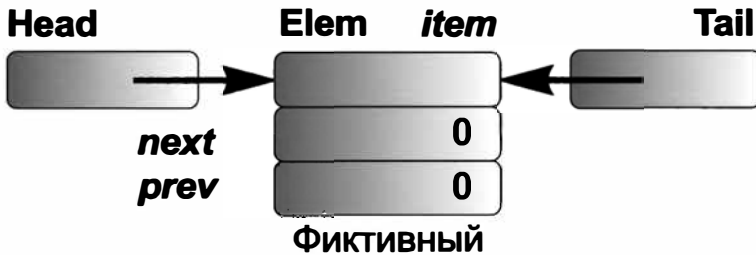


Рис. 6.2. Пустой контейнер с фиктивным «запредельным» элементом

Реализация контейнера-дека

Обратимся теперь непосредственно к реализации. Сначала покажем составляющие класса `TDeque`, а затем — всю его структуру. В листинге 6.1 представлена приватная часть класса.

Листинг 6.1. Приватная часть класса `TDeque`

```
private:
    class Elem // элемент дека
    {
        friend class TDeque;
        friend class iterator;
        Elem(const double &a):item(a){ }
        Elem(){}
        ~Elem(){} // объявлять необязательно
        double item; // информационная часть элемента
        Elem *next; // следующий элемент
        Elem *prev; // предыдущий элемент
    };
    // запрещаем копировать и присваивать дека
    TDeque& operator=(const TDeque &);
    TDeque(const TDeque &);
    long count; // количество элементов
    Elem *Head; // Начало дека
```

продолжение ➤

Листинг 6.1 (продолжение)

```

Elem *Tail;                // указатель на запредельный элемент
// для итератора
iterator head;
iterator tail;

```

Вся внутренность класса Elem закрыта, поэтому классы TDeque и iterator объявлены друзьями, чтобы иметь доступ к конструкторам и указателю. Можно поступить и по-другому — просто сделать все члены класса Elem открытыми:

```

struct Elem
{
    Elem(const double &a):item(a){ }
    Elem(){}
    ~Elem(){}                // объявлять необязательно
    double item;              // информационная часть элемента
    Elem *next;               // следующий элемент
    Elem *prev;               // предыдущий элемент
};

```

В этом случае друзей объявлять не требуется — классы TDeque и iterator и так имеют доступ ко всем элементам класса Elem.

Чтобы не отвлекаться от главной задачи — изучения доступа посредством итератора, — мы объявили конструктор копирования и операцию присваивания закрытыми. Да и нет особой необходимости (пока) присваивать деки. Наличие объявления приводит к тому, что компилятор не будет создавать эти функции по умолчанию. Таким образом, мы запретили создавать копии контейнера-деки и присваивать один дек другому. Следствием является также и то, что контейнер нельзя передавать по значению в качестве параметра и возвращать в качестве результата.

Далее объявлены поля класса TDeque: счетчик элементов контейнера, реальные указатели на начало и конец списка. Счетчик увеличивается при каждом добавлении элемента и уменьшается при каждом удалении элемента. Поля-указатели *никогда не равны 0*, так как даже в пустом контейнере присутствует запредельный фиктивный элемент (см. рис. 6.2).

А вот программе-клиенту указатели недоступны — она работает с итераторами. Следовательно, нужны аналогичные поля для класса-итератора, которые этим классом и инициализируются. Сам класс-итератор (листинг 6.2) определен в открытой части класса TDeque.

Листинг 6.2. Класс iterator

```

class iterator
{
    friend class TDeque;
    iterator(Elem *el):the_elem(el){}
public:
    // конструкторы
    iterator():the_elem(0){}
    iterator(const iterator &it):the_elem(it.the_elem){}
    // присваивание итераторов - генерируется по умолчанию
    // сравнение итераторов
    bool operator==(const iterator &it) const

```

```

    { return (the_elem == it.the_elem); }
    bool operator!=(const iterator &it) const
    { return !(it == *this); }
    // продвижение к следующему элементу - только префиксная форма
    iterator& operator++()
    {
        if ((the_elem!=0)&&(the_elem->next!=0))
            the_elem = the_elem->next;
        return *this;
    }
    // продвижение к предыдущему элементу - только префиксная форма
    iterator& operator--()
    {
        if ((the_elem!=0)&&(the_elem->prev!=0))
            the_elem = the_elem->prev;
        return *this;
    }
    // получить ссылку на информационную часть
    // работает справа и слева от знака присваивания
    double& operator*() const
    {
        if (the_elem != 0) return the_elem->item;
        else { cout << "Null iterator!" << endl; abort(); }
    }
private:
    Elem *the_elem;           // вот это итератор скрывает!
};

```

Единственный закрытый элемент класса — указатель на элемент контейнера-дека. Именно этот указатель должен скрывать итератор, предоставляя пользователю более надежный способ доступа. Однако для первоначальной установки указателя нашему деку требуется конструктор с указателем. Поэтому реализован приватный конструктор, и класс-дек сделан другом класса-итератора.

Хотя мы не обрабатываем ошибки¹, тем не менее операции продвижения не приводят к непредсказуемому поведению программы — при некорректном указателе продвижение просто не выполняется и возвращается текущий итератор.

В операции разыменования `operator *` указатель на элемент проверяется на нуль. Эта проверка необходима, так как в классе-итераторе есть конструктор по умолчанию, обнуляющий этот указатель, — программа-клиент ведь может объявить итератор, но не инициализировать его. Операция возвращает ссылку, чтобы выражение `*iterator` можно было использовать слева от знака присваивания для изменения значения элемента контейнера.

Все методы удобнее реализовать именно внутри класса `iterator`, так как при реализации вне его (и вне класса `TDeque`) придется писать слишком длинные префиксы.

Разработанный с учетом всех этих соображений класс `TDeque` представлен в листинге 6.3.

В этом классе надо обратить внимание на несколько моментов. Во-первых, так как приватная часть прописана в конце, требуется опережающее объявление класса `Elem`, иначе класс `iterator` не сможет объявить свой указатель на элемент.

¹ Мы реализуем обработку ошибок после изучения механизма исключений.

Так как объявлено только имя класса, то создавать объекты не разрешается (компилятору на момент объявления не известен размер объекта), но определить указатель на такой объект вполне можно.

Листинг 6.3. TDeque — контейнер-дек

```
class TDeque
{   class Elem;                               // опережающее объявление
public:
    class iterator
    { //... см. листинг 6.2
    };
public:
    // создаем пустой дек - с фиктивным запредельным элементом
    TDeque():Head(new Elem()), Tail(Head), count(0)
    { Tail->next=Tail->prev=0;
      head = iterator(Head);           // инициализация для итератора
      tail = iterator(Tail);           // приватным конструктором
    }
    // создаем дек с единственным элементом
    TDeque(const double& a):Head(new Elem()), Tail(Head), count(0)
    { Tail->next=Tail->prev=0;
      head = iterator(Head);
      tail = iterator(Tail);
      push_front(a);
    }
    ~TDeque();
    bool isEmpty() const                    // есть ли элементы в deque?
    { bool t = (Head == Tail); return t; }
    long size() const { return count; }    // количество элементов в deque
    // методы доступа
    iterator begin() { return head; }      // первый элемент
    iterator end()   { return tail; }      // запредельный элемент
    // методы вставки и удаления на концах дека
    void push_front(const double &a);
    void push_back(const double &a);
    void pop_front();
    void pop_back();
private:
    // ... см. листинг 3.19
};
```

Во-вторых, мы непосредственно в классе объявили два конструктора с практически идентичными телами. Почему бы во втором конструкторе не использовать первый, как мы обычно поступали, например:

```
TDeque(const double& a)
{ TDeque t;
  t.push_front(a);
  *this = a;                               // ошибка - операция присваивания недоступна
}
```

Однако мы не можем этого сделать, так как закрыли операцию присваивания. Контейнер, содержащий один элемент, показан на рис. 6.3.

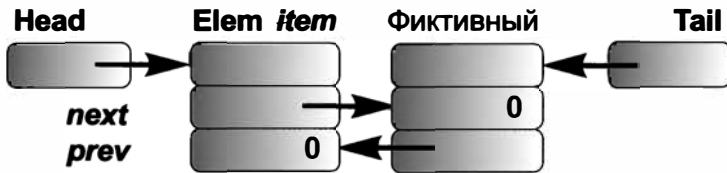


Рис. 6.3. Контейнер с единственным элементом

Рассмотрим теперь реализацию операций вставки и удаления в начале дека (листинг 6.4).

Листинг 6.4. Операции вставки и удаления в начале дека

```

// добавить элемент
void TDeque::push_front(const double &a)
{
    Elem *p = new Elem(a);           // образовали новый элемент
    p->next = Head; p->prev = 0;       // "привязали"
    Head->prev = p; Head = p;         // первым в деке
    head = iterator(Head);            // корректируем итератор
    ++count;                          // добавили элемент
}

// удалить элемент
void TDeque::pop_front()
{
    if (!isEmpty())                  // если есть элементы
    {
        Elem *p = Head;             // сохраняем указатель для удаления
        Head = Head->next;            // "отцепляем"
        Head->prev = 0;               // элемент
        head = iterator(Head);        // корректируем итератор
        --count;                     // уменьшаем количество
        delete p;                    // возвращаем память
    }
}

```

Как видите, реализация не очень сложная. Единственное, на что следует обратить самое пристальное внимание, — это порядок присваивания указателей. Кроме того, операция удаления обязана проверить наличие элементов в деке: если дек пуст, то операция ничего не делает.

В листинге 6.5 показана реализация операций вставки и удаления элементов в конце дека. И здесь тоже необходимо самым внимательнейшим образом отследить порядок работы с указателями. Кроме того, обратите внимание на то, что в данном случае не требуется корректировать итератор, так как указатель Tail фиксирован и никогда не изменяется — он всегда указывает на запредельный элемент.

Листинг 6.5. Вставка и удаление элементов в конце дека

```

// вставка элемента
void TDeque::push_back(const double &a)
{
    if (isEmpty()) push_front(a);    // если вставляем первый раз
    else
    {
        Elem *p = new Elem(a);       // создали элемент

```

продолжение ➤

Листинг 6.5 (продолжение)

```

        p->next = Tail;           // "привязываем"
        p->prev = Tail->prev;     // перед
        Tail->prev->next = p;     // фиктивным
        Tail->prev = p;          // элементом
    }
    ++count;                      // добавили элемент
};
// удаление элемента
void TDeque::pop_back()
{
    if (!isEmpty())              // если есть элементы
    {
        Elem *p = Tail->prev;    // сохраняем для удаления
        if (p == 0) pop_front(); // если элемент единственный
        else
        {
            p->prev->next = Tail; // отцепляем
            Tail->prev = p->prev; // элемент
            delete p;            // возвращаем память
        }
        --count;                // уменьшаем количество
    }
}

```

И наконец, рассмотрим деструктор, текст которого представлен в листинге 6.6.

Листинг 6.6. Деструктор дека

```

TDeque::~TDeque()
{
    Elem *delete_Elem = Head;    // удаляемый элемент
    for(Elem *p = Head; p!=Tail;) // пока не уперлись в запредельный
    {
        p = p->next;              // подготовили следующий
        delete delete_Elem; --count; // удалили элемент
        delete_Elem = p;          // подготовили для удаления
    }
    delete delete_Elem;          // удалили запредельный
}

```

В цикле удаляются все «значащие» элементы контейнера. После выхода из цикла у нас остается только запредельный элемент, который мы удаляем отдельным оператором delete.

Проверить работу операций вставки и удаления, а также доступа к элементам по итератору можно, выполнив программу, представленную в листинге 6.7.

Листинг 6.7. Тестовая программа для проверки работы контейнера-дека

```

int main()
{
    TDeque L(8.0);                // создали дек из одного элемента
    // добавляем 5 элементов в начало – стало 6
    L.push_front(1); L.push_front(2); L.push_front(3);
    L.push_front(4); L.push_front(5);
    cout << L.size() << endl;    // выводим количество элементов
    TDeque::iterator i;           // объявили итератор
    // вывод элементов контейнера
    for(i = L.begin(); i != L.end(); ++i) // изменение итератора
        cout << *i << ' ';      // вывод значения
}

```

```

cout << endl;
// добавляем пять элементов в конец дека - стало 11
L.push_back(1); L.push_back(2); L.push_back(3);
L.push_back(4); L.push_back(5);
// вывод элементов контейнера
for(i = L.begin(); i != L.end(); ++i)
    cout << *i << ' '; cout << endl;
cout << L.size() << endl; // выводим количество элементов
// Обратный перебор
i = L.end(); // на запредельный элемент
while(i != L.begin()) // пока не дошли до первого
{ --i; // сначала переходим на реальный элемент
    cout << *i << ' '; // выводим значение
}
cout << endl;
// удалили 4 элемента в начале дека - осталось 7
L.pop_front(); L.pop_front(); L.pop_front(); L.pop_front();
// вывод элементов контейнера
for(i = L.begin(); i != L.end(); ++i)
    cout << *i << ' '; cout << endl;
cout << L.size() << endl; // выводим количество элементов
// удалили 3 элемента в конце дека - осталось 4
L.pop_back(); L.pop_back(); L.pop_back();
// вывод элементов контейнера
for(i = L.begin(); i != L.end(); ++i)
    cout << *i << ' '; cout << endl;
cout << L.size() << endl; // выводим количество элементов
return 0;
} // работает деструктор, удаляя 4 элемента и запредельный элемент

```

Единственное, на что следует обратить внимание в этой программе, — это обратный перебор элементов дека. Итератор устанавливается на запредельный элемент, поэтому перед выводом значения он сначала должен быть переведен на последний реальный элемент. Это делает операция декремента.

Результат программы соответствует комментариям:

```

6 // количество элементов в деке
5 4 3 2 1 8 // элементы дека
5 4 3 2 1 8 1 2 3 4 5 // прямой перебор элементов дека
11 // количество элементов
5 4 3 2 1 8 1 2 3 4 5 // обратный перебор элементов дека
1 8 1 2 3 4 5 // удалили 4 элемента в начале
7 // количество элементов после удаления
1 8 1 2 // удалили 3 элемента в конце
4 // количество элементов после удаления

```

Стек

Реализуем еще один простой контейнер — стек. Мы могли бы использовать в качестве стека и наш класс-дек, однако он делает гораздо больше, чем нам надо, например, перебирать элементы стека, как правило, нет необходимости, поэтому

класс-итератор нам не нужен. Мы реализуем более простой вариант, обеспечив только необходимую функциональность:

- добавление элемента в стек — `push()`;
- удаление элемента из стека — `pop()`;
- получение значения элемента с вершины стека — `top()`;
- проверка, не пустой ли стек — `empty()`.

Для реализации такой дисциплины доступа к элементам нам достаточно *одно-связного* списка. Структура элемента односвязного списка (листинг 6.8) практически не отличается от структуры элемента двусвязного, только указатель для связи — единственный.

Очевидно, что стек является универсальной структурой, работа которой не зависит от типа элементов. Однако написать стек, который можно было бы использовать с данными произвольного типа, не так просто — тип элемента определяется во время компиляции и не может меняться динамически. Единственный универсальный тип в C++ — это нетипизированный указатель `void *`. Поэтому типом элементов должен быть указатель `void *`.

Листинг 6.8. Элемент односвязного списка

```
struct Elem
{ void *data;
  Elem *next;
  Elem (void *d, Elem *p)
    :data(d), next(p) { }
};
```

Так как в этом случае только владелец данных (программа-клиент, использующая стек) знает реальный тип элементов стека, то пусть она и «опорожняет» стек с помощью операции `pop()`. Таким образом, деструктор мы писать не будем¹.

Конструктор инициализации нам тоже ни к чему, так как изначально стек не содержит ни одного элемента. Присваивать и копировать стеки обычно не требуется. Таким образом, нам хватит только одного конструктора без аргументов, функцией которого является обнуление указателя на вершину стека. Нулевое значение указателя служит признаком отсутствия элементов в стеке. Реализация нашего стека с учетом всех этих соображений представлена в листинге 6.9.

Листинг 6.9. Реализация простого стека

```
class TStack
{ struct Elem                                // узел списка
  { void *data;                             // информационная часть
    Elem *next;                             // указательная часть
    Elem (void *d, Elem *p)                 // конструктор узла
      :data(d), next(p)
    { }
```

¹ Однако система создаст «пустой» деструктор по умолчанию.

```

};
Elem * Head; // вершина стека
TStack(const TStack &); // закрыли копирование
TStack& operator=(const TStack &); // закрыли присваивание
public:
    TStack(): Head(0) {} // пустой стек
    void push(void *d) // положить элемент в стек
    { Head = new Elem(d, Head); }
    void *top() const // получить элемент с вершины стека
    { return empty()? 0: Head->data; }
    void *pop() // удалить элемент из стека
    { if (empty()) return 0; // если стек пустой – ничего не делать
      void *top = Head->data; // сохранили для возврата
      Elem *oldHead = Head; // запомнили указатель
      Head = Head->next; // передвинули вершину
      delete oldHead; // возвратили память
      return top; // возвратили элемент
    }
    bool empty() const // стек пустой. если указатель = 0
    { return Head==0; }
};

```

Деструктор нам действительно не понадобился: удаление элемента стека выполняет метод `pop()`, который возвращает указатель на информационную часть, чтобы программа-клиент могла вернуть эту память системе.

ПРИМЕЧАНИЕ

Вообще-то при такой реализации `pop()` в некоторых редких случаях (при возникновении исключения – см. главу 7) возможна потеря данных. Поэтому в стандартной библиотеке подобные методы в классах-контейнерах не возвращают результата, а только удаляют элемент из контейнера.

Функции `pop()` и `top()` проверяют «пустоту» стека; признаком пустого стека является нулевой указатель `Head`.

При использовании стека нужно выполнять преобразование указателей:

```

TStack t;
t.push(new double(1)); // кладем в стек числа
t.push(new double(2));
t.push(new double(3));
while (!t.empty()) // пока стек не пустой
{ cout << *(double *)t.top() << endl; // выводим число с вершины
  double *p = (double *)t.pop(); // удаляем элемент из стека
  delete p; // возвращаем память
}

```

Этот фрагмент программы выведет на экран следующее:

```

3
2
1

```


Проблема универсальности

Пока все хорошо и не видно никаких сложностей. Более того, наш стек «слишком» универсален — он может содержать элементы разного типа!

Однако не все так прекрасно, как представляется. Во-первых, на указатели расходуется дополнительная память. Во-вторых, класс-стек понятия не имеет о реальном типе элементов в контейнере, поэтому не может выдать значение элемента, а вынужден выдавать указатель `void *`. Дальнейшее разыменование должна делать программа-клиент, обеспечивая еще и преобразование в нужный тип. При этом, конечно, приходится писать не самые красивые выражения.

Но гораздо более важным является то, что программа-клиент теперь обязана следить за памятью, ведь в контейнер помещается только указатель на некоторые данные. Этими данными *владеет* программа-клиент, и только она «знает», как с ними оперировать. И тут, как обычно это бывает при использовании указателей, может возникнуть множество проблем. Например, в стек вполне можно поместить адрес локальной переменной. Если стек имеет большее время жизни, чем эта переменная, то мы получим висячую ссылку. При использовании такого стека очень просто получить и потерянные ссылки. Если программа-клиент «забудет» освободить память, то при уничтожении стека-переменной (например, при выходе из блока) мы получим утечку памяти. Одним из решений проблем с указателями могут стать *интеллектуальные указатели*, которые в этой книге не рассматриваются¹.

Дек (см. листинги 6.1, 6.2, 6.3) реализован для элементов типа `double`. Однако по сути дек тоже является универсальным контейнером, работа которого не зависит от типа элементов. К сожалению, использование указателей типа `void *` приведет к тем же проблемам, что и со стеком. Более того, поскольку в классе `TDeque` реализован деструктор, существует очень большая опасность получения утечек памяти. Поэтому применение нетипизированных указателей для универсализации дека не является самым лучшим решением.

Таким образом, мы видим, что написать универсальный контейнер, работа которого не зависит от типа элементов, не так-то просто. В C++ есть несколько вариантов решения этой проблемы. Одним из вариантов является наследование (см. главы 8 и 9), другим — *шаблоны*, включенные в C++ специально для решения такого рода проблем (см. главу 11).

Однако есть один очень простой, но очень полезный прием, который повышает степень универсальности динамических структур данных и позволяет повторно использовать уже разработанные классы с другим типом данных. Если еще раз взглянуть на класс `TDeque` (см. листинг 6.3), то ни в одном из компонентов класса мы не обнаружим спецификатор типа `double`. Поэтому явную зависимость класса `TDeque` от элементов типа `double` можно уменьшить, использовав оператор `typedef`:

```
typedef double value_type;
```

¹ Описание интеллектуальных указателей можно найти в [20, 25, 26, 28].

После этого компоненты класса, явно зависящие от `double`, переписываются с типом `value_type`, например:

```
// в классе iterator
value_type& operator*() const
{   if (the_elem != 0) return the_elem->item;
    else { cout << "Null iterator!" << endl; abort(); }
}
// в классе TDeque - добавить элемент
void TDeque::push_front(const value_type &a)
{   Elem *p = new Elem(a);           // образовали новый элемент
    p->next = Head; p->prev = 0;       // "привязали"
    Head->prev = p; Head = p;         // первым в deque
    head = iterator(Head);            // корректируем итератор
    ++count;                           // добавили элемент
}
```

Таким образом, изменение типа *инкапсулируется* в операторе `typedef`: если нам потребуется дек с другим типом элементов, нужно изменить *единственную* строку в классе, дублируя остальное определение без изменений. Это практически гарантирует нам отсутствие ошибок в новом deque.

Аналогично можно обобщить и класс `TArray`. Давайте оставим в классе только те операции, которые не учитывают специфику числового типа элементов. Тогда класс `TArray` фактически будет представлять собой определение динамического массива (листинг 6.10).

Листинг 6.10. Динамический массив

```
class TArray
{   public:
    // определение типов
    typedef std::size_t    size_type;
    typedef std::size_t    index_type;
    typedef double         value_type;
    typedef double &       reference;
    typedef double *       iterator;
    typedef double *       pointer;

    TArray(size_type size, value_type k=0.0);
    TArray(const TArray &a);
    TArray(const TArray &a, index_type begin, size_type k);
    TArray(const iterator begin, const iterator end);
    ~TArray();

    reference operator[](index_type index);
    const reference operator[](index_type index) const;
    TArray& operator=(const TArray &a);
    TArray& assign(const TArray &a, index_type l, index_type r);
    TArray& assign(const iterator begin, const iterator end);
    size_type size()const { return size_array; };
    iterator find(const value_type &a);
    friend ostream& operator <<(ostream& to, const TArray &a);
    friend istream& operator >>(istream& to, TArray &a);
    private:
        size_type size_array;
        pointer data;
};
```

И в этом классе нет ни одного метода, использующего числовую специфику элементов. Поэтому, заменяя объявление типов в операторах `typedef`, мы получаем возможность дублировать определение класса `TArray` для любого нужного нам типа элементов.

Резюме

Поскольку встроенных массивов явно не хватает для обработки групп однотипных данных, программистское сообщество предложило более общую конструкцию объединения однородных данных в группу — *контейнер*. Все контейнеры обладают некоторыми свойствами. Одним из важнейших характеристик контейнера является способ доступа к элементам: последовательный, прямой или ассоциативный.

Прямой и ассоциативный варианты доступа обычно реализуются посредством перегрузки операции индексирования `operator[]`, а последовательный — реализацией класса-итератора. Такая реализация имеет еще и то преимущество, что отделяет интерфейс доступа от интерфейса контейнера. Так как итератор тесно связан с внутренней структурой контейнера, его обычно создают как вложенный класс.

Для контейнеров реализуется множество операций, однако одна из наиболее часто используемых — операция объединения контейнеров, реализуемая самыми разнообразными способами.

Динамические структуры данных, как правило, являются универсальными структурами, работа которых не зависит от типа элемента. Таковыми являются стек, очередь и дек, которые различаются только дисциплиной добавления и удаления элементов. Как правило, такие структуры реализуются с помощью связанных списков. Однако написать универсальный класс без использования механизма наследования или шаблонов достаточно сложно — как правило, приходится задействовать нетипизированные указатели, которые потенциально опасны.

Контрольные вопросы

1. Дайте определение контейнера.
2. Какие виды встроенных контейнеров в C++ вы знаете?
3. Какие способы доступа к элементам контейнера вам известны?
4. Чем отличается прямой доступ от ассоциативного?
5. Перечислите операции, которые обычно реализуются при последовательном доступе к элементам контейнера.
6. Дайте определение итератора.
7. Можно ли реализовать последовательный доступ без итератора? В чем преимущества реализации последовательного доступа с помощью итератора?

8. Что играет роль итератора для массивов C++?
9. Дайте определение вложенного класса.
10. Можно ли класс-итератор реализовать как внешний класс? А как вложенный? В чем различия этих способов реализации?
11. Может ли объемлющий класс иметь неограниченный доступ к элементам вложенного класса? А вложенный класс — к элементам объемлющего?
12. Ограничена ли глубина вложенности классов?
13. Можно ли определить вложенный класс внешним образом? Зачем это может понадобиться?
14. Каким образом вложенный класс может использовать методы объемлющего класса? А объемлющий — методы вложенного?
15. Что такое «запредельный» элемент, какую роль он играет в контейнерах?
16. Дайте определение стека, очереди и дека.
17. Объясните назначение методов `begin()` и `end()`.
18. Обязательно ли при реализации контейнеров программировать операцию присваивания? А конструктор копирования?
19. Приведите минимальный интерфейс, который необходимо реализовать при разработке класса-стека.
20. Объясните, по каким причинам трудно написать универсальный контейнер, элементы которого могут иметь произвольный тип.

Упражнения

Во всех заданиях обязательно должны быть реализованы инициализирующий конструктор, конструктор копирования, деструктор, ввод-вывод. Подходящие операции реализуются как методы класса, а остальные — как внешние дружественные функции.

1. Модифицировать класс `TDeque` (см. листинги 6.1–6.3), добавив операцию простого сцепления. Перегрузить для этого операцию `operator+`.
2. Добавить в класс `TDeque` операции сортировки и слияния отсортированных списков.
3. Реализовать итератор класса `TDeque` как внешний класс.
4. Используя в качестве основы класс `TDeque`, реализовать операции объединения и пересечения множеств. Если потребуется, реализовать дополнительные методы.
5. Для изучения иностранного языка имеются специальные карточки, каждая из которых представляет собой структуру `WordCard`, содержащую иностранное слово и его перевод. Используя в качестве образца класс `TDeque`, реализовать класс словаря иностранных слов `Dictionary`, который содержит список карточек иностранных слов. Карточки добавляются в словарь и удаляются из

него. В словаре не должно быть дублирования. Реализовать поиск слова как отдельный метод. Реализовать операцию объединения словарей. При объединении новый словарь должен содержать без повторений все слова, содержащиеся в обоих словарях-операндах. Реализовать операцию ассоциативного доступа; ключом является иностранное слово.

6. Реализовать очередь нуждающихся в улучшении жилищных условий. Элементом очереди является структура с полями фамилии, количества членов семьи и даты постановки в очередь. Для реализации поля даты использовать упрощенную версию класса `Date` (см. упражнение 4 в главе 2). Реализовать операцию поиска элемента по фамилии. Реализовать операцию объединения двух очередей (если очередь есть в обеих — включать только один раз).
7. Реализовать стек символов в виде односвязного списка. Используя этот стек, написать функцию реверсирования массива символов.
8. Реализовать класс `ListPerson` (см. упражнение 10 в главе 4) с использованием списка, аналогичного `TDeque`.
9. Реализовать класс `Money` (см. упражнение 4 в главе 4), используя список в качестве динамического контейнера.
10. Реализовать класс `Polinom` (см. упражнение 5 в главе 4), разработав вложенный класс односвязного списка.
11. Реализовать стек символов в виде двусвязного списка. Используя этот стек, написать функцию, проверяющую правильность задания скобок в строке символов.

Глава 7

Исключения

При выполнении любой программы бывают аварийные ситуации, вызванные самыми разнообразными причинами. Если мы делаем программу «для себя», то встроить в нее механизм обработки некоторых ошибок с выдачей соответствующих сообщений труда не представляет. Но в профессиональном программировании программы пишутся не для себя, а для других, поэтому такой метод обработки ошибок неприемлем.

При реализации класса строк (см. листинг 4.2) мы столкнулись как раз с такой ситуацией: одна из операций индексирования при неправильном параметре возвращала нулевой байт. Такое решение допустимо только для строк, так как в любой кодировке символ с нулевым кодом, как правило, входит в подмножество специальных символов. Для числовых массивов, очевидно, это решение не подходит — отсутствует специальное значение. Поэтому лучше просто не выполнять операцию, а сообщить о неверном параметре программе-клиенту.

Аналогичные проблемы возникают при реализации конструкторов. В классе `TString` (см. листинги 4.3–4.7) любой конструктор создавал пустую строку, если получал неправильные параметры. Для строк это решение можно было реализовать, так как в классе `TString` определено поле-массив фиксированного размера. Однако создать «умный» массив, если неправильно задана длина, нельзя, поскольку память выделяется динамически. Поэтому конструктор «умного» массива (см. листинги 5.2–5.4) завершался аварийно. В профессиональном программировании такое поведение программы совершенно неприемлемо — об аварии требуется сообщить программе-клиенту.

В обычной функции-методе для этих целей можно использовать либо дополнительный параметр, либо возвращаемое значение. Однако перегруженная операция не может иметь лишних параметров, поэтому должна в таких случаях возвращать некоторое «аварийное» значение. У конструктора же, как и у перегружаемых операций, в отличие от обычных функций, нет и такой возможности.

Таким образом, необходим языковой механизм, с помощью которого методы и конструкторы класса могли бы сообщать программе-клиенту об аварийной ситуации. В стандарт C++ для этих целей включен *механизм обработки исключений* (см. п. 15 в [1]).

Принципы обработки исключений

В C++ исключение — это объект. Хотя обычно говорят об исключительной ситуации в программе, такая точка зрения мало что дает, так как с ситуацией сделать ничего нельзя. Поэтому в C++ при возникновении исключительной ситуации программа должна *генерировать*¹ *объект-исключение*. Собственно, сама генерация объекта-исключения и создает исключительную ситуацию. Такой подход очень удобен, так как с объектами, в отличие от ситуаций, мы можем много чего делать. Например, объект-исключение можно объявить как обычную переменную, передать его как параметр любым из возможных способов или возвратить в качестве результата. Можно объявлять массивы исключений или включать объекты-исключения в качестве полей в другие классы. В дальнейшем мы будем использовать термин «исключение», понимая под этим объект-исключение.

Общая схема обработки исключений такова: в одной части программы, где обнаружена аварийная ситуация, исключение *порождается*; другая часть программы *контролирует* возникновение исключения, *перехватывает* и *обрабатывает* его. В C++ есть три зарезервированных слова: `try` (контролировать), `catch` (перехватывать), `throw` (порождать), — которые и используются для организации процесса обработки исключений.

Генерация исключений

Генерируется исключение оператором `throw` (см. п. 15.1 в [1]), который имеет следующий синтаксис

```
throw выражение_генерации_исключения;
```

Выражение генерации исключения на практике означает либо константу, либо переменную некоторого типа. Тип объекта-исключения может быть любым, как встроенным, так и определяемым программистом. Например:

```
throw 7;  
throw "Ошибка: деление на ноль!";  
throw Message[i];  
throw M_PI;
```

В первом случае объект-исключение — это целая константа, которая может быть условным номером-кодом ошибки. В общем случае этот код ошибки может вычисляться, например:

```
throw 3*v-i;
```

¹ Обычно говорят, что исключение возбуждается; или выбрасывается. Но так как исключение — это объект, вполне уместно слово «генерировать», или «порождать».

Выражение, конечно, может быть не только целочисленным.

Второй вариант — это символьная константа, которая фактически является сообщением об ошибке. Все сообщения об ошибках могут быть записаны в массиве, например:

```
string Message[326];
```

Тогда в третьем варианте объект-исключение тоже представляет собой строку — сообщение об ошибке. В последнем примере в качестве объекта используется дробная константа — число .

Программист может и сам определить собственный тип объекта-исключения, объявив новый класс, например:

```
class NegativeArgument{};
NegativeArgument exeption;
if (x>0) double t = x/sqrt(x);
else throw exeption;
```

Пустые классы неожиданногодились! Однако объявлять переменную совсем не обязательно, можно обойтись и без этого, например:

```
throw NegativeArgument();
```

Здесь в качестве выражения генерации исключения мы использовали явный вызов конструктора без аргументов, и это наиболее распространенная форма генерации исключения.

Объект-исключение может быть и динамическим, например:

```
throw new NegativeArgument();
```

Однако в этом случае возникает вопрос об уничтожении объекта-исключения и возврате памяти — неизвестно, в каком месте программы это можно сделать. Лучше не создавать себе проблем и не использовать указатели в качестве параметров блоков catch.

Перехват и обработка исключений

Сгенерировать исключение — это только полдела. Как уже отмечалось, исключение надо *перехватить* и *обработать*. Проверка возникновения исключения делается с помощью оператора try, с которым неразрывно связаны одна или несколько блоков обработки исключений — catch. Оператор try объявляет в любом месте программы *контролируемый блок*, который имеет следующий вид:

```
try { /* контролируемый блок */ }
```

Контролируемый блок, помимо функции контроля, обладает функциями обычного блока: все переменные, объявленные внутри него, являются локальными в этом блоке и не видны вне его.

После блока try обязательно прописывается один или несколько блоков catch, которые обычно называют *обработчиками исключений*, или *секциями-ловушками*. Форма записи секции-ловушки следующая:

```
catch (спецификация_параметра_исключения) { /* блок обработки */ }
```


Спецификация параметра исключения может иметь следующие три формы:

```
(тип имя)
(тип)
(...)
```

Тип должен быть одним из допустимых типов исключений: либо встроенным, либо определенным программистом¹. Первый вариант спецификации означает, что объект-исключение передается в блок обработки, чтобы там его как-то использовать, например, для вывода информации в сообщении об ошибке.

ВНИМАНИЕ

При выполнении оператора `throw` создается временный объект-исключение, который и передается в секцию-ловушку (см. п. п. 15.1/3 в [1]).

При этом объект-исключение может передаваться в секцию-ловушку любым способом: по значению, по ссылке или по указателю, например:

```
catch (TException e)           // по значению
catch (TException& e)         // по ссылке
catch (const TException& e)    // по константной ссылке
catch (TException *e)         // по указателю
```

Однако, в отличие от параметров функций, никаких преобразований по умолчанию *не производится*². Это означает, что если в программе написан следующий заголовок обработчика, то попадают в соответствующую ловушку только те исключения, тип которых совпадает с `double`:

```
catch (double e)               // по значению
```

Еще пример:
`throw 1`

Этот оператор генерирует исключение целого типа, поэтому будет обрабатываться секцией-ловушкой с заголовком, в котором прописан целый тип исключения, например, одним из следующих способов:

```
catch (int e)                  // по значению
catch (int)                    // без передачи информации в секцию-ловушку
```

Это легко проверить, выполнив следующую простую тестовую программу:

```
int main()
{
    try { throw 1; }           // генерация исключения
    catch(unsigned int)
    { cout << "unsigned integer" << endl; }
    catch(int)                 // перехватывается здесь
}
```

¹ В качестве параметра секции-ловушки ничто не мешает нам задать, например, указатель на функцию, а затем обычным образом вызвать ее в блоке обработки.

² Это касается только встроенных типов и независимых классов. *Наследование классов* (см. главу 8) вносит некоторую специфику в это сопоставление типов.

```
        { cout << "integer" << endl;
        }
    catch(double)
    { cout << "double" << endl;
    }
    return 0;
}
```

На экране появится слово `integer`, так как константа `1` по умолчанию имеет тип `int`.

Первые две формы из приведенных спецификаций исключения предназначены для обработки конкретного типа исключений. Если же на месте спецификации исключения написано многоточие (как в функциях с переменным числом параметров), то такой обработчик перехватывает *все* исключения.

Работа конструкции `try...catch` напоминает работу оператора `switch`. Секции-ловушки похожи на альтернативы `case`, а блок `catch` с многоточием соответствует альтернативе `default` оператора-переключателя. Если в блоке `try` возникает исключение, то начинается сравнение типа сгенерированного исключения и типов параметров в секциях-ловушках. Выполняется тот блок `catch`, тип параметра которого совпал с типом исключения. Если такого типа не найдено, выполняется блок `catch` с многоточием. Если же такого блока в текущем списке обработчиков не обнаруживается, то ищется другой список обработчиков в вызывающей функции. Этот поиск продолжается вплоть до функции `main()`. Если же и там не обнаружится нужного блока `catch`, то вызывается стандартная функция завершения `terminate()` (см. п. п. 15.5.1 в [1]), которая вызывает функцию `abort()`.

Таким образом, очень важен порядок записи секций-ловушек после контролируемого блока. Если в качестве первого обработчика после блока `try` задан блок `catch` с параметром-многоточием, то такой обработчик будет обрабатывать *все* возникающие исключения — до остальных секций-ловушек дело просто не дойдет. Поэтому усвойте следующее простое правило: блок `catch` с параметром-многоточием всегда должен быть последним. Секция-ловушка с многоточием — это крайняя мера: если уж мы в нее попали, то в программе произошло что-то совсем непредусмотренное. Поэтому в такой секции обычно выводят сообщение о непредвиденном исключении и завершают работу программы.

Выход из секции-ловушки выполняется одним из следующих способов:

- выполняются все операторы обработчика, и происходит неявный переход к оператору, расположенному после конструкции `try...catch`;
- в обработчике выполняется оператор `goto`¹, при этом разрешается переходить на любой оператор вне конструкции `try...catch`, а внутрь контролируемого блока или в другую секцию-ловушку переход запрещен;
- в обработчике выполняется оператор `return`, после чего происходит нормальный выход из функции;

¹ Разрешается также использовать операторы `break` и `continue`.

- в секции-ловушке выполняется оператор `throw`;
- в обработчике генерируется другое исключение.

Рассмотрим подробнее два последних способа выхода. Оператор `throw` без выражения генерации исключения генерирует повторное исключение того же типа. Такая форма оператора допустима только внутри секции-ловушки.

ПРИМЕЧАНИЕ

Выполнение этого оператора вне секции-ловушки приведет к немедленному аварийному завершению программы (см. п. п. 15.1/8 в [1]).

Однако это не приводит к рекурсивному входу в тот же обработчик — ищется другой обработчик выше по иерархии вложенности. Аналогично, при генерации в секции-ловушке исключения другого типа ищется его обработчик выше по иерархии вложенности. Если нужного блока `catch` не обнаруживается, то программа аварийно завершается. Таким образом, мы имеем возможность «распределить» обработку исключения по разным частям программы.

Выход из обработчика по исключению может привести к выходу из функции. Очень важно, что стандарт (см. п. п. 15.2/1 в [1]) в этом случае *гарантирует* вызов деструкторов для уничтожения локальных объектов. Этот процесс уничтожения локальных объектов при выходе по исключению называется «раскруткой» (*unwinding*) стека. Нужно подчеркнуть, что раскрутка стека выполняется только для *локальных* объектов — для динамических объектов, созданных операцией `new`, деструкторы автоматически не вызываются. Маленький пример продемонстрирует эту ситуацию:

```
void f1(void)
{ //...
    MyObject t;
    throw MyException();
}
void f2(void)
{ //...
    MyObject t = new MyObject();
    throw MyException();
}
```

Генерация исключения в функции `f1()` приведет к вызову деструктора для уничтожения локального объекта `t`. При генерации исключения в функции `f2()` деструктор класса `MyObject` *не вызывается*, так как объект создан в динамической памяти. Более того, уничтожается локальный указатель, и это приводит к утечке памяти. Обычно эта проблема решается с помощью *интеллектуальных указателей* [20, 21, 24, 25, 28].

ПРИМЕЧАНИЕ

В библиотеке STL реализован один из видов интеллектуальных указателей — `auto_ptr`.

После выполнения операторов блока `catch` при отсутствии явных операторов перехода или оператора `throw` выполняются операторы, расположенные после всей конструкции `try...catch`. Если во время работы в блоке `try` не обнаружено исключительной ситуации, то все блоки `catch` пропускаются и программа продолжает выполнение с первого после всех блоков `catch` оператора.

Блоки `try...catch` могут быть вложенными, причем как в блок `try`, так и в блок `catch`:

```
try {    // блок, который может инициировать исключения
    try {    //вложенный блок
    }
    catch(...){    }
}
catch (исключение) {    // обработка исключения
    try {    //вложенный блок
    }
    catch(...){    }
}
```

Необходимо отметить, что исключение может быть сгенерировано в одном месте программы, а обработано совершенно в другом.

Спецификация исключений

В главе 5 мы уже столкнулись со *спецификацией исключений* (см. п. п. 15.4 в [1]), когда рассматривали прототипы функций `new()` и `delete()`. Спецификацию исключений можно задать в заголовке любой функции, метода, конструктора или деструктора — до сих пор мы этого не делали. Если в заголовке функции не указана спецификация исключений, считается, что функция может порождать любое исключение. Однако можно специфицировать в заголовке явный список исключений, которые функция может генерировать, например:

```
void f1(void) throw(const char *s, string);
void f2(void) throw(ZeroDevide);
```

Первая функция может генерировать объекты-исключения строкового типа, вторая — объекты-исключения типа `ZeroDevide`.

Если в заголовке скобки спецификации исключений пустые, то считается, что функция исключений не генерирует:

```
void f3(void) throw();
```

Если спецификация исключений объявлена в заголовке, то она должна быть указана и во всех прототипах. Несмотря на это, спецификация исключений не входит в прототип функции, поэтому функции с различными спецификациями исключений не считаются разными при перегрузке.

Наличие спецификации исключений тем не менее не является ограничением при реальной генерации исключений — функция может сгенерировать исключение любого типа. Однако в стандарте определено (см. п. п. 15.5.2 в [1]), что, если

функция генерирует исключение, не заданное в спецификации исключений, запускается стандартная функция `unexpected()`, которая вызывает функцию `terminate()`. Это инициирует аварийное завершение программы.

ПРИМЕЧАНИЕ

К сожалению, не все компиляторы исполняют стандарт в этом пункте. Например, в справочной системе Visual C++.NET 2003 явно написано, что этот пункт стандарта не выполняется — функция `unexpected()` не вызывается. Однако ее разрешено вызывать явным образом.

Применение исключений

Прежде чем переписывать конструкторы и методы разработанных ранее классов с использованием механизма обработки исключений, давайте обратимся к этому механизму для проверки параметров в обычной функции. Пусть нам требуется написать функцию, вычисляющую высоту, опущенную на одну из сторон. Параметрами являются три стороны треугольника, а вычислять нужно высоту, опущенную на сторону, указанную первой. Если параметры-стороны заданы в порядке a , b , c , то вычислять надо высоту, опущенную на сторону a . Как известно, высота, опущенная на сторону a , вычисляется по формуле

$$h_a = \frac{2}{a} \sqrt{p(p-a)(p-b)(p-c)},$$

где $p = (a + b + c)/2$ — полупериметр треугольника. Квадратный корень представляет собой формулу Герона для вычисления площади треугольника. Таким образом, нам надо проверить, что:

- ☐ ни один из параметров (сторона треугольника) не меньше нуля и не равен нулю;
- ☐ выполняются неравенства треугольника для любых сочетаний сторон.

Если эти условия не выполняются, то функция должна генерировать исключение, которое является сообщением об ошибке. Вывод этого сообщения должен быть выполнен в блоке `catch`. Используем тип `string`, а передавать объект-исключение в секцию-ловушку будем по константной ссылке. Нам требуется решить, кто будет обрабатывать исключение: та же функция, где ситуация возникла, или программа-клиент. Рассмотрим сначала первый вариант (листинг 7.1).

Листинг 7.1. Функция вычисления высоты треугольника с обработкой исключений

```
double Ha(double a, double b, double c)
{ try {
    if ((a>0)&&(b>0)&&(c>0)) // контролируемый блок
    { if ((a+b>c)&&(a+c>b)&&(b+c>a)) // если параметры правильные
      { double p = (a+b+c)/2; // если треугольник
        return 2*sqrt(p*(p-a)*(p-b)*(p-c))/a; // вычисляем высоту
      }
    }
    else throw "Не треугольник!"; // генерация исключения
}
```

```

    }
    else throw "Неправильный параметр!"; // генерация исключения
    } // конец try-блока
    catch(const string &s) // обработка исключения
    { cout << s << endl; }
}

```

Уже по тексту функции видно, что механизм обработки исключения в данном варианте является «лишним» — можно просто выводить сообщение. Использование исключений в функции существенно усложнило текст. Тем не менее все-таки проверим функцию с помощью простейшей тестовой программы:

```

int main()
{ cout << Ha(3,4,5) << endl; // выполняется правильно
  cout << Ha(1,2,3) << endl; // ничего не выводится!
  return 0;
}

```

Первый оператор выполняется нормально, так как параметры заданы правильные — это стороны прямоугольного треугольника. А вот со вторым оператором возникли проблемы — программа завершается аварийно, но на экран не выводится никаких сообщений, ради которых мы, собственно, и «городили огород» с исключениями! Такое впечатление, что секция-ловушка просто не выполняется.

Так оно и есть на самом деле! Причина состоит в том, что тип генерируемого объекта-исключения *не соответствует* типу, заявленному в секции-ловушке `catch`. Операторы `throw` генерируют объект, тип которого `const char *`, а в заголовке секции-ловушки заявлен `const string&`. Преобразования типов по умолчанию, как отмечалось ранее, не выполняются. Если бы преобразования были разрешены, невозможно было бы перехватывать исключение конкретного типа. Поэтому перепишем заголовок блока `catch` с правильным типом параметра:

```
catch (const char *s)
```

Программа заработала, однако на экране появляются «лишние» сообщения¹:

```

4
Не треугольник!
-1.#IND

```

Последнее сообщение появляется из-за того, что мы не возвращаем никакого значения в аварийных случаях. Между тем функция, выполнив блок `catch`, выполняет операторы после него, а у нас там ничего нет! Первое решение — поставить в функцию после секции-ловушки оператор

```
return 0;
```

Теперь наша тестовая программа выводит на экран два числа: 4 и 0. Однако использование механизма обработки исключений в таком варианте выглядит абсолютно нецелесообразным — гораздо проще сделать все то же самое с помощью

¹ В Visual C++.NET 2003.

обычных операторов `if`. Поэтому реализуем другой вариант обработки: функция будет только генерировать исключение, а обрабатывать его будет программа-клиент, в данном случае — наша тестовая программа. Текст функции с программой-клиентом представлен в листинге 7.2.

Листинг 7.2. Функция вычисления высоты треугольника с генерацией исключений

```
double Ha(double a, double b, double c)
{ if ((a>0)&&(b>0)&&(c>0))           // если параметры правильные
  { if ((a+b>c)&&(a+c>b)&&(b+c>a))     // если треугольник
    { double p = (a+b+c)/2;          // вычисление высоты
      return 2*sqrt(p*(p-a)*(p-b)*(p-c))/a;
    }
    else throw "Не треугольник!";    // генерация исключения
  }
  else throw "Неправильный параметр!"; // генерация исключения
}

int main()
{ try {                               // контролируемый блок
  cout <<Ha(3,4,5)<< endl;           // нормальное вычисление
  cout <<Ha(1,2,3)<< endl;           // генерирует исключение
  cout <<Ha(1,0,3)<< endl;           // не выполняется!
}
  catch(const char *s)                // обработка исключения
  { cout << s << endl; }
  return 0;
}
```

Обратите внимание на то, что в функции не выполняется возврат «аварийного» значения 0: в данном случае этого не требуется, так как оператор генерации исключения осуществляет *выход из функции*. При этом выполняется «раскрутка» стека: вызываются все необходимые деструкторы для уничтожения локальных объектов. Поэтому вернуться в функцию после генерации исключения невозможно — разве что заново ее вызвать.

Первый оператор вывода срабатывает нормально. Во время выполнения второго возникает исключение, и управление передается в секцию-ловушку. При этом пропускается третий оператор вывода. Причем мы не можем «вернуться» к его выполнению после выполнения секции-ловушки, даже если пометим его и добавим оператор `goto`: по стандарту оператор `goto` не может использоваться для перехода *внутри* контролируемого блока. Но можно передать управление на начало блока `try` и выполнить весь контролируемый блок сначала. Правда, в данном случае — без корректировки параметров во втором вызове — это не изменит поведения программы: третий оператор все равно не будет работать.

Наша функция не имеет спецификации исключений. Наличие спецификации в заголовке никак не изменяет поведения программы, например:

```
double Ha(double a, double b, double c) throw(const char *)
```

Если мы напишем спецификацию исключения с типом исключения, не соответствующим типу генерируемого, то программа должна закончиться аварийно.

Однако в Visual C++.NET 2003 задание спецификации с другим типом исключения на видимое поведение программы никак не влияет, например:

```
double Ha(double a, double b, double c) throw(string)
```

То же самое происходит при полном запрете исключений:

```
double Ha(double a, double b, double c) throw()
```

Главное, чтобы генерируемое исключение было где-нибудь перехвачено.

Передача информации в блок обработки

При обработке исключения в нашей программе просто выводится сообщение. В то же время было бы чрезвычайно полезно иметь дополнительную информацию об ошибке, например, знать конкретные значения параметров функции, которые вызвали сбой программы. Однако параметр в секцию ловушку передается только один — объект-исключение. Поэтому, чтобы передать в блок обработки больше информации, мы должны реализовать собственный тип исключений в виде полноценного класса. Этот класс, естественно, должен быть специализирован под задачу.

В нашем случае, помимо сообщения, полезно передать в блок обработки вместе с объектом-исключением значения параметров, которые получила функция (стороны треугольника) и которые вызвали генерацию исключения. Для этого надо в классе-исключении объявить поля и определить конструктор инициализации. В листинге 7.3 представлен текст этого класса-исключения вместе с функцией и тестовой программой.

Класс-исключение определен с помощью структуры для того, чтобы в обработчике не иметь проблем с доступом к полям. Три поля `double` соответствуют сторонам треугольника, а еще одно поле предназначено для сообщения об ошибке. Теперь мы можем использовать для сообщений тип `string`, так как в структуре объявлен конструктор инициализации, который и заполняет поля. При передаче символьной константы в конструктор выполняется преобразование в тип `string`.

Листинг 7.3. Класс-исключение. Передача информации в обработчик

```
struct ErrorTriangle
{ double a, b, c; // параметры функции
  string message; // сообщение
// конструктор
ErrorTriangle(double x, double y, double z, const string& s)
: a(x), b(y), c(z),
  message(s) {} // пустое тело
}
// указана спецификация исключений
double Ha(double a, double b, double c) throw(ErrorTriangle)
{ if ((a>0)&&(b>0)&&(c>0))
{ if ((a+b>c)&&(a+c>b)&&(b+c>a))
{ double p = (a+b+c)/2;
  return 2*sqrt(p*(p-a)*(p-b)*(p-c))/a;
}
}
```


Листинг 7.3 (продолжение)

```

    else throw ErrorTriangle(a,b,c,"Не треугольник!"); // конструктор
  }
  throw ErrorTriangle(a,b,c,"Неправильные параметры!"); // конструктор
}
int main()
{ try {
    cout <<Ha(3,4,5)<< endl;           // нормально выполняется
    cout <<Ha(1,2,3)<< endl;           // вызывает исключение
    cout <<Ha(1,0,3)<< endl;           // не выполняется
  }
  catch(const ErrorTriangle& e)         // получаем полноценный объект
  { cout << e.message << endl;         // используем поля
    cout << e.a << ' ' << e.b << ' ' << e.c << endl;
  }
}

```

В операторе `throw` в качестве выражения генерации исключения указан явный вызов конструктора. Секция-ловушка получает объект-исключение по константной ссылке и использует информацию, предоставленную конструктором, для вывода значений на экран.

Классы и исключения

Исключения предоставляют прекрасный механизм решения проблем, связанных с ошибками при выполнении конструкторов и перегрузке операций. Перепишем конструкторы и операции ранее написанных нами классов. Начнем с класса `TString` и операции индексирования (см. листинг 4.9). Первое, что требуется определить, — это типы исключений. Не будем пользоваться стандартными встроенными типами, а определим собственный тип с «говорящим» названием. Воспользуемся возможностью объявлять внутри класса вложенный класс и объявим в открытой части класса `TString` пустой класс-исключение, например:

```
class bad_Index {};
```

Теперь мы сможем генерировать объекты-исключения типа `bad_Index` всякий раз, когда индекс символа окажется вне допустимого диапазона 0–254.

Перепишем сначала методы индексирования — теперь мы можем обрабатывать ошибку индексирования одинаково в обоих методах: так как нам нужно только сообщить клиенту об аварии, функция должна только генерировать исключение, а перехватывать и обрабатывать его будет программа-клиент. С учетом этих соображений новый текст методов индексирования представлен в листинге 7.4.

Листинг 7.4. Операция индексирования с генерацией исключений

```

char& operator[](const byte &index)
{   if (index<255) return s[index];           // правильный индекс
    else throw bad_Index();                     // аварийное завершение
}
const char& operator[](const byte &index) const
{   if (index<255) return s[index];           // правильный индекс
    else throw bad_Index();                     // аварийное завершение
}

```

Где-то в программе-клиенте, которая использует наш класс TString, должны быть написаны такие строки:

```
try { //...
    TString ss;                                // создаем объект-строку
//...
    ...ss[i]...                                // контролируется!
//...
}
catch (TString::bad_Index)                    // по типу
{ cout << "Ошибка! Индекс символа вне диапазона [0,254]!" << endl;
  //...
}
//...
```

Как обычно, тип исключения (в заголовке секции-ловушки) требуется писать с префиксом-именем класса, в котором этот тип определен.

Теперь проанализируем конструкторы класса TString. Очевидных претендентов на генерацию исключения bad_Index два — те, которые имеют аргумент-индекс (см. листинги 4.6 и 4.7). Текст модифицированных конструкторов представлен в листинге 7.5.

Листинг 7.5. Конструкторы класса TString с генерацией исключения bad_Index

```
TString::TString(const char S[], byte index, byte count)
{ if (index<255)                                // проверка индекса
  { TString t(S+index, S+index+count);
    *this = t;
  }
  else throw bad_Index();                        // аварийное завершение
}
TString::TString(const TString &S, byte index, byte count)
{ if (index<255)                                // проверка индекса
  { TString t;
    if ((count > S.size)||                        // проверяем количество
        (index+count > S.size))
        t.size = S.size-index;                  // неправильное количество
    else t.size = count;                         // правильное количество
    if (t.size>0)
        memcpy(t.s, (S.s+index), t.size);      // копируем строку
    *this = t;                                  // заменяем текущий
  }
  else throw bad_Index();                        // аварийное завершение
}
```

Теперь при создании объекта с помощью этих конструкторов индекс будет контролироваться, и в случае неправильного значения генерируется исключение. Заметьте, что таким образом мы контролируем очень многие методы и операции — ведь они выполняют свою работу, создавая локальные объекты с помощью одного из этих конструкторов.

Для получения дополнительной информации о том, в какой из функций произошла ошибка, мы, конечно, должны определить в качестве исключения более

развитый класс с конструктором. Можно, конечно, контролировать и параметр `count`, но для строк особого смысла в этом нет, так как мы просто усекаем строку до 255 символов.

Но нам потребуется еще один тип исключений — для проверки диапазона указателей в конструкторе:

```
TString(const char *First, const char *Last);
```

Назовем этот тип исключения `bad_Range`. Тогда новый вид конструктора будет таким, как показано в листинге 7.6.

Листинг 7.6. Конструктор с проверкой диапазона

```
TString::TString(const char *First, const char *Last)
{ if (First < Last)
  { TString t;
    if (Last - First > 255) t.size = 255;
    else                    t.size = Last - First;
    memcpy(t.s, First, t.size);
    *this = t;
  }
  else throw bad_Range();
}
```

Теперь займемся классом `TArray` (см. листинг 5.2). Претендентами на изменения являются все методы и конструкторы, в которых в качестве аргументов задаются индексы, размер или указатели начала и конца последовательности, их прототипы представлены в листинге 7.7.

Листинг 7.7. Конструкторы и методы класса `TArray`, генерирующие исключения

```
TArray(UINT size, double k=0.0);
TArray(const double *begin, const double *end);
TArray(const TArray &a, UINT begin, UINT k);
double& operator[](UINT index);
const double& operator[](UINT index) const;
TArray& assign(const TArray &a, UINT begin, UINT k);
TArray& assign(const double *begin, const double *end);
double summa(UINT l, UINT r);
double product(UINT l, UINT r);
```

Исключительная ситуация может возникать, когда левый индекс больше правого или заданный индекс (`index`) больше размера. Если мы хотим различать эти случаи, то должны определить два класса исключений. Пусть это будут следующие классы:

```
class bad_Index {};           // индекс вне диапазона
class bad_Range {};          // левый индекс больше правого
```

Однако из нашего списка можно исключить методы `assign()`, `summa()` и `product()`, так как в реализации (см. листинги 5.14 и 5.16) для создания локального объекта используется один из конструкторов, который проверяет параметры — он и генерирует исключение.

Однако в некоторых методах выполняется проверка на соответствие размеров массивов — это все методы, которые оперируют двумя массивами:

```
TArray& operator+=(const TArray &rhs);
TArray& operator-=(const TArray &rhs);
TArray& operator*=(const TArray &rhs);
TArray& operator/=(const TArray &rhs);
TArray& operator%=(const TArray &rhs);
friend double product_dot (const TArray &a, const TArray &b);
friend TArray operator+(const TArray &a, const TArray &b);
friend TArray operator*(const TArray &a, const TArray &b);
friend TArray operator-(const TArray &a, const TArray &b);
friend TArray operator/(const TArray &a, const TArray &b);
friend TArray operator%(const TArray &a, const TArray &b);
```

Для контроля размеров определим еще один тип исключения:

```
class bad_Size {};           // для неправильного размера
```

Исключение этого типа генерируется конструктором при задании нулевой длины, а также при несовпадении размеров массивов в операциях с двумя массивами.

Классы исключений должны быть определены в открытой части класса TArray (см. листинг 5.2), чтобы клиент, использующий класс, мог перехватить эти исключения. Тогда новая реализация конструкторов и методов выглядит так, как показано в листинге 7.8.

Листинг 7.8. Конструкторы и методы класса TArray с исключениями

```
// конструкторы
TArray::TArray(UINT size, double k)
{
    if (size > 0)
    {
        size_array = size;
        data = new double[size_array];
        for(UINT i = 0; i<size_array; ++i)    data[i] = k;
    }
    else throw bad_Size();                  // неправильный размер
}

TArray::TArray(const double *begin, const double *end)
{
    if (begin < end)                        // защита
    {
        size_array = (end - begin);
        data = new double[size_array];
        for(UINT i = 0; i<size_array; ++i) data[i] = *(begin+i);
    }
    else throw bad_Range();                 // неправильный диапазон
}

TArray::TArray(const TArray &a, UINT begin, UINT k)
{
    if ((begin<a.size())&&(k<a.size()-begin))
    {
        size_array = k;
        data = new double[size_array];
        for(UINT i = 0; i<size_array; ++i)
            data[i] = a.data[i+begin];
    }
    else throw bad_Index();                 // неправильный индекс
}
```

продолжение ➤

Листинг 7.8 (продолжение)

```

// операция индексирования
double& TArray::operator[](UINT index)
{ if (index<size_array) return data[index];
  else throw bad_Index();           // неправильный индекс
}
const double& TArray::operator[](UINT index) const
{ if (index<size_array) return data[index];
  else throw bad_Index();           // неправильный индекс
}
// другие методы
TArray& TArray::operator+=(const TArray &rhs)
{ if (size_array == rhs.size())     // защита
  { for(UINT i = 0; i<size_array; ++i)
    data[i] = data[i]+rhs.data[i];
    return *this;
  }
  else throw bad_Size();             // неправильный размер
}
TArray operator+(const TArray &a, const TArray &b)
{ TArray temp = a;
  if (a.size() == b.size())         // защита
  { temp+=b; return temp; }
  else throw TArray::bad_Size();    // неправильный размер
}

```

В последней функции при возникновении исключения автоматически вызывается деструктор, чтобы уничтожить локальный объект `temp`.

Остальные методы и функции, оперирующие двумя массивами, реализуются совершенно аналогично. Как обычно, в функциях-друзьях нам приходится указывать уточненное имя объекта-исключения, тогда как в функциях-методах разрешается писать имя без префикса `TArray`.

Теперь рассмотрим контейнер-дек `TDeque` (см. главу 6). Явными претендентами на изменения являются методы удаления элементов `pop_front()` и `pop_back()` (см. листинги 6.4 и 6.5), а также операция разыменования `operator *` класс-итератора (см. листинг 6.2). Так как удалять из пустого дека нельзя, методы удаления должны генерировать исключение — назовем его `emptyDeque`. Естественно, соответствующий класс должен быть прописан в открытой части класса-контейнера:

```
class emptyDeque {};
```

Тогда реализация этих методов будет такой, как показано в листинге 7.9.

Листинг 7.9. Методы удаления из дека с генерацией исключений

```

void TDeque::pop_front()
{ if (!isEmpty())
  { Elem *p = Head;
    Head = Head->next;
    Head->prev = 0;
    head = iterator(Head);
  }
}

```

```

        --count;
        delete p;
    }
    else throw emptyDeque();           // генерация исключения
}
void TDeque::pop_back()
{
    if (!isEmpty())
    {
        Elem *p = Tail->prev;
        if (p == 0) pop_front();
        else
        {
            p->prev->next = Tail;
            Tail->prev = p->prev;
            delete p;
        }
        --count;
    }
    else throw emptyDeque();           // генерация исключения
}

```

Операция разыменования итератора `operator *`, очевидно, должна генерировать другое исключение — назовем его `invalidIterator`. Текст операции с генерацией исключения представлен в листинге 7.10.

Листинг 7.10. Операция разыменования с генерацией исключения

```

value_type& operator*() const
{
    if (the_elem != 0) return the_elem->item;
    else throw invalidIterator();
}

```

Нужно еще обратить внимание на то, что операция `new` может генерировать *стандартное* исключение¹ `bad_alloc` (см. п. п. 18.4.2.1 в [1]), если запрос на память не может быть удовлетворен. Таким образом, программа-клиент, использующая наши динамические контейнеры, обязана перехватывать и обрабатывать не только объявленные нами исключения, но и исключение `bad_alloc`.

Отметим, что операция `new` выполняется в конструкторе, однако утечки памяти не происходит, так как память не была выделена, — вместо этого генерируется исключение. И в этом случае выполняется обычная раскрутка стека, то есть для всех локальных объектов, определенных в конструкторе, вызываются деструкторы и объекты уничтожаются. Затем начинается поиск подходящей секции-ловушки.

Исключения в списке инициализации конструктора

Как мы знаем, инициализировать поля класса можно с помощью списка инициализации (см. листинг 2.7), причем в качестве инициализирующего выражения может стоять вызов функции или явный вызов конструктора. Тогда возникает естественный вопрос: а что, если при выполнении инициализирующих выражений

¹ Стандартные исключения описаны далее в этой главе.

в списке возникнет исключение? Хорошо бы перехватить и обработать его тут же, в конструкторе. Однако контролируемый блок `try...catch` не позволяет справиться с этой проблемой, так как он определяется в теле конструктора, а список инициализации находится вне тела и выполняется до начала выполнения тела. Специально для решения этой проблемы был придуман и внесен в стандарт контролируемый блок, являющийся телом функции. В стандарте такой блок имеет название *function-try-block*, что можно перевести как «контролируемый блок-функция». В стандарте (см. п. п. 15/3 в [1]) приводится следующий пример, представленный в листинге 7.11.

Листинг 7 11. Контролируемый блок — тело конструктора

```
int f(int);                      // функция, определенная в другом месте
class C
{ int i; double d;              // поля
public:
    C(int, double);             // конструктор инициализации
// ...
};
C::C(int ii, double id)         // реализация конструктора
try                             // контролируем
: i(f(ii)), d(id)               // список инициализации
{ // ...                        // тело конструктора
}
catch(...)                     // обработчик исключений
{ // ...
}
```

Обратите внимание, что слово `try` написано перед списком инициализации конструктора — этим подчеркивается, что список инициализации контролируется. Любое исключение, которое сгенерирует функция `f()`, будет перехвачено и обработано в секции-ловушке. При такой организации конструктора отсутствуют операторы после блока `try...catch`. Но они и не нужны, так как если конструктор перехватил исключение, значит, объект «не доделан» и нормального завершения конструирования ждать не приходится. Выход из такой ситуации только один — выполнить ту обработку, которая возможна, и послать исключение дальше, выполнив один из операторов:

- `throw` без аргумента — перехваченное исключение транслируется далее;
- `throw` с аргументом — генерируется новое исключение.

Интересно, что можно объявить контролируемый блок-функцию в любой функции. Например, мы можем контролировать все исключения, возникающие в теле главной функции, написав такую конструкцию:

```
int main()
try { // контролируемый блок — тело функции ...
}
catch(...)
{ // сообщения об исключениях ...
}
```

Если в контролируемом блоке не возникнет исключений, то блок функции нормально завершается. Если же исключения возникнут, то, как обычно, выполняется секция-ловушка, и только после этого программа завершается. Секций-ловушек, естественно, может быть прописано столько, сколько необходимо.

Если подобная конструкция используется в обычной функции или в методе, то выйти из секции-ловушки разрешается обычным оператором возврата `return`.

ПРИМЕЧАНИЕ

Не все компиляторы поддерживают контролируемый блок-функцию. Например, в компиляторе Borland C++ Builder 6 эта форма блока `try` не поддерживается.

Исключения и деструкторы

Теперь разберемся с тем, как и в каком порядке вызываются деструкторы при генерации исключения. Для этого, во-первых, добавим в деструктор класса `TArray` вывод сообщения, чтобы видеть вызов деструктора:

```
TArray::~TArray()
{
    cout << "Destructor" << endl;
    delete[] data; data = 0;
};
```

Аналогично модифицируем конструктор копирования, чтобы видеть на экране входы в конструктор.

Напишем несколько функций, последовательно вызывающих друг друга. В каждой функции создадим «умный» массив с помощью конструктора копирования (чтобы конструктор не вызывался при передаче параметров, будем передавать их по константной ссылке). В самой «внутренней» функции сгенерируем исключение. И наконец, в главной программе будем перехватывать и обрабатывать возникающие исключения (листинг 7.12).

Листинг 7.12. Проверка «раскрутки» стека

```
TArray f2 (const TArray &t)           // самая "внутренняя" функция
{ cout << "function f2" << endl;      // отслеживаем вход
  TArray r(t);                       // конструируем массив
  cout << r << endl;                  // выполняется
  throw TArray::bad_Index();          // генерация исключения
  return r;                           // не выполняется!!!!
}

TArray f1 (const TArray &t)           // отслеживаем вход
{ cout << "function f1" << endl;      // создается три массива
  TArray R[] = { t, t, t };           // выполняется
  cout << R[0] << endl;               // выполняется
  cout << R[1] << endl;               // выполняется
  cout << R[2] << endl;               // выполняется
  TArray r = f2(t);                   // r не создается - не "успевает"
  return r;                           // не выполняется!!!!
}

TArray f(const TArray& a)
```

продолжение ➤

Листинг 7.12 (продолжение)

```

{ cout << "function f" << endl;           // отслеживаем вход
  TArray t = f1(a);                       // t не создается – не "успевает"
  cout << t << endl;                     // не выполняется!!!!
  return t;                               // не выполняется!!!!
}
int main()
{ double a[5] = {1,2,3,4,5};
  TArray B(a, a+(sizeof(a)/sizeof(double)));
  try { f(B);                             // контролируем
  }
  // секции-ловушки
  catch(TArray::bad_Index) {cout <<"Индекс плохой!"<< endl;}
  catch(TArray::bad_Range) {cout <<"Диапазон плохой!"<< endl;}
  catch(TArray::bad_Size) {cout <<"Размер плохой!"<< endl;}
  // непредвиденное исключение
  catch(...){cout <<"Exceptions!"<< endl;}
  return 0;
}

```

При запуске программы мы увидим на экране следующее:

```

function f
function f1
Конструктор копирования!
Конструктор копирования!
Конструктор копирования!
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
function f2
Конструктор копирования!
1 2 3 4 5
Destuctor
Destuctor
Destuctor
Destuctor
Индекс плохой!

```

Последовательность действий в этом примере следующая:

1. В главной функции объявляется массив *a* и из него конструируется «умный» массив *B*.
2. Выполняется вход в контролируемый блок и вызывается функция *f()*, которой передается параметр — «умный» массив *B*.
3. В функции первым выполняется оператор вывода, и на экране появляется первая строка:

```
function f
```

4. Во второй строке функции *f()* прописано объявление «умного» массива *t*, однако до вызова конструктора дело не доходит, так как вызывается функция *f1()*.
5. Первый оператор функции *f1()* выводит на экран строку:

```
function f1
```

6. В функции `f1()` создается массив `R` из трех «умных» массивов, и конструктор копирования вызывается трижды.
7. Выполняется вывод на экран трех элементов-массивов.
8. Хотя объявление «умного» массива `r` прописано, вызова конструктора не происходит, так как вызывается функция `f2()`.
9. Функция `f2()` выводит на экран строку:

```
function f2
```

Создается локальный «умный» массив `r`, конструктор копирования отрабатывает один раз.

10. Содержимое массива `r` выводится на экран.

11. Генерируется исключение `bad_Index`.

К этому моменту у нас создаются один локальный массив в функции `f2()` и массив `R` из трех массивов в функции `f1()`, которые находятся в стеке. При генерации исключения деструктор вызывается четыре раза, что мы и наблюдаем на экране. После этого происходит переход в секцию-ловушку и выводится строка «Плохой индекс». Затем выполняется выход из секции в тело главной функции и программа завершает работу.

Теперь распределим обработку исключения по разным функциям. Добавим в функцию `f1()` контролирующий блок и идентифицируем вывод в секции-ловушке в главной программе (листинг 7.13).

Листинг 7.13. Распределенная обработка исключения

```
TArray f1 (const TArray &t)
{ cout << "function f1" << endl;
  TArray R[] = { t, t, t };
  cout << R[0] << endl;
  cout << R[1] << endl;
  cout << R[2] << endl;
  try {
    TArray r = f2(t);           // контролируем вызов f2
    return r;                   // локальный объект
                                // нормальное завершение
  }
  catch(TArray::bad_Index)
  { cout <<"f1. Плохой индекс!"<< endl;
    throw;                      // повторная генерация исключения
  };
}

int main()
{ double a[5] = {1,2,3,4,5};
  TArray B(a, a+(sizeof(a)/sizeof(double)));
  try { f(B);                   // контролируем
  }
  // секции-ловушки
  catch(TArray::bad_Index) {cout <<"Main. Индекс плохой!"<< endl;}
  catch(TArray::bad_Range) {cout <<"Диапазон плохой!"<< endl;}
  catch(TArray::bad_Size) {cout <<"Размер плохой!"<< endl;}
```

Листинг 7.13 (продолжение)

```
// непредвиденное исключение
catch(...){cout <<"Exceptions!"<< endl;}
return 0;
}
```

Остальные функции оставим без изменения. При запуске этой программы на экране появится следующее:

```
function f
function f1
Конструктор копирования!
Конструктор копирования!
Конструктор копирования!
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
function f2
Конструктор копирования!
1 2 3 4 5                                // до этого момента поведение повторяется
Destructor                                // при выходе из f2
f1. Плохой индекс!                       // секция-ловушка в функции f1
Destructor                                // выход из f1
Destructor
Destructor
Main. Плохой индекс!                     // секция-ловушка в main()
```

Поведение программы повторяется до генерации исключения. Выполнение оператора `throw` приводит к выходу из функции `f2()`, при этом вызывается деструктор. В вызывающей функции обнаруживается секция-ловушка нужного типа, которая и выполняется. В ее конце срабатывает оператор повторной генерации исключения. Происходит выход из функции `f1()` — при этом трижды вызываются деструктор для уничтожения элементов массива `R`. В вызывающей функции — а это уже главная функция `main()` — ищется секция-ловушка нужного типа. Она у нас в программе есть, и мы наблюдаем ее работу в последней строке, выведенной на экран.

В связи с тем, что деструктор вызывается для уничтожения локальных объектов при генерации любого исключения, сам деструктор генерировать исключения не должен (но, вообще говоря, может). Как указывает Герб Саттер в [21], «все деструкторы должны разрабатываться так, как если бы они имели спецификацию исключения `throw()`, то есть ни одному исключению не должно быть позволено выйти за пределы деструктора». Если же это произойдет, то программа завершается аварийно. Это означает, что исключения, возникающие в деструкторе, должны быть перехвачены и обработаны там же.

Деструкторы наших классов `TArray` и `TDeque` исключений не генерируют — они просто возвращают память. Как мы знаем, по стандарту (см. п. 18.4 в [1]) функции возврата памяти имеют прототипы:

```
void operator delete (void *) throw();
void operator delete [](void *) throw();
```

Пустая спецификация исключений показывает, что стандарт гарантирует отсутствие исключений при возврате памяти.

Стандартные исключения

В языке C++ в составе стандартной библиотеки реализован ряд *стандартных* исключений, которые организованы в *иерархию классов* (см. п. п. 18.6 и 19.1 в [1]). Иерархическое представление часто бывает очень полезно, так как позволяет разбить задачу на части, повысить понимание и упростить реализацию. В этом случае задействуется третий механизм объектно-ориентированного программирования — *наследование*. Подробности наследования мы изучим в следующей главе, а здесь рассмотрим простейший случай организации исключений.

При реализации класса `TArray` с генерацией исключений (см. листинг 7.8) были определены три независимых типа исключений: `bad_range`, `bad_index` и `bad_size`. Но все они имеют нечто общее: эти исключения генерируются в классе `TArray`. Хотелось бы как-то обозначить этот факт явным образом. Это можно сделать, используя механизм наследования: мы объявим класс `TArray_exception`, а остальные классы объявим его *наследниками*:

```
class TArray_exception {};           // базовый класс
class bad_range: public TArray_exception {}; // класс-потомок
class bad_index: public TArray_exception {}; // класс-потомок
class bad_size : public TArray_exception {}; // класс-потомок
```

Мы объявили общий тип исключений, связанных с классом `TArray`, а затем определили ряд более специализированных подвидов типа `TArray_exception`. Тем самым мы получили возможность все исключения, связанные с классом `TArray`, перехватывать и обрабатывать с помощью единственной секции-ловушки с типом параметра `TArray_exception`. Никто, однако, не запрещает использовать и специализированные исключения. Естественно, если нам потребуется задать несколько секций-ловушек для обработки исключений класса `TArray`, то блок с типом исключения `TArray_exception` должен стоять последним, так как это — исключение общего вида.

Несколько более сложную структуру (см. п. 19.1 в [1]) имеет иерархия стандартных исключений (листинг 7.14).

Листинг 7.14. Иерархия стандартных исключений

```
class exception {///... };
class logic_error : public exception {///... };
    class domain_error : public logic_error {///... };
    class invalid_argument : public logic_error {///... };
    class length_error : public logic_error {///... };
    class out_of_range : public logic_error {///... };
class runtime_error : public exception {///... };
    class range_error : public runtime_error {///... };
    class overflow_error : public runtime_error {
        class underflow_error : public runtime_error {///... };
class bad_alloc : public exception {///... };
class bad_cast : public exception {///... };
class bad_typed : public exception {///... };
class bad_exception : public exception {///... };
class ios_base::failure : public exception {///... };
```

Эта иерархия служит основой для создания собственных исключений и иерархий исключений. Мы можем определять собственные исключения, унаследовав их от класса `exception`. Для работы со стандартными исключениями в программе надо прописать оператор

```
#include <stdexcept>
```

ПРИМЕЧАНИЕ

Так как обработка исключений во время выполнения программы сопровождается расходами времени и памяти, в интегрированной среде, как правило, необходимо включить соответствующий режим, разрешающий обработку исключений.

Названия производных классов `logic_error` и `runtime_error`, в общем-то, условны. Предполагается, что исключения первого типа сигнализируют об ошибках в логике программы, например о невыполнении некоторого условия. Категория `runtime_error` — это ошибки, которые возникают в результате непредвиденных обстоятельств при выполнении программы, например переполнение при операциях с дробными числами.

Эти исключения программа должна генерировать сама оператором `throw`. А вот следующие 5 стандартных исключений генерируют различные механизмы C++.

ПРИМЕЧАНИЕ

Мы тоже можем использовать эти исключения в операторе `throw` явным образом, однако так делать не рекомендуется во избежание путаницы.

Как уже было сказано, исключение `bad_alloc` (см. п. п. 18.4.2 в [1]) генерирует операция `new` (или `new[]`), если запрос на память не может быть удовлетворен. Исключения `bad_cast` (см. п. п. 18.5.2 в [1]) и `bad_typeid` (см. п. п. 18.5.3 в [1]) генерируются механизмом RTTI (Run-Time Type Identification — динамическая идентификация типов). Естественно, исключение `ios_base::failure` (см. п. п. 27.4.2.1.1 в [1]) генерируется системой ввода-вывода. Исключение `bad_exception` (см. п. п. 18.6.2.1 в [1]) играет важную роль при нарушении функцией спецификации исключения (см. далее).

Класс `exception` определен в стандартной библиотеке (см. п. п. 18.6/1 в [1]) так, как показано в листинге 7.15.

Листинг 7.15. Стандартный класс исключения

```
class exception {
public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&) throw();
    virtual ~exception () throw();
    virtual const char* what () const throw();
};
```

Что означает слово `virtual`, мы разберемся в главе 9, а сейчас обратите внимание на то, что *все* конструкторы и методы имеют спецификацию, запрещающую генерацию исключений. Это и понятно: если механизм исключений сам будет генерировать исключения, хорошего ждать не придется — программа закончится аварийно.

Как видим, стандартные исключения включают функцию-метод `what()`, которая, очевидно, выдает строку-сообщение об ошибке. Использование стандартных исключений продемонстрируем с помощью немного модифицированного примера из справочника интегрированной среды C++ Builder 6 (листинг 7.16).

Листинг 7.16. Использование метода `what()`

```
#include <stdexcept>
#include <iostream>
#include <string>
using namespace std;
void f() // функция, генерирующая исключение runtime
{ throw runtime_error("Ошибка runtime!"); }
int main ()
{ string s;
  try { // контролируем метод replace класса string
    s.replace(100,1,1,'c');
  }
  catch (const exception& e)
  { cout << "Получили исключение: " << e.what() << endl; }
  try { f(); // контролируем нашу функцию
  }
  catch (const exception& e)
  { cout << "Получили исключение: " << e.what() << endl; }
  return 0;
}
```

Эта программа выведет на экран две строки:

```
Получили исключение: basic_string
Получили исключение: Ошибка runtime
```

В этом простом примере мы видим два варианта генерации исключений: явный — в нашей функции и неявный — в стандартном методе `replace()`. Первая строка — это результат работы первого обработчика при неправильном использовании метода `replace()` строки `s`, вторая — это уже результат явной генерации исключения типа `runtime_error` в функции `f()`.

Обратите внимание на одну замечательную особенность блоков `catch`: в качестве типа формального параметра используется *базовый класс*, тогда как при выполнении в обработчик попадает объект-исключение совсем другого типа — *наследника* от базового. Это очень важное свойство наследования — принцип подстановки — мы рассмотрим при изучении наследования. Кроме того, исключение передается в секцию-ловушку по ссылке. Почему это важно — тоже разберемся при изучении наследования.

В качестве еще одного простого примера можно привести пример использования стандартных исключений в наших динамических классах. Давайте перепишем

конструкторы класса `TArray` с применением стандартных исключений. Вместо исключения `bad_range` задействуем стандартное исключение `range_error`, наше исключение `bad_index` заменим исключением `invalid_argument`, а вместо исключения `bad_size` используем исключение `length_error`. Это нам позволит получить в секцию-ловушку диагностическую строку. Новая реализация конструкторов выглядит так, как показано в листинге 7.17.

Листинг 7.17. Конструкторы и методы класса `TArray` со стандартными исключениями

```
// конструкторы
TArray::TArray(UINT size, double k)
{
    if (size > 0)
    {
        size_array = size;
        data = new double[size_array];
        for(UINT i = 0; i < size_array; ++i)    data[i] = k;
    }
    else throw length_error("Неправильно задан размер массива");
}

TArray::TArray(const double *begin, const double *end)
{
    if (begin < end)                // защита
    {
        size_array = (end - begin);
        data = new double[size_array];
        for(UINT i = 0; i < size_array; ++i) data[i] = *(begin+i);
    }
    else throw range_error("Неправильный диапазон указателей");
}

TArray::TArray(const TArray &a, UINT begin, UINT k)
{
    if ((begin < a.size()) && (k < a.size() - begin))
    {
        size_array = k;
        data = new double[size_array];
        for(UINT i = 0; i < size_array; ++i)
            data[i] = a.data[i + begin];
    }
    else throw invalid_argument("Неправильный индекс ");
}
```

Использование стандартных исключений позволяет написать единственный обработчик исключений с параметром типа `exception`, который будет перехватывать все генерируемые исключения. А конкретную причину мы узнаем, используя в секции-ловушке метод `what()`, как в предыдущем примере.

Подмена функций стандартного завершения

Стандарт (см. п. 18.6 в [1]) позволяет подменить вызов стандартной функции `terminate()` функцией, определенной программистом. Как уже было сказано, эта функция вызывается либо из функции `unexpected()`, если нарушена спецификация исключения, либо механизмом обработки исключений, если отсутствует подходящая секция-ловушка. Аналогично можно подменить и вызов функции

`unexpected()`. В стандартной библиотеке (надо подключить заголовок `<except-ion>`) прописаны следующие объявления:

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();
void unexpected();
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw();
void terminate();
```

Эти объявления показывают, что для подмены стандартной функции `terminate()` мы должны определить собственную функцию с прототипом

```
void F();
```

Затем нужно прописать ее имя в вызове функции `set_terminate()`:

```
set_terminate(F);
```

Можно сохранить адрес прежнего обработчика:

```
void (*old_terminate)() = set_terminate(F);
```

После этого вместо `terminate()` при обработке неперехваченных исключений будет вызываться наша функция `F()`. Простой пример демонстрирует это (листинг 7.18). Исполнять его необходимо из командной строки, так как при запуске в интегрированной среде она (среда) перехватывает все необработанные исключения.

Листинг 7.18. Подмена функции `terminate()`

```
void f ()
{ cout << "Неперехваченное исключение!" << endl; }
int main()
{   set_terminate(f);           // установка нашей функции
    try { throw 1; }           // генерация неперехватываемого исключения
    catch(double) {};
    return 0;
}
```

При трансляции в системе Visual C++.NET 2003 на экране появятся следующие сообщения:

Неперехваченное исключение!

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

Первое сообщение, очевидно, наше. А вот следующие два — это то, что система вставляет в нашу программу при трансляции исходного текста.

Система C++ Builder 6 ведет себя немного иначе — на экране появляются сообщения:

Неперехваченное исключение!

Abnormal program termination

Последнее сообщение — это сообщение стандартной функции завершения `abort()` (см. п. 18.3 в [1]).

Функция-«терминатор» не должна возвращать управление оператором `return`, не должна генерировать исключение оператором `throw` — она может лишь завершить программу функцией `exit()` или `abort()`. Но перед завершением можно выполнить действия, которые позволят разобраться в причине вызова «терминатора», например, сохранить информацию об ошибке в файле.

Аналогично реализуется подмена стандартной функции `unexpected()`, только надо вызвать другую функцию `set_...`:

```
set_unexpected(f);           // установка нашей функции
```

Функция обработки неперехваченного исключения, как и функция-терминатор, не должна возвращать управление оператором `return` и может завершить программу функцией `exit()` или `abort()`. Однако помимо этого она может сгенерировать исключение, заданное в спецификации исключений. Произойдет подмена неперехваченного исключения «легальным», и далее обработка исключений пойдет «нормальным» путем.

Функция может сгенерировать другое исключение, не указанное в спецификации, или просто «отправить» незааявленное исключение «дальше» оператором `throw`. В этом случае, если в спецификации исключений отсутствует исключение `bad_exception`, вызывается функция `terminate()`. А вот если спецификация исключений содержит исключение `bad_exception`, то сгенерированное исключение подменяется на `bad_exception` и начинается поиск его обработчика.

Как использовать подмену функции `unexpected()` с пользой — показал Скотт Мейерс в [24]. Так как предвидеть тип неожиданного исключения невозможно, предлагается подменить *все* неперехваченные исключения одним типом: либо стандартным исключением `bad_exception`, либо нашим собственным типом, например `UnexpectedException`. Первый вариант реализуется функцией

```
void convertUnexpectedToBad_Exception()
{
    throw;                               // генерация bad_exception
}
set_unexpected(convertUnexpectedToBad_Exception);
```

Второй вариант реализуется так:

```
class UnexpectedException {};           // наше исключение
void convertUnexpected()
{
    throw UnexpectedException();        // генерация нашего исключения
}
set_unexpected(convertUnexpected);
```

Во все спецификации исключений функций надо включить соответствующий тип — и можно забыть о неожиданных исключениях.

Нестандартные исключения

Все, что было написано об исключениях ранее в этой главе, соответствует стандартному механизму. Любой компилятор, если он претендует на гордое звание компилятора C++, обязан этот механизм поддерживать. Однако исключения

были добавлены в стандарт C++ едва ли не в последнюю очередь, поэтому во многих системах были реализованы нестандартные расширения языка C++, связанные с обработкой исключений. Обычно в составе системы программирования поставляются фирменные библиотеки классов, которые включают в себя собственную иерархию классов-исключений. Инструментальная среда порой поддерживает несколько разных механизмов, и бывает непросто разобраться, как и в каких ситуациях корректно использовать те или иные средства. Как правило, объединять в одной программе разные механизмы обработки исключений не рекомендуется, а то и вовсе запрещается. Нужно как-то ориентироваться в этом многообразии, поэтому кратко рассмотрим некоторые возможности обработки исключений, которые не являются стандартными, но предоставляются средой программирования. Рассмотрим систему Visual C++.NET 2003. В справочнике этой системы описаны следующие механизмы обработки исключений:

- стандартный механизм обработки исключений C++ (C++ exception handling);
- обработка исключений в «управляемом» языке C++;
- механизм обработки исключений библиотеки Microsoft Foundation Class (MFC exception handling);
- механизм структурной обработки исключений Windows (Structured Exception Handling, SEH).

Аналогичная картина наблюдается в системе Borland C++ Builder 6 — в справочнике системы описаны:

- стандартный механизм обработки исключений;
- исключения, реализованные в составе библиотеки VCL/CLX;
- механизм SEH.

Поскольку в обеих системах реализован механизм SEH, рассмотрим его подробнее на примере системы Visual C++.NET 2003.

Структурная обработка исключений (SEH) [47] реализована и в Visual C++.NET 2003 [45] и в C++ Builder 6 [49] как расширение стандартного языка C++. Поддержка SEH обеспечивается операционной системой Windows, поэтому программы, в которых используется SEH, *непереносимы*. Кроме того, при структурной обработке исключений *не выполняется вызов деструкторов* — это наиболее важное отличие SEH от стандартного механизма. Поэтому Microsoft не рекомендует смешивать стандартные и структурные исключения в одной программе.

Структурная обработка исключений предоставляет две возможности: обработку исключений `try...except` и обработку завершения `try...finally`. Обработка завершения проще, поэтому начнем с нее. В системе Visual C++.NET 2003 синтаксис обработчика завершения выглядит так:

```
_try { }           // защищенный блок  
_finally { }       // блок завершения
```

Ключевые слова `_try` и `_finally` пишутся с двумя подчеркиваниями — обычно в системе Visual C++.NET 2003 так обозначаются все расширения стандартного языка C++.

Обработчик завершения гарантирует, что блок завершения будет выполнен при любой попытке выхода из защищенного блока — независимо от способа выхода. Выход из блока `_try` осуществляется одним из следующих способов:

- нормальное выполнение всех операторов блока от начала до конца;
- возникновение исключения во время выполнения операторов блока;
- выполнение одного из операторов перехода (`break`, `continue`, `goto`, `return`), хотя этого в [47] рекомендуется избегать.

Поведение программы после выполнения финального блока зависит от способа выхода из блока `_try`. Если в финальном блоке нет никаких операторов перехода и он выполняется до конца, то далее в соответствии со способом выхода из защищенного блока происходит следующее:

- выполняются операторы после финального блока;
- выполняется оператор перехода, который вызвал выход из защищенного блока.

Синтаксис обработчика исключений выглядит так:

```
_try { }           // защищенный блок
_except (фильтр)
{ }               // блок обработки исключения
```

Обратите внимание, что в SEH после защищенного блока следует *единственный* обработчик: это либо обработчик завершения, либо обработчик исключения. Несколько обработчиков писать нельзя, тем более нельзя писать несколько разных обработчиков. Однако разрешается конструкцию `try...finally` вкладывать в блок `try...except` и наоборот, уровень вложенности не ограничен. Правда, в этом случае существует опасность «запутаться» в порядке выполнения блоков `_except` и `_finally`.

При нормальном ходе выполнения программы (когда исключения не возникает) после выполнения операторов защищенного блока блок-обработчик пропускается, и программа продолжает работу со следующего после него оператора. А вот если при выполнении операторов защищенного блока возникло исключение, то обработка его зависит от фильтра. Фильтр — это выражение (в том числе вызов функции), которое должно принимать одно из трех возможных значений:

- `EXCEPTION_CONTINUE_EXECUTION (-1)`;
- `EXCEPTION_CONTINUE_SEARCH (0)`;
- `EXCEPTION_EXECUTE_HANDLER (1)`.

Очень часто эти константы задаются в качестве фильтра непосредственно. Только последний вариант означает собственно обработку исключения: выполняются операторы блока обработки, после чего управление передается на первый оператор¹ после него.

Если результат вычисления фильтра равен первой константе, то исключение отклоняется (`exception is dismissed`). Код в обработчике прерываний *никогда не*

¹ Естественно, если в блоке обработки нет операторов передачи управления.

выполняется. Управление возвращается в точку возникновения прерывания¹ и делается попытка снова выполнить ту же инструкцию, которая вызвала исключение. На первый взгляд поведение не совсем логичное, ведь снова возникнет исключение. Однако не будем забывать, что фильтр — это выражение. Это означает, что на месте фильтра может быть задано несколько выражений через запятую. В этих выражениях можно исправить причину, вызвавшую исключение. Если необходимо выполнить более сложную работу, то в качестве фильтра может быть прописан вызов функции. Функция может получить в качестве параметров любые переменные из защищенного блока, выполнить с ними нужную работу и вернуть константу `EXCEPTION_CONTINUE_EXECUTION` в качестве результата. Например, если произошло деление на ноль, то функция может изменить значение делителя и программа «похромает» дальше.

Фильтр `EXCEPTION_CONTINUE_SEARCH` по своему действию похож на оператор `throw` без аргумента, то есть он «отправляет исключение дальше». Код в обработчике с таким фильтром не выполняется никогда. И только третий вид фильтра обеспечивает выполнение тела обработчика.

Генерация программных исключений выполняется API-функцией Windows под названием `RaiseException()`. Ее мы рассматривать не будем — подробное изучение SEH в нашу задачу не входит. Однако на одном моменте нужно остановиться. Чем хорош механизм SEH, так это возможностью «отлавливать» аппаратные исключения и точно их идентифицировать, естественно, с помощью API-функции:

```
DWORD GetExceptionCode();
```

Функция возвращает идентификатор типа исключения, которые определены в файле `WinBase.h`. Но подключать надо заголовок `<windows.h>`. В табл. 7.1 представлены некоторые из этих идентификаторов.

Таблица 7.1. Аппаратные исключения

Значение	Описание
<code>EXCEPTION_ACCESS_VIOLATION</code>	Попытка обратиться к «чужой» памяти
<code>EXCEPTION_FLT_DIVIDE_BY_ZERO</code>	Деление на ноль дробных чисел
<code>EXCEPTION_FLT_OVERFLOW</code>	Переполнение для дробных чисел
<code>EXCEPTION_INT_DIVIDE_BY_ZERO</code>	Деление на ноль для целых чисел
<code>EXCEPTION_INT_OVERFLOW</code>	Переполнение для целых чисел
<code>EXCEPTION_STACK_OVERFLOW</code>	Переполнение стека

В составе Windows API реализованы и другие функции, используемые в SEH. Останавливаться подробно мы на них не будем, так как это не входит в нашу задачу. В справочной системе Visual C++.NET 2003 можно найти пример, который с некоторыми модификациями представлен в листинге 7.19.

¹ Это еще одно отличие SEH от стандартных исключений.

Листинг 7.19. Структурная обработка исключений

```
// exceptions_try_except_Statement.cpp
// Пример try-except и try-finally
#include <iostream>
#include <windows.h> // для EXCEPTION_ACCESS_VIOLATION
using std::cout;
using std::endl;
// функция-фильтр
int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep) {
    cout<<"in filter."<< endl;
    if (code == EXCEPTION_ACCESS_VIOLATION) {
        cout<<"caught AV as expected."<<endl;
        return EXCEPTION_EXECUTE_HANDLER;    // если авария
    }
    else {
        cout<<"didn't catch AV, unexpected."<< endl;
        return EXCEPTION_CONTINUE_SEARCH;    // если нет аварии
    };
}

int main()
{
    int* p = 0x00000000;                // специально нулевой указатель
    cout<<"hello"<< endl;
    _try{
        cout<<"in try 1"<< endl;
        _try{
            cout<<"in try 2"<< endl;
            *p = 13;                    // access violation
        }_finally{
            cout<<"in finally. termination: "<< endl;
            cout<<(AbnormalTermination() ? "\tabnormal" : "\tnormal")<< endl;
        }
    }_except(filter(GetExceptionCode(), GetExceptionInformation())){
        cout<<"in except"<< endl;
    }
    cout<<"world"<< endl;
}
```

Программа специально содержит ошибку (нулевой указатель), чтобы показать обработку возникающего исключения по нарушению доступа к памяти (access violation). Демонстрируется также обязательное выполнение финального блока, несмотря на исключение. Программа выводит на экран следующее:

```
hello
in try 1
in try 2
in filter.
caught AV as expected.
in finally. termination:
    abnormal
in except
world
```

Механизм SEH очень подробно описан в [46].

Резюме

Механизм обработки исключений — это объектно-ориентированный механизм обработки ошибок, возникающих при неправильной работе программ. Этот механизм был добавлен в C++, в первую очередь, в связи с необходимостью как-то реагировать на некорректное поведение программы во время работы конструкторов и перегруженных операций.

В языке C++ исключения являются объектами любого допустимого типа. В стандартной библиотеке определен ряд стандартных исключений, которые можно использовать в своей программе. Можно определить собственные типы исключений с помощью классов. Программа может генерировать (порождать) объект-исключение, контролировать возникновение исключения, перехватывать и обрабатывать его. Если исключение не перехватывается программой, то система предоставляет стандартный механизм обработки, выполняющий аварийное завершение программы.

Исключение может быть сгенерировано в одном месте программы, а обработано совершенно в другом. Если в процессе поиска подходящего блока обработки потребуется выйти из вложенных вызовов функций, то выполняется «раскрутка» стека, в процессе которой осуществляется корректный вызов деструкторов для всех локальных объектов.

В заголовке функции может быть задана спецификация исключений. Если функция генерирует исключение, отличающееся от указанных в спецификации, то по стандарту должно быть инициировано аварийное завершение программы. Программе разрешается вместо стандартных установить собственные функции обработки неперехваченных исключений и аварийного завершения программы.

Контрольные вопросы

1. Объясните, зачем нужны исключения?
2. Назовите ключевые слова C++, которые используются при обработке исключений.
3. Исключение — это:
 - 1) событие;
 - 2) ситуация;
 - 3) объект;
 - 4) ошибка в программе;
 - 5) прерывание.
4. Каким образом исключение генерируется?
5. Каковы функции контролируемого блока?

6. Что обозначается ключевым словом `catch`?
 - 1) контролируемый блок;
 - 2) блок обработки исключения;
 - 3) секция-ловушка;
 - 4) генератор исключения;
 - 5) обработчик прерывания.
7. Какого типа может быть исключение?
8. Сколько параметров разрешается писать в заголовке секции-ловушки?
9. Какими способами разрешается передавать исключение в блок обработки?
10. Объясните, каким образом преодолеть ограничение на передачу единственного параметра в блок обработки.
11. Почему нельзя выполнять преобразования типов исключений при передаче в секцию-ловушку?
12. В каких случаях преобразование типа все же выполняется?
13. Напишите конструкцию, которая позволяет перехватить любое исключение.
14. Могут ли контролируемые блоки быть вложенными?
15. Объясните термин «раскрутка стека».
16. Зачем нужен «контролируемый блок-функция» и чем он отличается от обычного контролируемого блока?
17. Перечислите возможные способы выхода из блока обработки.
18. Каким образом исключение «передать дальше»?
19. Сколько секций-ловушек должно быть задано в контролируемом блоке?
20. Что такое «спецификация исключений»?
21. Что происходит, если функция нарушает спецификацию исключений?
22. Учитывается ли спецификация исключений при перегрузке функций?
23. Отличается ли функция с «пустой» спецификацией `throw()` от функции, у которой отсутствует спецификация исключений?
24. Что такое «стандартные» исключения? Назовите два-три типа стандартных исключений.
25. Поясните «взаимоотношение» исключений и деструкторов.
26. Объясните, зачем может понадобиться подмена стандартных функций завершения.
27. Может ли конструктор генерировать исключение? А деструктор?
28. Какие виды нестандартных исключений вы знаете?
29. В чем отличие механизма структурной обработки исключений Windows от стандартного механизма?
30. Какие стандартные функции можно подменить, и каким образом это делается?

Упражнения

1. Написать три варианта функции, которая вычисляет корень квадратного уравнения $ax^2 + bx + c = 0$. Все три варианта должны генерировать исключения в случае какой-либо ошибки. Первый вариант функции должен включать полную спецификацию исключений, второй — «пустую» спецификацию `throw()`, в третьем варианте спецификация исключений должна отсутствовать. Определить собственные типы исключений.
2. Выполнить предыдущее задание, реализовав подмену стандартной функции `unexpected()`. Пользовательская функция должна выводить сообщение об отсутствии обработчика исключения и заканчивать работу.
3. Реализовать класс-стек (см. листинг 6.9), обеспечив обработку исключений. Исключения определить как независимые пустые классы.
4. Определить подходящие исключения в открытой части и модифицировать методы класса `Double` (см. упражнение 2 в главе 2): в случае возникновения ошибок все методы должны генерировать подходящие исключения.
5. Создать класс `Triangle` для представления треугольника. Поля данных являются сторонами треугольника. Определить подходящие исключения. Должны быть реализованы операции получения и изменения полей данных, вычисления площади, вычисления периметра, вычисления высот, вычисления углов, определения вида треугольника (равносторонний, равнобедренный, прямоугольный). В случае возникновения ошибок методы должны генерировать исключения.
6. Модифицировать класс `DoubleArray` (см. упражнение 1 в главе 5), обеспечив генерацию исключений в аварийных ситуациях. Задействовать подходящие стандартные исключения; в секциях-ловушках использовать метод `what()`.
7. Реализовать динамический класс `Rational` (см. упражнение 10 в главе 2), обеспечив генерацию исключений в случае ошибок. Определить спецификации исключений во всех методах, где необходимо. Задействовать подходящие стандартные исключения; в секциях-ловушках использовать метод `what()`.
8. Реализовать динамический класс `Money` (см. упражнение 4 в главе 4), обеспечив обработку исключений. Переопределить функцию завершения `terminate()`.
9. Написать функцию, выполняющую деление комплексных чисел A и B :

$$(a, b) / (c, d) = (ac + bd, bc - ad) / (c^2 + d^2).$$

Комплексное число представлено структурой с двумя полями: `first` и `second`. Определить в заголовке спецификацию исключения, используя собственные исключения. В случае ошибки функция должна генерировать неперехваченное исключение. В контролируемом блоке определить секцию-ловушку, которая перехватывает все исключения.

10. Написать функцию, выполняющую деление комплексных чисел. Определить в заголовке спецификацию исключения, используя стандартные исключения.

В случае ошибки функция должна генерировать неперехваченное исключение. Осуществить подмену неперехваченного исключения стандартным исключением `bad_exception`.

11. Реализовать функцию деления комплексных чисел из предыдущего упражнения. Для реализации обработки исключений разработать класс-исключение с полями и конструктором инициализации. Определить в контролируемом блоке несколько секций-ловушек с разными способами передачи параметра.
12. Выполнить упражнение 3, определив собственные исключения как наследники от стандартного класса-исключения. Классы-наследники должны содержать необходимые поля данных, а секции-ловушки — выводить информацию о неправильных значениях.

Глава 8

Наследование

В предыдущей главе при знакомстве со стандартными исключениями мы столкнулись с механизмом наследования (см. п. 10 в [1]). *Наследование* — это третий «кит», на котором (наряду с полиморфизмом и инкапсуляцией) стоит объектно-ориентированное программирование (ООП). Наследование — важнейшая и неотъемлемая часть ООП. Именно наследование позволяет повторно использовать существующий код. Имея написанный и работающий базовый класс, мы можем его больше не модифицировать, а механизм наследования даст нам возможность приспособить его для работы в различных ситуациях. Чтобы задействовать существующий код, программист создает новый класс, но не «с нуля», а на базе уже существующих классов. Таким образом, наследование в ООП исполняет две роли: с одной стороны, предотвращает дублирование кодов, с другой стороны, позволяет развивать работу в нужном направлении.

Мы уже создавали класс «на базе существующих классов» и без всякого наследования — вспомните класс `TCount` (см. листинг 1.21). В этом случае мы использовали не специальный языковый механизм, а прием, который применяется уже лет 40 — с того момента, как в языках программирования появились структуры. Как известно, элементом структуры может быть поле — объект другой структуры. Аналогично, элементом класса может быть объект другого класса. В объектно-ориентированном программировании этот прием называется *композицией*, поскольку новый (композитный) класс «собирается» из объектов существующих классов.

Однако при наследовании между классами устанавливаются гораздо более тесные связи, чем при композиции. При наследовании обязательно имеется хотя бы один класс-предок (родитель) и хотя бы один класс-наследник (потомок). Класс-предок часто называют еще суперклассом (или порождающим классом), а класс-наследник — подклассом (порожденным классом). В C++ принято порождающий класс называть базовым, а порожденный класс — производным. Все эти термины эквивалентны, и мы будем использовать их все.

Отношения между родительским классом и его потомками называются иерархией наследования. Мы уже познакомились с простейшим вариантом этого механизма, когда рассматривали иерархию исключений. Вообще говоря, глубина наследования ничем не ограничена: мы можем наследовать столько раз, сколько требуется для решения нашей задачи.

Использовать наследование для создания развитой иерархии достаточно сложно — для этого требуется сначала разработать классификацию моделируемых объектов. Например, всем известны виды четырехугольников: квадрат, ромб, прямоугольник, параллелограмм, трапеция, произвольный четырехугольник. Можно построить простую иерархию от специального четырехугольника к произвольному, например:

Квадрат Прямоугольник Четырехугольник

А можно построить и такую иерархию:

Четырехугольник Трапеция Параллелограмм Прямоугольник Квадрат

Или такую:

Четырехугольник Параллелограмм Ромб Квадрат

Однако может быть построена и более сложная иерархия, показанная на рис. 8.1. В данном случае учитывается, что квадрат обладает и свойствами прямоугольника (прямые углы), и свойствами ромба (равные стороны).



Рис. 8.1. Иерархия наследования

Первые два варианта иерархии реализуются путем простого наследования, а последний — посредством множественного. Эти примеры показывают, что «правильной» иерархии не существует — многое определяется задачей, опытом и вкусами программиста.

Простое наследование

Простым (или *одиночным*) называется наследование, при котором производный класс имеет только одного родителя. Мы уже видели пример простого наследования при знакомстве с иерархией исключений. Формально наследование одного класса от другого можно задать следующей конструкцией:

```
class имя_класса_потомка: [модификатор_доступа] имя_базового_класса  
{ тело_класса }
```

Класс-потомок наследует структуру (все элементы данных) и поведение (все методы) базового класса. Модификатор доступа определяет доступность элементов базового класса в классе-наследнике. Квадратные скобки не являются элементом синтаксиса, а просто показывают, что заключенный в них модификатор может отсутствовать. Этот модификатор мы будем называть *модификатором наследования*.

При разработке различных классов мы использовали два модификатора доступа к элементам класса: `public` (общий) и `private` (приватный). При наследовании используется еще один (см. п. 11 в [1]) — `protected` (защищенный). Зачем он понадобился? Рассмотрим пример из реальной жизни. Типичной иерархией наследования является семья, состоящая из нескольких поколений. Все члены семьи имеют общую фамилию и участвуют в общественной жизни (`public`). С другой стороны, у каждого из членов семьи есть свои личные (`private`) интимные тайны. Однако члены семьи являются родственниками и участвуют еще и в общей семейной жизни. Семейные тайны хранятся внутри семьи и недоступны обществу, но известны всем членам семьи. Именно «семейная жизнь» объектов потребовала наличия модификатора `protected`.

Возможны четыре варианта наследования: класс от класса, класс от структуры, структура от структуры и структура от класса. Доступность элементов базового класса из классов-наследников изменяется в зависимости от модификаторов доступа в базовом классе и модификатора наследования. Однако при любых вариантах модификатора наследования приватные элементы базового класса недоступны в наследниках. В табл. 8.1 приведены все варианты доступности элементов базового класса в производном классе при любых вариантах модификатора наследования.

Если в качестве модификатора наследования записано слово `public`, то такое наследование называется *открытым*. Соответственно, при использовании модификатора `protected` мы имеем *защищенное* наследование, а слово `private` означает *закрытое* наследование.

Структуры редко участвуют в иерархии наследования, обычно их используют вполне традиционным способом — для реализации кортежей. Кортеж — это небольшой фиксированный контейнер с элементами разного типа. Общепринятой практикой является правило — наследовать только от классов. Мы тоже будем стараться этому правилу следовать.

Таблица 8.1. Доступ к элементам базового класса в классе-наследнике

Модификатор доступа в базовом классе	Модификатор наследования	Доступ в производном классе	
		struct	class
public	Отсутствует	public	private
protected	Отсутствует	public	private
private	Отсутствует	Нет доступа	Нет доступа
public	public	public	public
protected	public	protected	protected
private	public	Нет доступа	Нет доступа
public	protected	protected	protected
protected	protected	protected	protected
private	protected	Нет доступа	Нет доступа
public	private	private	private
protected	private	private	private
private	private	Нет доступа	Нет доступа

Простое открытое наследование

Вернемся к разработке класса денег. Ранее мы реализовали простой класс, позволивший нам выполнять операции с рублями (см. главы 1 и 2). Однако практически в любой финансовой системе необходимо оперировать и другой валютой, например долларами или франками. Давайте рассмотрим мультивалютную систему, в которой используются рубли, доллары и евро. Понятно, что разные валюты обладают некоторыми общими свойствами, поэтому можно использовать механизм наследования. Однако так как мы пока не имеем опыта применения механизма наследования, нам, очевидно, наряду с разработкой классов-наследников потребуется время от времени модифицировать базовый класс.

Совершенно ясно, что арифметические операции с любой валютой — одни и те же: нам надо складывать и вычитать деньги, делить деньги на деньги, умножать и делить деньги на число. Даже операция ввода может быть унифицирована, так как практически для всех валют (по крайней мере, для долларов и евро) каждый «рубль» состоит ровно из 100 «копеек». Различия проявляются только при выводе денежной суммы на экран — необходимо указывать, в какой валюте эта денежная сумма представлена. Таким образом, мы можем реализовать базовый класс `TCurrency`, в котором определено поле для денежной суммы и реализованы все перечисленные операции, кроме операции вывода. Тогда конкретную валюту (рубли, доллары или евро) можно будет реализовать путем открытого наследования. Интерфейс этого класса представлен в листинге 8.1. Реализация не изменяется, за исключением названия класса, поэтому приводить мы ее не будем.

Листинг 8.1. Интерфейс базового класса TCurrency для работы с денежными суммами

```

class TCurrency
{ public:
    typedef unsigned int uint;
    TCurrency(const long double &t=0.0);           // конструктор
// методы
    TCurrency& operator+=(const TCurrency &b);
    TCurrency operator-() { Summa = -Summa; return *this; };
    TCurrency& operator++(){ return (*this+=1.0); }
    TCurrency operator++(int)
    { TCurrency t = *this; *this+=0.01; return t; }
    TCurrency operator-(const TCurrency &a);
    TCurrency& operator-=(const TCurrency &b);
    TCurrency& operator/=(const double &b);
    long double operator/(const TCurrency &b);
    TCurrency operator/(const double &b);
    TCurrency& operator*=(const double &b);
    bool isNegative() { return (Summa < 0); }
// дружественные функции
    friend TCurrency operator+(const TCurrency &a, const TCurrency &b);
    friend TCurrency operator*(const long double &a, const TCurrency &b);
    friend TCurrency operator*(const TCurrency &a, const long double &b);
    friend bool operator==(const TCurrency &a, const TCurrency &b);
    friend bool operator!=(const TCurrency &a, const TCurrency &b);
    friend bool operator<(const TCurrency &a, const TCurrency &b);
    friend bool operator>=(const TCurrency &a, const TCurrency &b);
    friend bool operator>(const TCurrency &a, const TCurrency &b);
    friend bool operator<=(const TCurrency &a, const TCurrency &b);
    friend ostream& operator>>(ostream& t, TCurrency &r);
private:
    long double Summa;
    string toString() const;
    long double round(const long double &r);
};

```

Разберемся, какие дополнения или изменения придется внести в классы-наследники. Во-первых, это, конечно, операция вывода на экран. При этом наша приватная функция перевода числа в символьный вид может быть реализована в базовом классе, так как она должна работать одинаково для трех валют. Напишем первый вариант наших классов-наследников (листинг 8.2).

Листинг 8.2. Классы-наследники от TCurrency

```

// Рубли
class Roubles: public TCurrency
{ public:
    friend ostream& operator<<(ostream& t, const Roubles &r);
};
ostream& operator<<(ostream& t, const Roubles &r)
{ string s = r.toString(); return (t << s << "p."); }
// Доллары
class Bucks: public TCurrency
{ public:
    friend ostream& operator<<(ostream& t, const Bucks &r);
};

```

Листинг 8.2 (продолжение)

```
ostream& operator<<(ostream& t, const Bucks &r)
{ string s = r.toString(); return (t << s << "$"); }
// Евро
class Euros: public TCurrency
{ public:
    friend ostream& operator<<(ostream& t, const Euros &r);
};
ostream& operator<<(ostream& t, const Euros &r)
{ string s = r.toString(); return (t << s << "evro"); }
```

Однако при трансляции тут же выясняется, что функция `toString()` недоступна в производных классах, так как в базовом классе она определена в приватной части. Поэтому необходимо создать в базовом классе секцию `protected` и перенести туда прототип функции `toString()`:

```
class TCurrency
{ public:
    typedef unsigned int uint;
    TCurrency(const long double &t=0.0);           // конструктор
// методы
// ...
// дружественные функции
// ...
protected:
    string toString() const;
private:
    // ...
};
```

Так как все три класса практически пока одинаковые, будем разбираться с нюансами открытого наследования, экспериментируя с рублями и модифицируя базовый класс, если это потребуется.

Конструкторы, деструкторы и наследование

В первую очередь разберемся с объявлениями новых объектов. Хотя в классе-наследнике отсутствуют конструкторы, мы тем не менее имеем возможность объявлять объекты без инициализации, а также объекты, инициализируемые другими объектами:

```
Roubles d; Roubles f = d;
```

Это означает, что и при наследовании для производного класса система по умолчанию создает конструктор без аргументов и конструктор копирования. Однако конструкторы не совсем «пустые»: поскольку класс `Roubles` является наследником от `TCurrency`, в этих конструкторах вызывается конструктор базового класса. В этом легко убедиться одним из двух способов¹:

- сделать конструктор базового класса приватным — программа просто перестанет транслироваться из-за недоступности конструктора;

¹ Третий способ — прочитать стандарт C++ (см. п. п. 12.6.2 в [1]).

- задать в конструкторе базового класса вывод сообщения на экран — тогда для всех объявлений объектов производного класса на экране появится сообщение, выдаваемое конструктором базового класса.

Такое поведение системы естественно: только конструктор базового класса «знает» все нюансы внутреннего устройства своего класса. Поэтому в создаваемых системой по умолчанию конструкторах он и вызывается для выполнения этой ответственной работы.

Отметим важнейшую принципиальную особенность наследования: конструкторы *не наследуются* производным классом, а *создаются* в производном классе (если не определены программистом явно). Система поступает с конструкторами следующим образом:

- если в базовом классе нет конструкторов или есть конструктор без аргументов (или аргументы присваиваются по умолчанию, как в нашем случае), то в производном классе конструктор можно не писать — будут созданы конструктор копирования и конструктор без аргументов;
- если в базовом классе все конструкторы с аргументами, то производный класс должен иметь конструктор, в котором явно вызывается конструктор базового класса.

Если мы уберем в конструкторе класса `TCurrency` значение по умолчанию, то тут же получим сообщение об ошибке для показанных ранее объявлений переменных-рублей: в базовом классе все конструкторы с аргументами, а в производном классе конструкторы отсутствуют. Поэтому напомним конструктор инициализации для класса `Roubles` (листинг 8.3).

Листинг 8.3. Конструктор инициализации для класса `Roubles`

```
Roubles(const long double &r=0.0)
:TCurrency(r)                // явный вызов базового конструктора
{ };
```

Конструктор базового класса явно вызывается в списке инициализации. Значение по умолчанию присваивать необходимо. Если этого не сделать, тогда нужно реализовать конструктор без аргументов, чтобы показанные объявления не вызвали ошибки.

Рассмотрим еще один простой пример наследования (листинг 8.4):

Точка на плоскости Точка в пространстве

Листинг 8.4. Конструкторы при наследовании

```
class Point2D                                // точка на плоскости
{ public:
    // Point2D():x(0.0), y(0.0){}            // конструктор без аргументов
    Point2D(double x, double y):x(x), y(y) {} // конструктор инициализации
    double getx() const { return x; }        // координата x
    double gety() const { return y; }        // координата y
private:
    double x,y;
```


Листинг 8.4 (продолжение)

```

};
class Point3D: public Point2D           // точка в пространстве
{
    public:
        Point3D(double x, double y, double z) // конструктор инициализации
            :Point2D(x,y),z(z)
        { }
        double getz() const { return z; }     // координата z
    private:
        double z;
};

```

Мы определили класс с именем `Point2D`, в котором реализовали конструктор инициализации. В производном классе с именем `Point3D` должен быть конструктор инициализации, в котором конструктор базового класса вызывается явным образом в списке инициализации.

Если мы определим поля в базовом классе как защищенные (`protected`), то вполне допустима инициализация полей в теле конструктора класса-наследника, например:

```

Point3D(double x, double y, double z):z(z)
{ this->x = x; this->y = y; }

```

Ни в базовом классе, ни в классе-наследнике не определен конструктор без аргументов. К тому же он не создается системой автоматически, так как определен конструктор инициализации. Это приводит к тому, что следующие объявления сопровождаются сообщениями об ошибках трансляции:

```

Point2D a;
Point3D b;

```

Если мы раскомментируем конструктор без аргументов в базовом классе, то создать объект базового класса без инициализации будет можно, а объект производного класса — нельзя:

```

Point2D a;           // работает
Point3D b;           // по-прежнему не работает

```

Это и понятно — конструкторы не наследуются, а так как в производном классе определен конструктор инициализации, то конструктор без аргументов не создается. Осталось разобраться с деструкторами, а также выяснить порядок вызова конструкторов и деструкторов:

- ❑ при создании объекта производного класса сначала вызывается конструктор базового класса, затем — производного;
- ❑ деструкторы, как и конструкторы, не наследуются, однако при отсутствии деструктора в производном классе система формирует деструктор по умолчанию;
- ❑ деструктор базового класса вызывается в деструкторе производного класса автоматически (см. п. п. 12.4/6 в [1]) независимо от того, определен он явно или создан системой;
- ❑ деструкторы вызываются в порядке, обратном порядку вызова конструкторов.

Простой пример, текст которого представлен в листинге 8.5, демонстрирует эти положения стандарта.

Листинг 8.5. Порядок вызова конструкторов и деструкторов

```
class Base                                // базовый класс
{ public:
    Base() { cout << "Base" << endl; }
    ~Base() { cout << "~Base" << endl; }
};
class Derived: public Base                // класс-наследник
{ public:
    Derived() { cout << "Derived" << endl; }
    ~Derived() { cout << "~Derived" << endl; }
};
void f(void)
{   Derived a;                           // создали объект производного класса
}                                         // объект разрушен
int main()
{   f();
    return 0;
}
```

Выполнив эту программу, получим на экране:

```
Base
Derived
~Derived
~Base
```

Таким образом, создание и уничтожение объектов выполняется по принципу LIFO: «последним создан — первым уничтожен». Проверить, создается ли деструктор по умолчанию, в котором вызывается деструктор базового класса, тоже очень просто — достаточно закомментировать (или удалить) в классе-наследнике определение деструктора и снова выполнить программу. Вывод деструктора производного класса, естественно, исчезнет, но мы увидим, что деструктор базового класса все-таки вызывается, несмотря на то, что объектов базового типа мы не объявляли.

Поля и методы при наследовании

Как уже отмечалось, класс-наследник получает в наследство все поля базового класса (хотя, если они были приватные, доступа к ним он не имеет). В этом легко убедиться, выведя на экран размеры базового класса `TCurrency` и класса-наследника `Roubles`:

```
cout << sizeof(TCurrency) << endl;
cout << sizeof(Roubles) << endl;
```

Так как мы не добавляли полей в производный класс, то на экран в обоих случаях будет выдано одно и то же число — размер поля `Summa`. В классе `Point3D` мы добавили поле `z`, поэтому размер класса увеличился по сравнению с классом `Point2D` на размер типа `double`.

Добавляемые поля в наследнике могут совпадать и по имени, и по типу с полями базового класса. Конечно, специально так делать не следует, но это не запрещено стандартом, поэтому нужно разобраться, как обращаться к полям с одинаковыми именами. Модифицируем наш пример из листинга 8.5, добавив в базовый и производный классы поля с одинаковыми типами и именами (листинг 8.6).

Листинг 8.6. Одинаковые поля в базовом и производном классах

```
class Base
{ protected:
    int x;
public:
    Base(int x):x(x) { cout << "Base" << endl; }
    ~Base() { cout << "~Base" << endl; }
};
class Derived: public Base
{ int x;
public:
    Derived(int x):x(x),Base(x+1) { cout << "Derived" << endl; }
    ~Derived() { cout << "~Derived" << endl; }
    int f() { return (Base::x+x); }
};
```

В базовом классе мы объявили поле в защищенной секции, чтобы класс-наследник мог непосредственно к нему обращаться. Естественно, в производном классе новое поле скрывает поле базового класса, поэтому для доступа к полю базового класса необходимо использовать квалификатор класса, что и показано в методе `f()`.

Класс-потомок наследует все поведение базового класса. Это означает, что нет необходимости заново определять операции в классе-наследнике, если их поведение не изменяется по сравнению с поведением в базовом классе. Снова обратимся к классу `Roubles`, наследнику класса `TCurrency`, и проверим работу операций, выполнив простую программу, представленную в листинге 8.7.

Листинг 8.7. Проверка работы операций в классе-наследнике

```
int main()
{
    Roubles d;      cout << d << endl;          // d = 0.0
    ++d; d++;      cout << d << endl;          // d = 1.01
    d+=4.02;      cout << d << endl;          // d = 5.03
    return 0;
}
```

В примере мы не стали проверять все методы, определенные в базовом классе, — и так понятно, что все они правильно работают с объектами класса-наследника, хотя определены в базовом классе.

Естественно, в классе-наследнике допустимо определять новые методы, что можно видеть в листинге 8.4: в производном классе `Point3D` определен метод `getz()`. Во вновь определяемых методах разрешается вызывать любые доступные методы базового класса. Если в классе-наследнике имя метода совпадает с именем

метода базового класса, то метод производного класса скрывает все методы базового класса с таким именем. При этом прототипы могут не совпадать.

Таким образом, чтобы в методе-наследнике вызвать одноименный метод родительского класса, нужно задавать его с квалификатором класса. Добавим в класс `Point2D` метод `Print()` для вывода на экран координат точки:

```
void Print() const
{ cout << x << ', ' << y; }
```

В производном классе `Point3D` определим метод с тем же именем. Чтобы вывести координаты x и y , вызовем одноименный метод базового класса:

```
void Print() const
{ Base::Print(); cout << ', ' << z; }
```

Аналогично можно выполнять метод базового класса для объекта-наследника, например:

```
Poinr3D T;
T.Print();           // вызов "родного" трехмерного метода
T.Base::Print();     // вызов "родительского" двухмерного метода
```

Вложенные классы и наследование

Поскольку в C++ вложенные классы разрешены (см. п. 9.7 в [1]), нужно выяснить, могут ли они участвовать в наследовании. Собственно, возникают три вопроса:

1. Может ли внешний класс наследовать от вложенного класса?
2. Может ли вложенный класс наследовать от внешнего класса?
3. Может ли вложенный класс наследовать от другого вложенного класса?

Ответ на все три вопроса — да! Простой пример демонстрирует это (листинг 8.8).

Листинг 8.8. Наследование и вложенные классы

```
class A {};           // внешний класс
class B
{ public:
    class C: public A // вложенный класс наследует от внешнего
    {};
};
class D: public B::C // внешний класс наследует от вложенного
{};
class E
{ class F: public B::C // вложенный класс наследует от вложенного
  {};
};
```

Таким образом, ограничений в наследовании вложенных классов нет. Единственное, за чем нужно следить, — видимость базового класса в точке наследования. Например, от вложенного класса `E::F` «снаружи» наследовать не получится, так как класс находится в приватной части класса `E` и невидим вне его. Однако

вполне допустимо, чтобы еще один вложенный класс внутри E наследовал от вложенного класса E::F, например:

```
class E
{ class F: public B::C
  {}
  class G: public F
  {}
};
```

Принцип подстановки

Помимо конструкторов, не наследуются еще два вида функций: операция присваивания и дружественные функции. Последнее понятно: дружественные функции не входят в состав методов класса, а являются внешними, хотя и имеют доступ к внутренней структуре класса. Ситуация не очень приятная, так как нам придется дублировать дружественные функции для каждого класса-наследника TCurrency. Например, функция-операция сложения для класса Roubles выглядит совершенно так же, как функция сложения для класса TCurrency:

```
Roubles operator+(const Roubles &a, const Roubles &b)
{ Roubles t = a; t+=b;
  return t;
}
```

Очевидно, что при наличии большого количества дружественных функций класс не слишком пригоден для наследования.

Однако оказывается, что при открытом наследовании можно не дублировать дружественные функции, если правильно воспользоваться *принципом подстановки*¹. Открытое наследование устанавливает между классами отношение «является»: класс-наследник *является* разновидностью класса-родителя. И это не просто словесная формулировка — она непосредственно поддерживается компилятором. Когда мы пишем, что класс Derived (производный) открыто наследует от класса Base (базовый), мы сообщаем компилятору, что каждый объект типа Derived является также объектом класса Base. С практической точки зрения это означает, что везде, где может быть использован объект типа Base, вместо него разрешается подставлять объект типа Derived — это и есть принцип подстановки. Этот принцип работает и для ссылок, и для указателей: вместо ссылки (указателя) на базовый-класс может быть подставлена ссылка (указатель) на класс-наследник. Преобразование типа при этом указывать не требуется — оно выполняется автоматически. В [11] такое приведение типов названо *повышающим*.

Обратное — неверно! Например, будильник является часами, но не всякие часы — будильник. Здесь часы — базовый класс, а будильник — производный. В иерархии стандартных исключений (см. раздел «Стандартные исключения» в главе 7)

¹ Впервые этот принцип сформулировала Барбара Лисков, поэтому иногда его называют принципом подстановки Лисков (Liscov's substitution principle, LSP).

наблюдаем ту же картину: любой класс-потомок класса `exception` также является исключением `exception`, но обратное — неверно. Например, исключение `bad_alloc` является одним из видов исключения `exception`, но `exception` не является исключением `bad_alloc`. Именно принцип подстановки работал при обработке исключений в программе, представленной в листинге 7.16.

Таким образом, теоретически мы можем не определять дружественные функции для производного класса, так как можно использовать функции, реализованные для базового класса — любой объект типа `Roubles` является одновременно и объектом типа `TCurrency`. На практике — не все так просто. Если мы попытаемся выполнить следующий фрагмент, то получим сообщение об ошибке трансляции:

```
Roubles d(15);
cout << d+d << endl;
```

Причина — отсутствие определения операции `operator<<` с параметром типа `TCurrency`. С одной стороны, операция сложения рублей не вызвала протестов компилятора, так как он обнаружил определение этой операции с аргументами базового типа и воспользовался принципом подстановки. С другой стороны, результат операции сложения тоже имеет тип `TCurrency`, а в базовом классе операция вывода `operator<<` не определена. Выполнять преобразование из базового типа в наследуемый компилятор не умеет, поэтому не может воспользоваться определением операции вывода для рублей.

Эту проблему можно решить с помощью *виртуальных функций* (см. следующую главу), но сначала попробуем «зайти с другого боку»:

```
Roubles d(15), f;
f = d+d;           // ошибка трансляции!
cout << f << endl;
```

Как бы не так! Компилятор ничего не имеет против вывода рублей `f`, но он категорически «возражает» против присваивания значения типа `TCurrency` (результат сложения `d + d`) переменной типа `Roubles`.

Разберемся теперь с операцией присваивания. Так как в базовом классе операция `operator=` не определена, компилятор создает ее по умолчанию. Как мы уже знаем (см. п. п. 12.8/10 в [1]), такая операция для базового класса имеет прототип:

```
TCurrency& operator=(const TCurrency &);
```

Для класса-наследника создается аналогичная функция:

```
Roubles& operator=(const Roubles &);
```

Наличие этих функций позволяет нам выполнять следующие присваивания:

```
TCurrency c1(10), c2(11); // переменные базового класса
Roubles r1(20), r2(21);  // переменные производного класса
c1 = c2;                  // операция базового класса
r1 = r2;                  // операция производного класса
c1 = r2;                  // базовый = производный; операция базового класса
```

В последнем случае работает принцип подстановки: справа от присваивания задан объект производного класса, который подставляется на место объекта базового класса в операции присваивания.

Однако мы не можем выполнить присваивание

```
r1 = c1; // производный = базовый
```

Неявное преобразование типа не работает, никакие явные преобразования не помогают. Такое присваивание не проходит и для указателей (ссылок), например:

```
TCurrency c1(10); // переменная базового класса
Roubles *pr; // указатель производного класса
pr = &c1; // ошибка трансляции
```

Компилятор сообщает, что он не может преобразовать адрес объекта базового класса в адрес производного. Чтобы это присваивание работало, требуется определить операцию присваивания с прототипом

```
Roubles& operator=(const TCurrency &);
```

Стандарт не запрещает писать такие операции присваивания. Более того, в любом классе можно неоднократно перегрузить операцию присваивания, в одной из которых, например, ни аргумент, ни результат не являются определяемым классом (см. п. п. 12.8/9 в [1]). Единственным ограничением является видимость определения нужных классов в точке определения операции присваивания.

В данном случае мы фактически должны написать функцию, аналогичную функции преобразования типа из базового типа в производный. Такое приведение в [11] названо *понижающим*.

ВНИМАНИЕ

В C++ понижающее приведение можно выполнить с помощью оператора преобразования `dynamic_cast` (см. п. п. 10.3/1 в [1]), которое работает для полиморфных классов. Полиморфный класс — это класс с виртуальными функциями.

Именно из-за отсутствия такой операции не работал оператор

```
f = d+d;
```

Рассмотрим, что должна делать эта функция и какие при этом возникают проблемы. В базовом классе такую функцию определить невозможно, так как базовый класс «понятия не имеет» о своих наследниках. Попытка ее создания немедленно приводит к ошибке трансляции класса `TCurrency`, так как `Roubles` не является определенным к этому моменту типом. Не спасает и предварительное объявление класса:

```
class Roubles;
```

Так как в этом случае возникает другая ошибка — отсутствие определения класса. Следовательно, нужно определить требуемую функцию в производном классе. Но и в этом случае некоторые проблемы остаются. Во-первых, мы тогда должны переопределять эту функцию в каждом классе-наследнике (так как присваивание не наследуется, как и конструкторы с деструкторами). Во-вторых, что важнее, наша функция должна присвоить поля базового класса соответствующим полям производного класса, однако непосредственно она не может этого сделать, так как поля базового класса недоступны — они приватные.

«А давайте..!» — нет, открывать закрытые поля мы не будем. Не для того городился огород с инкапсуляцией, чтобы вот так просто его разрушить. Даже перевод полей в защищенные открывает «ящик Пандоры» — любой наследник получает к ним доступ. Вы только подумайте: достаточно унаследовать от `TCurrency` — и можно делать с суммами все, что заблагорассудится! Это — тот самый путь, который ведет в Ад.

Решение этой проблемы можно найти в стандарте (см. п. п. 12/1 в [1]):

```
Roubles& Roubles::operator=(const TCurrency &t)
{ this->TCurrency::operator=(t);          // вызов родительской операции
  return *this;
}
```

Собственно, самая главная «фишка» — это вызов родительской операции в дочерней, причем в функциональной форме. Это работает, несмотря на то, что в родительском классе операция присваивания не была явно определена.

При наследовании денег не просматривается еще одна проблема, связанная с принципом подстановки. Чтобы понять, в чем дело, обратимся к классам точек и рассмотрим простой пример с двухмерными и трехмерными точками.

```
Point3D b(1,2,3);
Point2D a = b;          // подстановка в конструкторе копирования - срезка
a = b;                  // подстановка в присваивании - срезка
```

Работает принцип подстановки, однако нас поджидает неприятность: так как базовый класс ничего не знает о своих наследниках, то в переменную `a` копируется только двухмерная (`Point2D`) часть трехмерной точки. Этот эффект называется *срезкой* [2], или *расщеплением* [11]; он частенько приводит к ошибкам. Например, при передаче параметра по значению, как мы знаем, работает конструктор копирования, поэтому в таких случаях тоже может произойти срезка. При передаче параметра по ссылке (или по указателю) ничего подобного не происходит. При передаче параметров в блок обработки исключения тоже может произойти срезка, поэтому параметр в секцию-ловушку лучше передавать по ссылке.

Применение открытого наследования

Ранее мы реализовали простой стек, элементами которого были указатели `void *` (см. листинг 6.9). Во-первых, нетипизированные указатели могут быть источником ошибок, так как преобразования в тип `void *` делаются «молча». В стек, предназначенный, например, для хранения указателей на числа, нечаянно может попасть указатель на строку. Хуже того, в стек может попасть указатель на локальную переменную, которую и уничтожать не требуется. Во-вторых, очевидный недостаток такой универсальности — необходимость явного приведения типа при каждой выборке указателя из контейнера. Мы можем избавиться от явного приведения типа за счет наследования. Например, мы можем организовать стек с указателями на `double` (листинг 8.9).

Листинг 8.9. Применение наследования для специализации указателей стека

```

class doubleStack: public TStack
{
public:
    void push(double *s)           // положить в стек
    { TStack::push(s); }
    double *top() const            // выдать элемент с вершины
    { return (double *)TStack::top(); }
    double *pop()                  // удалить с вершины
    { return (double *)TStack::pop(); }
};

```

Мы упрятали (инкапсуляция — в действии) явное преобразование в методы. Использовать стек столь же просто, и при этом нет нужды выполнять преобразование указателей:

```

doubleStack t;
t.push(new double(11));           // добавляем
t.push(new double(22));           // элементы
t.push(new double(33));           // в стек
while (!t.empty())
{ cout << *(t.top()) << endl;     // без преобразования
  double *p = t.pop();             // без преобразования
  delete p;
}

```

Программа выведет на экран

```

33
22
11

```

Заменяв в листинге 8.9 слово `double` словом `string`, получим стек с указателями на строки (листинг 8.10).

Листинг 8.10. Применение наследования для специализации указателей стека

```

class StringStack: public TStack
{
public:
    void push(string *s) { TStack::push(s); }
    string *top() const { return (string *)TStack::top(); }
    string *pop() { return (string *)TStack::pop(); }
};

```

Комментарии излишни. Использование такого стека тоже не отличается сложностью:

```

StringStack s;
s.push(new string("111"));
s.push(new string("222"));
s.push(new string("333"));
string *ss;
while ((ss=s.pop())!=0)           // без преобразования
{ cout << *ss << endl;           // без преобразования
  delete ss;
}

```

Благодаря наследованию мы можем реализовать стек не указателей, а самих чисел типа `double` (листинг 8.11).

Листинг 8.11. Реализация стека без указателей

```
class StackDouble: public TStack
{
public:
    void push(const double& d)
    { double *p = new double(d);           // организовали указатель
      TStack::push(p);                     // поместили указатель
    }
    double top() const
    { double *p = (double *)TStack::top();
      return *p;
    }
    double pop()
    { double *p = (double *)TStack::pop();  // получили указатель
      double t = *p;                       // сохранили значение
      delete p;                            // возвратили память
      return t;                            // возврат значения
    }
};
```

Мы упрятали внутрь методов не только преобразование, но и указатели. Теперь метод `push()` сам организует указатель, помещаемый в стек, а метод `pop()` возвращает память системе. Работа с таким стеком для пользователя стала проще:

```
StackDouble t;
t.push(11/2.0);
t.push(22);
t.push(33);
while (!t.empty())
{ double p = t.pop();
  cout << p << endl;
}
```

Пользователь этого стека и не подозревает, что работает с указателями.

Закрытое наследование

Для тех же целей мы можем применить закрытое (или защищенное) наследование (листинг 8.12).

Листинг 8.12. Специализация стека путем закрытого наследования

```
class StringStack: private TStack
{
public:
    void push(string *s) { TStack::push(s); }
    string *top() const   { return (string *)TStack::top(); }
    string *pop() { return (string *)TStack::pop(); }
    bool empty() const { return TStack::empty(); }
};
```

Проверка выполняется точно так же, как и в примере с открытым наследованием (см. листинг 8.10).

Чисто внешне разница заключается только в том, что мы заменили одно слово другим. Кроме того, в явном виде реализован метод `empty()`, чего мы не делали

при открытом наследовании. Дело в том, что при закрытом наследовании все элементы класса-наследника становятся приватными и недоступными клиенту. Поэтому мы должны в наследнике либо реализовать нужные нам методы, либо открыть методы базового класса с помощью объявления `using` (см. п. п. 7.3.3 в [1]). Обычное его применение связано с использованием *пространств имен*, однако и в классах такое объявление позволяет разрешить проблемы видимости. Синтаксис объявления `using` очень прост:

```
using <имя базового класса>::<имя в базовом классе>;
```

Например, мы можем реализовать стек для чисел типа `double`, закрыто унаследовав от класса `TDeque`. Тогда `<имя базового класса>` — это имя `TDeque`, а `<имя в базовом классе>` — это имя открываемого метода. Если в качестве вершины стека мы выберем начало дека, то реализация может быть такой, как показано в листинге 8.13.

Листинг 8.13. Использование закрытого наследования для реализации стека

```
class TStack: private TDeque
{ public:
    using TDeque::isEmpty;           // проверка отсутствия элементов
    using TDeque::push_front;        // поместить элемент на вершину
    using TDeque::pop_front;         // удалить элемент с вершины
    using TDeque::begin;             // получить доступ к элементу
    using TDeque::size;              // количество элементов в стеке
    using TDeque::iterator;          // вообще-то обычно не нужен
};
```

В данном случае закрытое наследование позволяет нам ограничить в наследнике предоставляемую функциональность. Однако мы вынуждены открывать итератор, так как в базовом классе `TDeque` значение элемента предоставляет операция разыменования итератора. В частности, метод `begin()` выдает итератор. Можно, как мы делали ранее, упрятать всю работу с итераторами в явно реализованный метод `top()`.

Использовать такой стек так же просто, как и реализованный непосредственно:

```
TStack S;
TStack::iterator is;
S.push_front(1);
S.push_front(2);
S.push_front(3);
S.push_front(4);
S.push_front(5);
while (!S.isEmpty())
{ is = S.begin(); cout << *is << ' '; S.pop_front(); }
```

На экране появится строка

```
5 4 3 2 1
```

Вспомним, что операция присваивания создается в любом классе, если не определена явно. Таким образом, при наследовании операция присваивания, создаваемая по умолчанию в производном классе, скрывает операцию присваивания

базового класса. Однако используя объявление `using`, мы можем открыть операцию базового класса в производном классе — это делается так:

```
using Base::operator=;
```

Закрытое наследование принципиально отличается от открытого в одной важной детали. При открытом наследовании выполняется принцип подстановки Лисков, и это означает, что *класс является разновидностью класса*. Закрытое наследование означает совсем другое: *класс реализован посредством класса*. Говорят, что закрытое наследование — это *наследование реализации*. Принцип подстановки при закрытом наследовании *не выполняется*. Это означает, что мы не сможем присвоить (во всяком случае, без явного преобразования типа) объект производного класса базовому. Поэтому закрытое наследование хорошо применять в тех случаях, когда мы хотим иметь функциональность базового класса, но нам не нужны копирование и присваивание. Для стеков закрытое наследование как раз подойдет, так как стеки обычно не требуется присваивать.

Подобное применение закрытого наследования — это один из вариантов реализации паттерна Adapter (адаптер) [17]. Этот паттерн применяется для того, чтобы использовать функциональность некоторого класса, но в новом интерфейсе. Мы адаптировали класс `TDeque` к новым потребностям.

Мы можем специализировать стек и без наследования. Обычно это делается с помощью композиции — снова реализуем стек указателей на тип `double` (листинг 8.14).

Листинг 8.14. Композиция для специализации стека

```
class DoubleStack
{
    TStack stack; // объект-универсальный стэк
public:
    void push(double *s) // поместить в стек
    { stack.push(s); }
    double *top() const // выдать элемент с вершины
    { return (double *)stack.top(); }
    double *pop() // удалить элемент с вершины
    { return (double *)stack.pop(); }
    bool empty() const // пустой ли стек?
    { return stack.empty(); }
};
```

Это решение¹ обладает полной функциональностью стека из листинга 8.9, но не требует наследования. Кроме того, реализованные таким образом стеки с разной специализацией, очевидно, не являются родственниками, поэтому их нельзя присвоить один другому — и без закрытого наследования. У этого решения только одна особенность, которую можно посчитать недостатком: размер «композитного» стека больше, чем размер наследуемого, так как появилось поле данных. Однако объект базового стека имеет размер указателя, так что в данном случае это несущественно. Если вас это волнует, то всегда можно включить в класс не объект, а указатель.

¹ Такое применение композиции — один из вариантов паттерна Adapter.

Как видим, в C++ можно одну задачу решать разными способами — все зависит от задачи, знаний и предпочтений программиста. Далее мы увидим, что проблема универсальности контейнера имеет еще не одно решение.

Но вернемся к закрытому наследованию. У Герба Саттера в [21] можно обнаружить еще один вариант применения закрытого наследования: ограничение принципа подстановки (см. задачу 3.11 на с. 215). Вернее, задача сформулирована следующим образом: как разрешить принцип подстановки только для заданной функции, запретив его для всех остальных.

Сначала рассмотрим простой пример с часами и будильником, в котором используется принцип подстановки (листинг 8.15).

Листинг 8.15. Неограниченный принцип подстановки

```
class Clock                      // базовый класс — часы
{ public:
    void print() const { cout << "Clock!" << endl; }
};
class Alarm: public Clock       // производный класс — будильник
{ public:
    void print() const          // переопределенный метод
    { cout << "Alarm!" << endl; }
};
void settime(Clock &d)           // функция установки времени
{ d.print(); }
void f1()
{ Alarm a;
    settime(a);                 // это надо разрешить
}
void f2()
{ Alarm a;
    settime(a);                 // это требуется запретить
}
```

Проблема состоит в том, чтобы в функции `f1()` во время вызова функции установки времени принцип подстановки сработал, а в аналогичной ситуации в функции `f2()` — нет. Задача решается за счет закрытого наследования и механизма друзей (листинг 8.16).

Листинг 8.16. Ограничение выполнения принципа подстановки

```
class Clock
{ public:
    void print() const { cout << "Clock!" << endl; }
};
class Alarm: private Clock      // закрываем доступ всем
{ public:
    void print() const { cout << "Alarm!" << endl; }
    friend void f1();           // открываем доступ только f1()
};
void settime(Clock &d)
{ d.print(); }
void f1()
```

```
{ Alarm a;
  setttime(a);                // нормально
}
void f2()
{ Alarm a;
  setttime(a);                // ошибка трансляции
}
```

В этом примере мы путем закрытого наследования закрыли доступ к «внутренностям» класса `Alarm`, но объявили функцию `f1()` другом. Как указано Гербом Саттером в [21], «код с доступом к закрытым частям `Derived` может полиморфно использовать объекты `Derived` вместо объектов `Base`». Такой доступ имеют методы и функции-друзья.

Резюме

Класс может наследовать поля и методы другого класса. Класс, от которого наследуют, называется базовым классом. Наследующий класс — это производный класс. Производный класс наследует все поля и методы класса-родителя. Не наследуются операция присваивания, конструкторы и деструктор. При наследовании используется еще один модификатор доступа — `protected` (защищенный), который ограничивает видимость элементов класса. Защищенные элементы видны только прямым наследникам класса.

В производном классе можно определить новые поля и методы, перегрузить или переопределить унаследованные методы. Если в производном классе возникает необходимость использовать одноименный элемент базового класса, требуется задавать префикс — имя базового класса.

Наследование бывает простое и множественное, открытое и закрытое. Если базовый класс — единственный, то наследование простое. Если базовых классов несколько, то наследование множественное. Открытое наследование устанавливает между базовым и производным классом отношение «является»: объект производного класса является разновидностью объекта базового класса. Это положение называется принципом подстановки и поддерживается компилятором. При подстановке объекта производного класса на место объекта базового класса происходит неявное преобразование типа. При этом может произойти расщепление объекта производного типа.

Закрытое наследование устанавливает отношение «реализован»: класс реализуется посредством класса. То же отношение устанавливается при композиции, когда объект одного класса включается в качестве поля данных в другой класс.

Контрольные вопросы

1. Какие две роли выполняет наследование?
2. Какие виды наследования возможны в C++?
3. Чем отличается модификатор доступа `protected` от модификаторов `private` и `public`?

4. Чем открытое наследование отличается от закрытого и защищенного?
5. Может ли структура наследовать от класса? А класс от структуры?
6. Какой тип наследования от структуры реализуется по умолчанию? А от класса?
7. В каких случаях в классе-наследнике недоступны элементы базового класса?
8. Какие функции не наследуются?
9. Сформулируйте правила написания конструкторов в производном классе.
10. Каков порядок вызова конструкторов? А деструкторов?
11. Если имя нового поля совпадает с именем унаследованного, то каким образом разрешить конфликт имен?
12. Каким образом в конструкторе-наследнике вызвать конструктор базового класса?
13. Что происходит, если имя метода-наследника совпадает с именем базового метода?
14. Каким образом в операции присваивания класса-наследника вызвать операцию присваивания базового класса?
15. Может ли вложенный класс наследовать от внешнего? А внешний от вложенного?
16. Сформулируйте принцип подстановки.
17. Когда выполняется понижающее приведение типов? А повышающее?
18. Объясните, что такое «срезка», или «расщепление».
19. В каких случаях используется объявление `using`?
20. Какой вид наследования «ближе» к композиции: открытое или закрытое?

Упражнения

1. Создать класс `Pair` (пара целых чисел); определить метод умножения на число и операцию сложения пар $(a,b) + (c,d) = (a + b, c + d)$. Определить класс-наследник `Money` с полями: рубли и копейки. Переопределить операцию сложения и определить методы вычитания и деления денежных сумм.
2. Определить класс `Pair` с полями типа `double`. Реализовать операции сложения пар и умножения на число как в упражнении 1. Определить производный класс `Complex` и реализовать методы умножения $(a, b) \cdot (c, d) = (ac - bd, ad + bc)$ и вычитания $(a, b) - (c, d) = (a - b, c - d)$.
3. Реализовать базовый класс-оболочку `Number` для числового типа `double` с набором арифметических операций, а также операций возведения в произвольную степень и сравнения. Реализовать класс-наследник с операциями получения обратной величины, вычисления произвольного корня, вычисления натурального, десятичного и двоичного логарифмов, вычисления произвольной показательной степени.
4. Создать класс `Triad` (тройка целых чисел); определить методы увеличения полей на 1. Определить производный класс `Date` с полями: год, месяц и день.

Переопределить методы увеличения полей на 1 и определить метод увеличения даты на n дней.

5. Использовать класс `BitString` (см. упражнение 9 в главе 2) в качестве базового. Реализовать класс-наследник `AdvancedBitString`, осуществив в классе-наследнике перегрузку операций. Определить методы сдвига влево и вправо на заданное количество битов.
6. Создать базовый класс `Array` — динамический массив типа `unsigned char` (см. главу 5). Реализовать конструктор инициализации, задающий количество элементов и начальное значение (по умолчанию — 0). Реализовать методы доступа к отдельному элементу, перегрузив операцию индексирования `operator[]`. При этом должна выполняться проверка индекса на допустимость. Определить класс `Decimal` (см. упражнение 5 в главе 5) как класс-наследник от класса `Array`.
7. Используя в качестве образца класс `TDeque`, разработанный в главе 6, создать базовый класс `TQueue`, реализующий очередь. Элементом очереди является структура с полями: фамилия, количество членов семьи, год постановки в очередь. Реализовать класс-наследник, в котором определить операцию сортировки очереди по году постановки в очередь.
8. Используя класс `TQueue` (см. предыдущее упражнение), реализовать класс-наследник `Set`, добавив операции объединения и пересечения.
9. Реализовать очередь из упражнения 7 как наследник, используя класс `TStack` (см. листинг 6.9) в качестве образца для базового класса. Применить закрытое наследование.
10. Реализовать класс `ListPerson` (см. упражнение 10 в главе 4) как наследник от базового класса `List`, аналогичного `TDeque`.
11. Создать базовый класс `Iterator` для контейнера с элементами типа `double`. Реализовать класс-список, в котором итератор является наследником класса `Iterator`.
12. Определить класс `Object` с подсчетом объектов (см. листинг 4.23) как базовый, а класс `Triad` (см. упражнение 4) как наследник от него. Реализовать класс-наследник `Date`.

Глава 9

Виртуальные функции

Обычная эволюция начинающего программиста, пишущего на C++, проходит в несколько этапов. Сначала осваиваются процедурные возможности C++. На втором этапе программист начинает применять классы для реализации новых типов. До этого уровня добраться легко, особых умственных усилий не требуется. Конструкция класса применяется примерно так же, как и конструкция функции в процедурном программировании: язык расширяется, но не просто новой функциональностью, а новым типом данных, например комплексными числами или датами. Такие новые типы данных Коплиен в [26] и Страуструп в [2] называют «конкретными». Параллельно программист осваивает принцип инкапсуляции и, как он думает, принцип полиморфизма, реализуя перегрузку операций. Следующим шагом в освоении C++ обычно становится реализация динамических структур данных — контейнеров. Освоение этого уровня требует хорошего понимания работы базовых методов класса: конструкторов, деструкторов и присваивания. На этом этапе программист изучает реализацию присваивания и операции индексирования, усваивая смысл понятий левостороннего (l-value) и правостороннего (r-value) значений.

Наконец, программист «добирается» до наследования и осваивает очень важный принцип подстановки. Вроде бы, больше изучать нечего: освоены все три «кита» объектно-ориентированного программирования. Но оказывается, только после этого начинается «настоящее» объектно-ориентированное программирование с использованием виртуальных функций. Несомненно, концепция виртуальных функций — одна из наиболее сложных концепций C++, которую новичкам усвоить непросто. Однако без понимания этого механизма программировать на C++ практически невозможно. Поэтому вооружитесь терпением и приступайте.

Зачем нужны виртуальные функции

При наследовании часто бывает необходимо, чтобы поведение некоторых методов базового класса и классов-наследников различалось. Решение на первый взгляд очевидное: переопределить соответствующие методы в производном классе.

Однако тут возникает одна проблема, которую лучше рассмотреть на простом примере (листинг 9.1).

Листинг 9.1. Необходимость виртуальных функций

```
#include <iostream>
using namespace std;
class Base                                // базовый класс
{ public:
    int f(const int &d)                    // метод базового класса
    { return 2*d; }
    int CallFunction(const int &d)        // предполагается
    { return f(d)+1; }                    // вызов метода базового класса
};
class Derived: public Base                // производный класс
{ public:
    int f(const int &d)                    // CallFunction наследуется
    { return d*d; }                       // метод f переопределяется
};
int main()
{   Base a;                               // объект базового класса
    cout << a.CallFunction(5)<< endl;     // получаем 11
    Derived b;                             // объект производного класса
    cout << b.CallFunction(5)<< endl;     // какой метод f вызывается?
    return 0;
}
```

В базовом классе определены два метода — `f()` и `CallFunction()`, причем во втором методе вызывается первый. В классе-наследнике метод `f()` переопределен, а метод `CallFunction()` унаследован. Очевидно, метод `f()` переопределяется для того, чтобы объекты базового класса и класса-наследника вели себя по-разному. Объявляя объект `b` типа `Derived`, программист, естественно, ожидает получить результат $5 \cdot 5 + 1 = 26$ — для этого и переопределялся метод `f()`. Однако на экран, как и для объекта `a` типа `Base`, выводится число 11, которое, очевидно, вычисляется как $2 \cdot 5 + 1 = 11$. Несмотря на переопределение метода `f()` в классе-наследнике, в унаследованной функции `CallFunction()` вызывается «родная» функция `f()`, определенная в базовом классе!

Аналогичная проблема возникает и в несколько другом контексте: при подстановке ссылки или указателя на объект производного класса вместо ссылки или указателя на объект базового. Рассмотрим опять пример с часами и будильником (листинг 9.2).

Листинг 9.2. Неожиданная работа принципа подстановки

```
class Clock                                // базовый класс — часы
{ public:
    void print() const { cout << "Clock!" << endl; }
};
class Alarm: public Clock // производный класс — будильник
{ public:
    void print() const // переопределенный метод
    { cout << "Alarm!" << endl; }
};
```

Листинг 9.2 (продолжение)

```

void settime(Clock &d)      // функция установки времени
{ d.print(); }             // предполагается вызов метода базового класса
//...
Clock W;                   // объект базового класса
settime(W);                // выводится "Clock"
Alarm U;                   // объект производного класса
settime(U);                // ссылка на производный вместо базового
Clock *c1 = &W;            // адрес объекта базового класса
c1->print();                // вызов базового метода
c1 = &U;                   // адрес объекта производного типа вместо базового
c1->print();                // какой метод вызывается, базовый или производный?

```

Опять в классе-наследнике переопределен метод для того, чтобы обеспечить различное поведение объектов базового и производного классов. Однако и при передаче параметра по ссылке базового класса в функцию `settime()`, и при явном вызове метода `print()` через указатель базового класса наблюдается одна и та же картина: всегда вызывается метод базового класса, хотя намерения программиста состоят в том, чтобы вызвать метод производного.

Для того чтобы разобраться в ситуации, необходимо уяснить, что такое *связывание*. Связывание — это сопоставление вызова функции с телом. В приведенных ранее примерах связывание выполняется на этапе трансляции (до запуска) программы. Такое связывание обычно называют *ранним*, или *статическим*.

При трансляции класса `Base` (см. листинг 9.1) компилятор ничего не знает о классах-наследниках¹, поэтому он не может предполагать, что метод `f()` будет переопределен в классе `Derived`. Его естественное поведение — «прочно» связать вызов `f()` с телом метода класса `Base`. Аналогично при трансляции функции `settime()` компилятору ничего не известно о типе реально передаваемого объекта во время выполнения программы. Поэтому вызов метода `print()` связывается с телом метода базового класса `Clock`, как и определено в заголовке функции `settime()`. Точно так же указатель на базовый класс «прочно» связывается с методом базового класса во время трансляции.

Конечно, при вызове метода по указателю в данном конкретном случае мы можем вызвать метод производного класса, задав явное преобразование указателя:

```
static_cast<Alarm*>(c1)->print();
```

Или так:

```
((Alarm *)c1)->print();           // "лишние" скобки нужны!
```

Однако для функции `settime()` и метода `CallFunction()` это сделать невозможно — нам необходимо именно разное поведение в зависимости от типа объекта. Да и с указателем не все так просто: если такой вызов прописан внутри функции, которая принимает этот указатель как параметр (например, `settime(Clock *c1)`), то мы имеем те же проблемы.

¹ Эти классы могут быть определены в других файлах многомодульной программы (см. главу 11).

Определение виртуальных функций

Получается, что в C++ должен существовать механизм, с помощью которого можно узнать тип объекта во время выполнения программы. Такой механизм в C++ есть, и он, как уже отмечалось, называется динамической идентификацией типов (RTTI). Однако в ситуациях, подобных описанному, применяется другой, более «сильный» и элегантный механизм C++ — механизм *виртуальных функций* (см. п. 10.3 в [1]).

Чтобы добиться разного поведения в зависимости от типа, необходимо объявить функцию-метод виртуальной; в C++ это делается с помощью ключевого слова `virtual`. Таким образом, в листинге 9.1 объявление метода `f()` в базовом и производном классах должно быть таким:

```
virtual int f(const int &d)           // в базовом классе
    { return 2*d; }
virtual int f(const int &d)           // в производном классе
    { return d*d; }
```

После этого для объектов базового и производного классов мы получаем разные результаты: 11 и 26.

Аналогично в листинге 9.2 объявление метода `print()` тоже должно начинаться со слова `virtual`:

```
virtual void print() const           // в базовом классе
{ cout << "Clock!" << endl; }
virtual void print() const           // в производном классе
{ cout << "Alarm!" << endl; }
```

После этого вызов `settime()` с параметром базового класса обеспечит нам вывод на экран слова «Clock», а с параметром производного класса — слова «Alarm». И при вызове по указателю наблюдается та же картина.

Вообще-то ключевое слово `virtual` достаточно написать только один раз — в объявлении функции базового класса. Определение можно писать без слова `virtual` — все равно функция будет считаться виртуальной. Однако лучше всегда это делать явным образом, чтобы всегда по тексту было видно, что функция является виртуальной.

Для виртуальных функций обеспечивается не статическое, а *динамическое* (позднее, отложенное) связывание, которое реализуется во время выполнения программы. Естественно, это влечет за собой некоторые накладные расходы, однако на них можно не обращать внимания, так как обеспечивается *динамический полиморфизм*. Александреску в [20] указывает, что в C++ реализованы два типа полиморфизма:

- статический полиморфизм, или полиморфизм времени компиляции (*compile-time polymorphism*), осуществляется за счет перегрузки и шаблонов функций;
- динамический полиморфизм, или полиморфизм времени выполнения (*run-time polymorphism*), реализуется виртуальными функциями.

С перегрузкой функций «разбирается» компилятор, правильно подбирая вариант функции в той или иной ситуации. И полиморфизм шаблонных функций тоже реализуется на этапе компиляции. Естественно, выбор осуществляется статически. Выбор же виртуальной функции происходит динамически — при выполнении программы. Класс, включающий в себя виртуальные функции, называется *полиморфным*.

Правила описания и использования виртуальных функций-методов следующие:

1. Виртуальная функция может быть только методом класса.
2. Любую перегружаемую операцию-метод класса можно сделать виртуальной, например, операцию присваивания или операцию преобразования типа.
3. Виртуальная функция, как и сама виртуальность, наследуется.
4. Виртуальная функция может быть константной.
5. Если в базовом классе определена виртуальная функция, то метод производного класса с *такими же именем и прототипом* (включая тип возвращаемого значения и константность метода) автоматически является виртуальным (слово `virtual` указывать необязательно) и замещает функцию-метод базового класса.
6. Конструкторы не могут быть виртуальными.
7. Статические методы не могут быть виртуальными.
8. Деструкторы могут (чаще — должны) быть виртуальными — это гарантирует корректный возврат памяти через указатель базового класса.

Переопределение и перегрузка виртуальных функций

В этих правилах особо нужно обратить внимание на пункт 5: виртуальность реализуется для функций с *одинаковыми прототипами*, но работающих по-разному в базовом и производном классах. В приведенных ранее примерах виртуальные функции базового класса переопределялись в производном классе именно таким образом. Остается разобраться с несколькими вопросами:

1. Можно ли перегружать виртуальную функцию в классе?
2. Какие последствия повлечет за собой переопределение виртуальной функции в классе-наследнике с другим списком параметров?
3. Можно ли при переопределении виртуальной функции изменить только тип возвращаемого значения (или только константность)?
4. Можно ли виртуальную функцию вызвать не виртуально?

Перегрузка виртуальных функций, как и любых других методов, вполне допустима. Разберемся со вторым вопросом. Предположим, мы определили базовый класс с перегруженными виртуальными функциями:

```
class Base
{ public:
    virtual int f() const
    { cout << "Base::f()" << endl; return 0; }
```

```
virtual void f(const string &s) const
{ cout << "Base::f(string)"<< endl; }
};
```

Никаких проблем при трансляции такой класс не вызывает. Определим наследника:

```
class Derived: public Base
{ public:
    virtual int f(int) const
    { cout << "Derived::f(int)"<< endl; return 0; }
};
```

Здесь мы переопределили виртуальную функцию `f()` с новым списком параметров. И этот класс при трансляции не вызывает у компилятора вопросов. Однако при вызове методов возникают проблемы. Рассмотрим простой пример (листинг 9.3).

Листинг 9.3. Вызов перегруженных виртуальных методов

```
int main()
{
    Base b, *pb;           // объекты базового типа
    Derived d, *pd = &d;   // объекты производного типа
    pb = &d;               // здесь нужна виртуальность
    pb->f();                // вызывается базовый метод
    pb->f("name");          // вызывается базовый метод
    pb->f(1);               // ошибка!
    return 0;
}
```

В первых двух вызовах вызываются методы базового класса. Последний вариант вызова:

```
pb->f(1);                  // ошибка!
```

Этот вариант вроде бы обеспечивает вызов метода-наследника через указатель базового класса, однако компилятор опять пытается вызвать базовый метод:

```
virtual void f(const string &s) const
```

После чего сообщает о невозможности преобразования `int` в `string`! Таким образом, следующий метод можно вызвать только через указатель класса-наследника (или выполнив явное преобразование типа):

```
virtual int f(int) const
```

Точно так же и вызовы базовых методов через указатель наследника не транслируются — как будто производный класс не унаследовал эти функции:

```
pd->f();                   // отсутствие аргумента
pd->f("name");             // попытка преобразования char[5]->int
d.f();                    // отсутствие аргумента
d.f("name");              // попытка преобразования char[5]->int
b.f(1);                   // попытка преобразования int->string
```

Таким образом, как и для обычных, неvirtуальных, методов, виртуальный метод-наследник с тем же именем, но отличающимся прототипом, просто скрывает одноименные методы базового класса.

ВНИМАНИЕ

Это относится и к константности методов, то есть константный метод считается отличающимся от неконстантного метода с таким же прототипом.

Теперь просто переопределим методы базового класса с теми же прототипами:

```
class Derived: public Base
{ public:
    virtual int f(int) const
    { cout<<"Derived::f(int)"<<endl;
      return 0;
    }
    virtual void f(const string &s) const
    { cout << "Derived::f(string)"<< endl; }
    virtual int f() const
    { cout << "Derived::f()"<< endl;
      return 0;
    }
};
```

В этом случае нормальным (виртуальным) образом работают вызовы

```
pb = &d;                // здесь нужна виртуальность
pb->f();                // метод-наследник
pb->f("name");          // метод-наследник
```

Однако следующий вызов ведет к ошибке трансляции:

```
pb->f(1);                // ошибка!
```

Компилятор по-прежнему пытается связать этот вызов с методом базового класса:

```
virtual void f(const string &s) const
```

После чего компилятор сообщает о невозможности преобразования `int` в `string`. Таким образом, через указатель базового класса нельзя вызывать новые методы, определенные только в производном классе. Если это все-таки необходимо, нужно выполнить явное приведение типа:

```
((Derived *)pb)->f(1);
```

Если мы не хотим переопределять родительские методы, а нам все-таки нужно их вызывать, то можно использовать объявление `using`:

```
using Base::f;           // разрешение использовать скрытые базовые методы
```

Наш класс-наследник выглядит тогда так:

```
class Derived: public Base
{ public:
    virtual int f(int) const
    { cout << "Derived::f(int)"<< endl; return 0; }
    using Base::f;          // разрешение использовать скрытые базовые методы
};
```

Объявление `using` действует на весь класс независимо от места записи. Например, мы могли бы вызвать родительский метод в методе-наследнике:

```
virtual int f(int) const
{ f(); return 0; }
```

Несмотря на то, что объявление `using` записано после метода, никаких сообщений об отсутствии определения `f()` не выдается.

Для ответа на третий вопрос напишем еще один класс-наследник:

```
class D: public Base
{ public:
    virtual void f() const
    { cout << "Derived::f()" << endl; }
};
```

Однако этот класс не транслируется: в таком виде изменять возвращаемое значение виртуальной функции запрещено стандартом (см. п. п. 10.3/5 в [1]). В то же время стандарт разрешает изменять возвращаемое значение, если это указатель или ссылка. Подробнее этот вопрос мы рассмотрим далее.

И наконец, может понадобиться вызывать виртуальную функцию не виртуально. Это значит, что мы должны явно указать компилятору, функцию какого класса нам требуется вызвать. Очевидно, что это можно сделать с помощью квалификатора класса, например:

```
Clock *c1 = new Alarm(); // адресуется объект производного класса
c1->print();             // вызов метода-наследника
c1->Clock::print();       // явно вызывается базовый метод
```

Возникает естественный вопрос: зачем может потребоваться такой *статический* вызов? Ведь метод специально сделан виртуальным, чтобы обеспечить динамическое связывание. Ответ очевиден: в базовом классе реализуются некоторые общие действия, которые должны выполняться во всех классах-наследниках. Мы познакомимся с этой техникой при изучении *чистых* виртуальных функций.

Размеры классов с виртуальными функциями

Ранее было сказано, что динамический полиморфизм влечет за собой некоторые накладные расходы. Как вы помните, обычные методы места в классе не занимают. Однако с виртуальными методами ситуация другая. Следующий пример показывает, что при наличии хотя бы одной виртуальной функции размер класса без полей равен четырем (листинг 9.4).

Листинг 9.4. Размеры классов с виртуальными функциями

```
class OneVirtual
{ virtual void f(void) {}
};
class TwoVirtual
{ virtual void f(void) {}
  virtual void g(void) {}
};
int main()
{ cout << sizeof(OneVirtual) << endl; // размер = 4
  cout << sizeof(TwoVirtual) << endl; // размер = 4
  return 0;
}
```


В то же время количество виртуальных функций, очевидно, роли не играет — размер класса остается равен четырем. В данном случае 4 — это размер указателя.

ПРИМЕЧАНИЕ

Мы работаем на платформе Intel под Windows. На других платформах размер указателя может быть другим.

Если в классе есть «родные» или унаследованные виртуальные функции, компилятор создает таблицу виртуальных функций — обычно ее называют VMT (Virtual Method Table). В этой таблице для каждой виртуальной функции содержится указатель на нее. Адрес этой таблицы и помещается в класс.

Таким образом, при наличии виртуальных функций расходуется дополнительная память. Выполняется программа тоже несколько медленнее, чем при статическом связывании. Во-первых, при создании объекта с виртуальными методами таблицу VMT требуется инициализировать — это должно быть сделано в конструкторе. Во-вторых, функции вызываются косвенно — через указатель таблицы. Тем не менее динамический полиморфизм — это настолько важный механизм, что разработчики языка Java решили сделать все методы виртуальными по умолчанию¹. В C++ виртуальность управляема, и такая плата² считается вполне приемлемой.

Виртуальные функции в конструкторах и деструкторах

Внутри конструкторов и деструкторов динамическое связывание не работает, хотя вызов виртуальных функций не запрещен. Обычно виртуальный вызов означает вызов метода наследника посредством ссылки или указателя на базовый класс. Однако в конструкторах и деструкторах вызывается всегда «родная» функция.

В конструкторах такое поведение оправдано тем, что код конструктора ничего не знает о производных классах. Во время работы конструктора были созданы объекты базовых классов, однако до производных классов дело еще не дошло. Данный конструктор сам мог быть вызван из конструктора производного класса, чтобы создать и проинициализировать объект своего класса. Однако конструктор не знает, кто его вызвал и зачем. Если бы вызов был виртуальным, то были бы возможны ситуации использования неинициализированных переменных производных классов.

В деструкторах ситуация несколько другая — виртуальный вызов опасен тем, что возможен вызов метода уже уничтоженного объекта. Поэтому даже в виртуальном деструкторе вызывается всегда «родной» метод.

¹ Аналогичное решение принято и в языках Smalltalk и Python.

² Платой за управляемую виртуальность является расход памяти (дополнительные 4 байта на указатель) и времени (инициализация таблицы виртуальных функций).

Чистые виртуальные функции

Когда мы создаем иерархию классов, вершиной иерархии становится класс, в котором перечислены максимально общие свойства всех потомков. Однако в каждом потомке эти свойства — разные. В качестве примера рассмотрим музыкальные инструменты. Инструменты бывают, например, духовыми, струнными и ударными, а духовые инструменты — деревянными и медными. Например, скрипка — струнный инструмент, а флейта — деревянный духовой. Схема иерархии классов может быть такой, как показано в листинге 9.5.

Листинг 9.5. Иерархия музыкальных инструментов

```
class Instrument;           // общий базовый класс
class Wind: public Instrument; // духовые
    class Brass: public Wind; // медные
        class trombone: public Brass; // тромбон
    class Woodwind: public Wind; // деревянные
        class flute: public Woodwind; // флейта
class Stringer: public Instrument; // струнные
    class violin: public Stringer; // скрипка
class Percussion: public Instrument; // ударные
    class cymbals : public Percussion; // тарелки
```

Все инструменты имеют некоторые общие свойства, например:

- ☐ каждый инструмент как-то конкретно называется;
- ☐ на каждом инструменте как-то играют.

Кроме того, все струнные инструменты настраивают. Однако для каждого конкретного инструмента эти свойства разные: на скрипке играют совсем не так, как на флейте, и настройка гитары сильно отличается от настройки фортепиано. Понятно, что функции, реализующие эти свойства, должны быть виртуальными. Здесь возникает одна небольшая проблема: как реализовать эти функции в базовом классе `Instrument`? Такие общие функции не могут ничего делать в базовом классе — вся работа должна выполняться в производных классах. Единственная возможность добиться этого — определить виртуальные функции с пустым телом, а в производных классах переопределить их. Например, функция для игры может быть такой:

```
enum note { Cmiddle, Csharp, Cflat }; // ноты до, до-диез, до-бемоль
virtual void play(note) const {} // играть ноту
```

Тогда функция настройки (в базовом классе `Stringer` для струнных инструментов) может выглядеть так:

```
virtual void adjust() const {} // настройка
```

Решение, конечно, приемлемое, но ... «пустое» тело на самом деле не означает отсутствия выполняемых команд. Обычно в оттранслированной функции при входе выполняется так называемый стандартный пролог, а при выходе — стандартный эпилог. Поэтому даже «пустое» тело при выполнении требует накладных расходов.

Для нашего общего предка хорошо подошли бы функции, *на самом деле* не имеющие тела. Набор таких функций мог бы обозначать только общий интерфейс класса¹ (контракт класса), а конкретная их реализация может быть выполнена позже — в классах-наследниках. Отсутствие тела нужно как-то обозначить, ведь мы не можем написать просто прототип — это требует определения функции. Осознав проблему, Б. Страуструп поступил очень просто: он обозначил отсутствие тела функции нулем! Называется такая виртуальная функция «чистой» (*pure*) и определяется следующим образом:

```
virtual тип имя(параметры) = 0;
```

Несмотря на то, что виртуальная функция — чистая, для такого класса все равно создается таблица виртуальных функций. Страуструп Б. не стал вводить в C++ новое ключевое слово вроде *pure* или *abstract*, чтобы обозначить отсутствие тела функции, — вместо этого присвоением нуля он подчеркнул, что в таблице виртуальных функций адрес этой функции равен нулю, поэтому вызвать ее нельзя.

Класс, в котором есть хотя бы одна чистая виртуальная функция, называется *абстрактным* (см. п. 10.4 в [1]). Абстрактный класс отличается от «нормального» класса тем, что объект абстрактного класса создать нельзя, даже динамически операцией *new*. И при передаче параметра в функцию невозможно передать объект абстрактного класса по значению — копию-то создать нельзя! Однако указатели (и ссылки) определять можно, так как для указателя размер класса не важен.

При наследовании абстрактность сохраняется: если класс-наследник не реализует унаследованную чистую виртуальную функцию, то он тоже является абстрактным. В C++ абстрактный класс выражает понятие *интерфейса*. Наследование от абстрактного класса — это *наследование интерфейса*. По этой причине абстрактные классы часто являются вершиной иерархии классов, предоставляя общий базовый интерфейс для всей иерархии. Для музыкальных инструментов базовый класс *Instrument*, очевидно, должен быть абстрактным классом, в котором определены две чистые виртуальные функции:

```
class Instrument
{ public:
    virtual void play(note) const = 0;      // играть ноту
    virtual string name() const = 0;       // выдать название
};
```

Рассмотрим простой пример наследования от абстрактных классов (листинг 9.6), и заодно продемонстрируем изменение возвращаемого значения виртуальной функции.

Листинг 9.6. Наследование от абстрактного класса

```
class Food                                // абстрактный класс "Пища животного"
{ public:
    virtual string typeFood() const = 0;    // тип пищи
};
```

¹ Герб Саттер в [21] и Скотт Мейерс в [23] называют такие классы классами-протоколами.

```

class Animal                                // абстрактный класс "Животное"
{ public:
    virtual string kind() const = 0;        // вид животного
    virtual Food& eats() = 0;              // что ест
};
class Tiger: public Animal                  // Тигр - конкретный вид животного
{ public:
    virtual string kind() const             // идентификация вида
    { return "Tiger"; }
    class TigerFood: public Food            // пища тигра
    { public:
        virtual string typeFood() const    // идентификация пищи тигра
        { return "Tiger food"; }
    };
    virtual TigerFood& eats()               // ест пищу тигра
    { return tiger; }
private:
    TigerFood tiger;                       // объект - пища тигра
};
class Dog: public Animal                    // Собака - конкретный вид животного
{ public:
    virtual string kind() const             // идентификация вида
    { return "Dog"; }
    class DogFood: public Food              // пища собаки
    { public:
        virtual string typeFood() const    // идентификация пищи собаки
        { return "Dog food"; }
    };
    virtual DogFood& eats()                 // ест пищу собаки
    { return dog; }
private:
    DogFood dog;                           // объект - пища собаки
};
int main()
{ Tiger t; Dog d;                          // объекты-животные
  Animal& pt = t; Animal& pd = d;          // подстановка ссылок
  Animal *p[] = { &t, &d };               // подстановка указателей
  // виртуальный вызов по ссылке
  cout << pt.eats().typeFood() << endl;
  cout << pd.eats().typeFood() << endl;
  // виртуальный вызов по указателю
  cout << p[0]->eats().typeFood() << endl;
  cout << p[1]->eats().typeFood() << endl;
  // присвоение измененного типа
  Tiger::TigerFood tf = t.eats();
  Dog::DogFood df = d.eats();
  return 0;
}

```

Сначала определены два абстрактных базовых класса: Food и Animal. Эти классы определяют для своих наследников минимальные общие интерфейсы (чистые виртуальные функции). Класс Tiger, наследуя от Animal, содержит вложенный класс TigerFood — наследника от Food. Класс TigerFood вложен в Tiger, так

как эти классы тесно связаны — тигр есть только свою пищу. Впрочем, исходный абстрактный базовый класс `Food` можно вложить и в исходный абстрактный класс `Animal`:

```
class Animal
{ public:
    class Food // вложенный абстрактный класс
    { public:
        virtual string typeFood() const = 0;
    };
    virtual string kind() const = 0;
    virtual Food& eats() = 0;
};
```

Тогда вложенный класс `TigerFood` будет наследником от вложенного абстрактного класса `Food`.

Метод `eats()`, который в базовом классе `Animal` возвращал ссылку на базовый класс `Food`, переопределен и возвращает ссылку на `TigerFood` — изменено возвращаемое значение. Аналогично можно изменять и тип возвращаемого указателя. Точно такое строение имеет и класс `Dog`. Главная программа демонстрирует правильные вызовы виртуальных функций `eats()` через ссылки и указатели базового типа.

Трудно переоценить роль абстрактных классов в практике программирования. Они широко применяются в паттернах проектирования [17, 18]. Вся компонентная технология [51, 52] построена на абстрактных классах-интерфейсах. Они нам еще не раз пригодятся.

Виртуальные деструкторы

Ключевое слово `virtual` не может быть использовано вместе с конструкторами, однако деструкторы могут и часто должны быть виртуальными (см. п. п. 12.4/7 в [1]).

Если мы работаем с объектом производного класса через указатель базового класса, то при уничтожении объекта будет вызван деструктор только базового класса. А нам нужно, чтобы сначала был вызван деструктор производного класса, а в нем уже — деструктор базового класса. Такие ошибки очень трудно находить: до поры, до времени программа работает нормально. Но в один «прекрасный» моме...

Здесь мы имеем все ту же проблему, для решения которой и были придуманы виртуальные функции. Естественно, «отец-основатель» прекрасно понимал всю серьезность проблемы. Поэтому деструкторы тоже могут быть объявлены виртуальными, и для таких деструкторов выполняется динамическое связывание. Простой пример демонстрирует работу виртуальных деструкторов (листинг 9.7).

Листинг 9.7. Виртуальные деструкторы

```
class Base
{ public:
    ~Base() // деструктор не виртуальный
    { cout << "Base::Not virtual destructor!" << endl; }
```

```
};
class Derived: public Base      // наследник
{ public:
    ~Derived()                  // деструктор не виртуальный
    { cout << "Derived::Not virtual destructor!"<< endl; }
};
class VBase
{ public:
    virtual ~VBase()            // деструктор виртуальный
    { cout << "Base::virtual destructor!"<< endl; }
};
class VDerived: public VBase
{ public:                        // деструктор виртуальный
    ~VDerived()                 // virtual можно не писать
    { cout << "Derived::virtual destructor!"<< endl; }
};
int main()
{
    Base *bp = new Derived();    // подстановка - повышающее приведение типа
    delete bp;                  // вызывается только базовый деструктор
    VBase *vbp = new VDerived(); // подстановка
    delete vbp;                 // вызывается производный, а потом базовый
    return 0;
}
```

При выполнении оператора `delete bp` будет вызван деструктор только базового класса — налицо утечка памяти, так как был создан и не уничтожен объект производного класса. При выполнении оператора `delete vbp` все работает правильно, так как вызывается сначала деструктор производного класса, а потом базового (происходит динамическое связывание виртуального деструктора).

Чистые виртуальные деструкторы

Деструкторам разрешено быть «чистыми». Объявляются такие деструкторы точно так же, как чистые виртуальные функции, например:

```
virtual ~VBase() = 0;
```

Класс, в котором определен чистый виртуальный деструктор, тоже является абстрактным, и создавать объекты этого класса запрещено. Однако чистые виртуальные деструкторы несколько отличаются от чистых виртуальных функций. Первое различие проявляется при наследовании. Вспомним, что деструкторы не наследуются, а при отсутствии явного определения автоматически создаются в производном классе. Это означает, что класс-наследник *не является* абстрактным классом!

ПРИМЕЧАНИЕ

Аналогичная ситуация возникает и при объявлении чистой виртуальной функцией операции присваивания `operator=`. В наследнике по умолчанию создается собственная операция присваивания, поэтому класс-потомок не будет абстрактным.

Второе различие заключается в том, что при объявлении чисто виртуального деструктора нужно написать и его определение. Да-да, не поднимайте удивленно брови — нужно написать определение чистого виртуального деструктора для

базового класса. Это несколько странное правило объясняется тем, что деструктор наследника обязательно вызывает деструктор базового класса, поэтому чистый деструктор все же должен быть определен. Пример в листинге 9.8 показывает, как это делать.

Листинг 9.8. Чистый виртуальный деструктор

```
class Abstract                                // абстрактный класс
{ public:
    virtual ~Abstract()=0;                    // чистый виртуальный деструктор
};
Abstract::~~Abstract()                        // определение чистого деструктора
{ cout << "Abstract"<< endl; }
// наследник - не абстрактный класс
class NotAbstract: public Abstract { };
```

Как видим, определение чистого виртуального деструктора ничем не отличается от обычного внешнего определения метода. Убедиться в том, что класс `Abstract` даже при наличии определения остается абстрактным, а его наследник — нет, очень легко — достаточно попытаться создать объект этих классов. Для объекта класса `Abstract` компилятор немедленно выдает сообщение о том, что нельзя создавать объекты абстрактного класса. Объект класса-наследника создается без проблем.

Определим в наследнике явный деструктор:

```
virtual ~NotAbstract()                        // явный деструктор
{ cout << "NotAbstract"<< endl; }
```

Напишем в программе код создания и удаления динамического объекта производного класса и при этом инициализируем указатель базового абстрактного класса:

```
Abstract *pt = new NotAbstract();
delete pt;
```

При выполнении этого фрагмента мы увидим, что сначала вызывается деструктор класса-наследника, потом — деструктор базового класса. Если же мы удалим определение чистого виртуального деструктора, то получим сообщение компилятора об отсутствии тела функции-деструктора.

Определять можно не только чистые деструкторы, но и чистые виртуальные методы. Класс, для которого это определение написано, по-прежнему является абстрактным классом. В отличие от чистых виртуальных деструкторов, даже определенная в базовом классе чистая виртуальная функция наследуется как чистая, поэтому производный класс при отсутствии собственного определения тоже будет абстрактным. В этом легко убедиться, написав пример, показанный в листинге 9.9.

Тогда для чего нужно определение в базовом классе? Ответ, пожалуй, только один: в определении чистого виртуального метода базового класса задается некоторый общий код, который могут использовать классы-наследники в своих реализациях этой функции (листинг 9.10). При этом обратите внимание, что чистый метод базового класса вызывается *статически*¹.

¹ Не путайте со статическими методами, которые мы рассматривали ранее в главе 4.

Листинг 9.9. Определение чистой виртуальной функции

```

class Animal                                // абстрактный класс
{ public:
    virtual void eats() = 0;                // чистый виртуальный метод
};
void Animal::eats()                          // определение чистого метода
{ cout << "Animal::eats"<< endl; }
class Tiger: public Animal                  // тоже абстрактный класс
{ };
int main()
{ Tiger t;                                // ошибка - объект создать нельзя!
}

```

Листинг 9.10. Использование определения чистого виртуального метода

```

class Animal                                // абстрактный класс
{ public:
    virtual void eats() = 0; // чистый виртуальный метод
};
// определение чистого виртуального метода
void Animal::eats()
{ cout << "Animal::eats"<< endl; }
class Tiger: public Animal                  // уже не абстрактный класс
{ public:
    virtual void eats();
};
void Tiger::eats()
{ Animal::eats();                          // статический вызов чистой базовой функции
  // собственные действия
}
class Cat: public Animal                   // уже не абстрактный класс
{ public:
    virtual void eats();
};
void Cat::eats()
{ Animal::eats();                          // статический вызов чистой базовой функции
  // собственные действия
}

```

Применение такой техники можно обнаружить в некоторых *паттернах* [17, 18].

Однокоренная иерархия

В главе 6 мы затронули проблему универсальности контейнеров, работа которых фактически не зависит от типа элементов. Открытое наследование и виртуальные деструкторы позволяют нам решить проблему универсализации по-другому. Идея состоит в последовательном применении полиморфизма: нужно определить единый базовый класс, а все остальные классы сделать наследниками от него. Тогда контейнер разрабатывается под элементы базового класса, однако принцип подстановки позволяет использовать его и с элементами производных классов. Базовый класс в простейшем виде может содержать только виртуальный деструктор, например:

```

class TObject
{ public:
    virtual ~TObject() = 0;
};
inline TObject::~~TObject() {}

```


Тогда контейнер-стек может быть реализован так, чтобы хранить указатели на TObject (листинг 9.11).

Листинг 9.11. Стек для хранения объектов базового класса

```
class TStack
{ struct Elem                                // узел списка
  { Object *data;                            // информационная часть
    Elem *next;                             // указательная часть
    Elem (TObject *d, Elem *p)              // конструктор узла
      :data(d), next(p)
    { }
  };
  Elem * Head;                              // вершина стека
  TStack(const TStack &);                   // закрыли копирование
  TStack& operator=(const TStack &);        // закрыли присваивание
public:
  TStack(): Head(0) {}                      // пустой стек
  void push(TObject *d)                     // поместить элемент в стек
  { Head = new Elem(d, Head); }
  void *top() const                          // получить элемент с вершины стека
  { return empty()? 0: Head->data; }
  TObject *pop()                            // удалить элемент из стека
  { if (empty()) return 0;                  // если стек пустой – ничего не делать
    Object *top = Head->data;                // сохранили для возврата
    Elem *oldHead = Head;                   // запомнили указатель
    Head = Head->next;                       // передвинули вершину
    delete oldHead;                         // возвратили память
    return top;                             // возвратили элемент
  }
  bool empty() const                        // стек пустой. если указатель = 0
  { return Head==0; }
};
```

В таком варианте в контейнер не попадут «случайные» указатели, как это было возможно в прежнем варианте с нетипизированными указателями. Наследуя от такого стека, можно реализовать стек для любого типа, унаследовавшего от TObject.

Такой подход не только имеет право на жизнь, но и достаточно широко применяется. В «самом объектно-ориентированном» языке программирования Smalltalk все классы считаются наследниками единственного базового класса, определенного разработчиками языка. Из современных языков, в которых реализована та же идея, можно отметить Java и C#. Интересно, что в этих языках отсутствуют средства работы с динамической памятью.

В C++ такого единого базового класса нет, однако было создано немало библиотек, построенных по этому принципу. Самые известные из них — библиотека MFC (Microsoft Foundation Classes) [46], в которой реализован единый базовый класс CObject, и библиотека VCL (Visual Component Library) [49, 54], в которой основным базовым классом является TObject.

Виртуализация внешних функций

А теперь вспомним наш пример со сложением денег и выводом результатов на экран. Несмотря на то, что дружественные функции не могут быть виртуальными, мы все-таки можем обеспечить различное поведение операции вывода `operator<<` для разной валюты с помощью механизма виртуальных функций. Следующий прием является уже практически стандартным: в базовом классе объявляется чистая виртуальная функция-метод, а в производных классах реализуется ее определение. Внешней независимой функции вывода передается параметр базового класса, для которого и вызывается базовая виртуальная функция. Простейшая работающая схема приведена в листинге 9.12.

Листинг 9.12. «Виртуализация» внешних функций

```
class TControl
{ public:
    virtual ostream& print (ostream &t) const = 0;
};
class TButton: public TControl
{ public:
    virtual ostream& print (ostream &t) const
    { return (t<<"TButton!"); }
};
class TListBox: public TControl
{ public:
    virtual ostream& print (ostream &t) const
    { return (t<<"TListBox!"); }
};
inline          // для снижения накладных расходов на виртуализацию
ostream& operator<<(ostream &t, const TControl &r)
{ return r.print(t); }
```

Обратите внимание: внешняя функция — единственная, и она даже не является дружественной! Опять задействован принцип подстановки: в определении внешней функции-операции второй параметр — ссылка на базовый тип, а вызывается операция с параметрами производных типов. Поэтому следующий фрагмент работает совершенно правильно:

```
TButton b;          cout << b << endl;
TListBox t;         cout << t << endl;
```

Вывод на экран выглядит так:

```
TButton!
TListBox!
```

Показанный прием в сочетании со статическим вызовом виртуальной функции базового класса и позволяет нам виртуализировать вывод в наших денежных классах `TCurrency`, `Rouples` и `Bucks`. Однако мы не можем в классе `TCurrency` объявить чистую виртуальную функцию — это сделает класс абстрактным, и возникнут ошибки, связанные с объявлением объектов в методах (например, в методах инкремента). Чтобы сделать класс `TCurrency` абстрактным, необходима существенная переработка методов — мы этого делать не будем.

Определим в базовом классе обычную (не чистую) виртуальную функцию `print()`, которая будет выполнять общую для любой валюты работу — преобразование суммы в строку. Для этого добавим в раздел `protected` базового класса строковое поле. Наследники вызывают базовую функцию статически (листинг 9.13).

Листинг 9.13. «Виртуализация» функции вывода для валют

```
class TCurrency
{ protected:
    string toString() const;
    string s;
public:
    virtual ostream& print(ostream &t)
    { s = this->toString();
      return t;                                // возвращаем ссылку на поток
    }
    //...
};
// Рубли
class Roubles: public TCurrency
{ public:
    Roubles(const long double &r=0.0)        // конструктор
    :TCurrency(r)                            // вызов базового конструктора
    { };
    virtual ostream& print(ostream &t)
    { TCurrency::print(t);                   // вызов базовой функции
      return (t << s << "p.");
    }
};
// Доллары
class Bucks: public TCurrency
{ public:
    Bucks(const long double &r=0.0):TCurrency(r) // конструктор
    { };
    virtual ostream& print(ostream &t)
    { TCurrency::print(t);                   // вызов базовой функции
      return (t << '$' << s);
    }
};
// внешняя функция вывода
inline
ostream& operator<<(ostream& t, TCurrency &r)
{ return r.print(t);                        // виртуальный вызов
}
int main()
{ Roubles t(15);
  ++t; t++;      cout << t << endl;        // виртуальный вызов
  Bucks b(10);
  ++b; b++;      cout << b << endl;        // виртуальный вызов
  return 0;
}
```

Вывод выполняется совершенно правильно.

Немного философии

Любой класс имеет две стороны медали: интерфейс и реализацию. Чтобы хоть как-то упорядочить наши знания в этом вопросе, давайте порассуждаем, что и в каких случаях мы наследуем. С закрытым наследованием все вроде бы ясно: это — наследование реализации, так как ни клиенты, ни последующие наследники «не видят» интерфейса при закрытом наследовании, если только его специально не открыть.

Совсем другое дело — открытое наследование. Здесь все видят все, поэтому важно понимать, что и в каких случаях наследуется. Понятно, что интерфейс класса наследуется всегда. Но при открытом наследовании существуют важные концептуальные различия в наследовании обычных, виртуальных и чисто виртуальных функций.

Совершенно очевидно, что объявление чисто виртуальной функции предполагает наследование только интерфейса, оставляя реализацию классам-наследникам. Даже если чисто виртуальная функция имеет определение, это определение не наследуется и класс-наследник остается абстрактным.

Если функция не объявлена виртуальной, то изменение ее поведения в наследниках, очевидно, не предполагается. Поэтому можно сказать (хотя это и не поддерживается непосредственно в языке), что реализация неvirtуальной функции является *обязательной* для классов-наследников — они не должны ее изменять.

Труднее разобраться, что же мы наследуем, если функция объявлена виртуальной. С одной стороны, мы вроде бы не собираемся наследовать реализацию — именно для этого функция объявлена виртуальной. С другой стороны, мы не объявили эту функцию чистой виртуальной — значит, предполагается, что реализация все-таки пригодится. Как правильно заметил Скотт Мейерс в [23], в этом случае мы наследуем *реализацию по умолчанию*: если мы не переопределим виртуальную функцию в наследнике, то будет работать базовая реализация по умолчанию.

Заметьте, что определение чистой виртуальной функции не является реализацией по умолчанию — мы должны явным образом вызывать такую функцию в методе-наследнике.

В [27] приводится интересная классификация форм наследования. Как и всякая классификация, эта — личное мнение автора книги [27]. Однако она может служить отправной точкой для понимания, с какой целью мы применяем этот мощный механизм — наследование. Автор считает, что порождение дочернего класса может быть выполнено по следующим причинам:

- **Специализация.** Класс-наследник является специализированной формой родительского класса — в наследнике просто переопределяются методы. Принцип подстановки выполняется. Очевидно, что такая форма наследования в C++ реализуется простым открытым наследованием. Примером является наследование часы будильник.

- *Спецификация.* Дочерний класс реализует поведение, описанное в родительском классе. Ясно, что в C++ эта форма реализуется простым открытым наследованием от абстрактного класса.
- *Конструирование.* Класс-наследник использует методы базового класса, но не является его подтипом (принцип подстановки не выполняется). В C++ такую форму можно реализовать простым закрытым наследованием.
- *Расширение.* В класс-потомок добавляют новые методы, расширяя поведение родительского класса; принцип подстановки в такой форме выполняется.
- *Обобщение.* Дочерний класс обобщает поведение базового класса. Обычно такое наследование требуется в тех случаях, когда мы не можем изменить поведение базового класса (например, базовый класс является библиотечным).
- *Ограничение.* Класс-наследник ограничивает поведение родительского класса. Очевидно, что в C++ такой вид наследования реализуется простым закрытым наследованием (пример — `TDeque` -> `TStack`).
- *Варьирование.* Базовый и производный классы являются вариациями на одну тему, однако связь «класс-подкласс» произвольна, например, «квадрат-прямоугольник» или «прямоугольник-квадрат». Эта форма фактически не отличается от «конструирования», так как класс-наследник, очевидно, «использует методы базового класса, но не является его подтипом».
- *Комбинирование.* Дочерний класс наследует черты нескольких классов — это множественное наследование (см. главу 10).

Наследование — это огромный пласт C++, которого мы в этой главе только коснулись.

Резюме

Особую роль при наследовании играют виртуальные функции, которые позволяют изменять поведение методов. С помощью виртуальных методов в C++ реализуется динамический полиморфизм времени выполнения. Виртуальные функции наследуются, их можно перегружать и переопределять. Конструкторы и статические методы не могут быть виртуальными. Виртуальную функцию можно вызывать не виртуально. В конструкторах и деструкторе виртуальность не работает — вызываются только «родные» методы. Наличие в классе виртуальных функций обычно увеличивает размер класса.

Деструкторы тоже могут быть виртуальными. Чистые виртуальные функции не имеют тела. Наличие в классе чистой виртуальной функции делает класс абстрактным; объекты абстрактного класса создавать запрещено, но указатели и ссылки — можно. Абстрактность наследуется. Деструктор тоже может быть чистым, при этом необходимо его определить; «чистота» деструктора не наследуется. Чистую виртуальную функцию тоже можно определить, однако ее «чистота» наследуется — класс-наследник остается абстрактным, если не реализует унаследованную чистую виртуальную функцию.

Для классов в иерархии наследования виртуальные функции позволяют реализовать разное поведение внешних функций, которые даже не являются дружественными.

Чистый виртуальный деструктор вкупе с простым открытым наследованием позволяют реализовать в языке однокоренную иерархию наследования. Во многих языках такая иерархия определяется в самом языке (Java, C#). Однако в C++ не существует стандартной иерархии, поэтому она реализована во многих библиотеках, например в MFC и VCL.

Контрольные вопросы

1. Объясните, зачем нужны виртуальные функции.
2. Что такое связывание?
3. Чем раннее связывание отличается от позднего?
4. Какие два вида полиморфизма реализованы в C++?
5. Влияет ли наличие виртуальных функций на размер класса?
6. Дайте определение полиморфного класса.
7. Может ли виртуальная функция быть дружественной функцией класса?
8. Наследуются ли виртуальные функции?
9. Каковы особенности вызова виртуальных функций в конструкторах и деструкторах?
10. Можно ли сделать виртуальной перегруженную операцию, например сложение?
11. Можно ли виртуальную функцию вызвать не виртуально?
12. Может ли конструктор быть виртуальным? А деструктор?
13. В каких случаях вызов виртуальной функции класса-наследника через указатель базового класса требует явного преобразования типа?
14. Как виртуальные функции влияют на размер класса?
15. Как объявляется чистая виртуальная функция?
16. Дайте определение абстрактного класса.
17. Наследуются ли чистые виртуальные функции?
18. Можно ли объявить деструктор чисто виртуальным?
19. Чем отличается чистый виртуальный деструктор от чистой виртуальной функции?
20. Зачем требуется определение чистого виртуального деструктора?
21. Можно ли сделать операцию присваивания виртуальной? А операцию индексирования? А чистой виртуальной?
22. Объясните, как можно реализовать «виртуальность» независимой функции.
23. Приведите классификацию целей наследования.
24. Объясните разницу между наследованием интерфейса и наследованием реализации.
25. Как связаны виртуальные функции и принцип подстановки?

Упражнения

1. В классе `Array` (см. упражнение 6 в главе 8) реализовать виртуальную функцию поэлементного сложения массивов. Реализовать классы-наследники `Decimal` и `BitString`, переопределив виртуальную функцию сложения.
2. В классе `Pair` (см. упражнение 1 в главе 8) операцию сложения пар реализовать как виртуальную. Реализовать класс-наследник `Rational`, переопределив операцию сложения как $(a, b) + (c, d) = (ad + bc, bd)$.
3. Создать абстрактный базовый класс `Number` с виртуальными методами — арифметическими операциями сложения, вычитания, умножения, деления, возведения в степень, получения остатка от деления. Создать производные классы `Integer` (целое) и `Real` (действительное).
4. Создать абстрактный базовый класс `Figure` с виртуальными методами вычисления площади и периметра. Создать производные классы: `Rectangle` (прямоугольник), `Circle` (круг), `Trapezium` (трапеция) со своими функциями площади и периметра.

Площадь трапеции

$$S = (a + b) \cdot h / 2.$$

Площадь круга

$$S = R^2.$$

5. Создать абстрактный базовый класс `Function` (функция) с виртуальными методами вычисления значения функции $y = f(x)$ в заданной точке x и вывода результата на экран. Определить производные классы `CubePolinom`, `Hyperbola` с собственными функциями вычисления y в зависимости от входного параметра x .

Уравнение полинома:

$$y = a \cdot x^3 + b \cdot x^2 + c \cdot x + d.$$

Уравнение гиперболы:

$$x^2 / a^2 - y^2 / b^2 = 1.$$

Реализовать функцию вывода по образцу листинга 9.12.

6. Реализовать однокоренную иерархию для объектов чисел на основе абстрактного базового класса `Number`. Иерархия должна включать действительные (`Real`), комплексные (`Complex`) и дробные (`Fraction`) числа. Создать класс-контейнер, аналогичный `TDeque`, с элементами типа `Number`. Реализовать виртуальную функцию вывода по образцу листинга 9.12.
7. Преобразовать класс `TCurrency` в абстрактный и реализовать классы-наследники `Rouble` и `Dollar`. Операции реализовать как внешние функции, используя прием виртуализации внешних функций (см. листинг 9.12).

Глава 10

Множественное наследование и RTTI

Одним из мощных средств разработки является множественное наследование (см. п. 10.1 в [1]). Впервые оно было реализовано в 1989 году [43]. Этот механизм с концептуальной точки зрения является совершенно естественным: если есть два класса, каждый из которых обладает частью необходимых свойств, то почему бы не объединить эти свойства в одном классе-наследнике? В конце концов, и в реальной жизни у каждого ребенка двое родителей.

Но, как обычно и бывает в этой жизни, «хотели как лучше, а получилось как всегда»: множественное наследование в том виде, в котором оно реализовано в C++, часто является источником проблем и подвергается залуженной критике. Недаром в новейших объектно-ориентированных языках Java и C# множественное наследование классов запрещено, а разрешено только множественное наследование интерфейсов.

Тем не менее при аккуратном и правильном использовании — это полезный и мощный механизм. Множественное наследование очень пригодились при реализации библиотек классов. Например, в объектно-ориентированной библиотеке ввода-вывода множественное наследование применяется для реализации класса `iostream`, объединяющего свойства классов `istream` и `ostream`.

Другим механизмом C++, который оказался востребованным при разработке больших библиотек классов, является уже упоминавшийся нами в главах 7 и 9 механизм динамической идентификации типов (RTTI). Использование этого механизма позволяет повысить «степень» универсальности библиотеки классов. Фактически разработка этого механизма осуществлялась одновременно с разработкой шаблонов [43], которые тоже предназначены для повышения универсальности. Однако механизм RTTI не очень сложно реализовать «вручную» [12, 26], чего нельзя сказать о реализации шаблонов. Поскольку именно RTTI часто создавали разработчики библиотек, Б. Страуструп, в конце концов, включил в C++ стандартный механизм RTTI [43].

RTTI, так же как и множественное наследование, при программировании на C++ можно рассматривать как «дополнительный» инструмент, являющийся в некотором роде альтернативой виртуальным функциям. И хотя сам Б. Страуструп призывает использовать именно виртуальные функции, принципы работы RTTI значительно понятнее новичкам, чем механизм виртуальных функций. К тому же в некоторых случаях, как пишет сам Б. Страуструп в [43], без RTTI невозможно обойтись.

Множественное наследование

Множественное наследование отличается от простого (одиночного) наличием нескольких базовых классов, например:

```
class A {};  
class B {};  
class D: public A, public B  
{};
```

Базовые классы перечисляются через запятую; количество их стандартом не ограничивается. Модификатор наследования для каждого базового класса может быть разным: можно от одного класса наследовать открыто, а от другого — закрыто.

При множественном наследовании выполняется все то же самое, что и при одиночном, то есть класс-потомок наследует структуру (все элементы данных) и поведение (все методы) всех базовых классов (см. п. п. 10.1/4 в [1]). Так как при множественном наследовании предков больше одного, возникает интересный вопрос: выполняется ли принцип подстановки при открытом множественном наследовании и каким образом? Пусть у нас определены два базовых класса и наследник от них (листинг 10.1).

Листинг 10.1. Принцип подстановки при множественном наследовании

```
class B1                                // базовый класс  
{ int x;                               // поле недоступно в наследнике  
public:  
    B1(const int &x = 0)                // конструктор инициализации и по умолчанию  
    :x(x)  
    { cout << "B1-init " << endl; }  
    B1(const B1 &b)                      // конструктор копирования  
    { cout << "B1-copy " << endl; }  
    virtual void print()                // вывод  
    { cout << "B1=" << x << endl; }  
};  
class B2                                // второй базовый класс  
{ protected:  
    string s;                           // поле доступно в наследнике  
public:  
    B2(const string &s = "Default"):s(s)  
    { cout << "B2-init " << endl; }
```

```

    B2(const B2 &b)
    { cout << "B2-copy " << endl; }
    virtual void print()           // вывод
    { cout << "B2=" << s << endl; }
};
class D: public B1, public B2 // наследник
{ double y;
public:
    D(const int &x, const string &s, double y=0.0)
    :B1(x), B2(s),                // вызов конструкторов базового класса
    ,y(y)
    { cout << "D-init " << endl; }
    D(const D &b)
    { cout << "D-copy " << endl; }
};

```

Во-первых, с конструкторами при множественном наследовании нужно поступать так же, как и при одиночном: если конструкторы в базовых классах определены, то и в классе-наследнике должен быть определен конструктор. При этом конструкторы-инициализаторы базового класса нужно явно вызвать в конструкторе класса-потомка.

Во-вторых, принцип подстановки выполняется, что демонстрирует следующий фрагмент программы:

```

B1 b1;
B2 b2("b2");
D d(1,"1-1");
b1 = d;                // базовый = производный
b2 = d;                // базовый = производный
B2 bd(d);              // базовый <- производный
B1 dd(d);              // базовый <- производный

```

Объявлено три объекта: b1, b2 — объекты базового класса, d — объект класса-потомка. Принцип подстановки работает в данном случае для обоих базовых объектов в присваивании (не забывайте о срезке):

```

b1 = d;
b2 = d;

```

Обратное присваивание, естественно, не проходит: объект базового класса, как мы знаем, по умолчанию нельзя присвоить объекту производного класса:

```

d = b1;
d = b2;

```

Принцип подстановки мы наблюдаем и в конструкторах копирования базового класса:

```

B2 bd(d);
B1 dd(d);

```

Следующие объявления, как и положено, вызывают ошибку трансляции:

```

D pp(b1);
D tt(b2);

```

В главе 9 мы рассматривали вопрос об универсализации контейнера на основе создания однокоренной иерархии классов от общего базового класса `TObject`. Такая иерархия имеет существенное ограничение — универсальный контейнер «не принимает» данные «посторонних» типов. Допустим, мы хотим, чтобы контейнер позволял хранить также объекты класса `TClass`, который не является наследником от общего базового класса `TObject`. Такая ситуация часто встречается при использовании внешних библиотек. Решить эту проблему помогает как раз множественное наследование: достаточно определить новый класс-наследник от обоих классов, например:

```
class TOClass: public TObject, public TClass
{ /* ... */ }
```

Методы класса `TOClass`, естественно, делегируют работу методам класса `TClass`. Так как принцип подстановки выполняется, то объекты `TOClass` могут теперь храниться в нашем универсальном контейнере.

Неоднозначность

Так же как и виртуальные функции, множественное наследование — это мощный инструмент, и применять его нужно аккуратно. Основная проблема, возникающая при множественном наследовании, — неоднозначность (см. п. 10.2 в [1]), которая проявляется в двух видах: в виде неоднозначности данных и в виде неоднозначности методов. Пусть, например, в показанном примере (см. листинг 10.1) в защищенной (чтобы наследник мог пользоваться) части класса `B1` тоже определено поле `s`. Тип его может быть каким угодно, не обязательно строковым. Тогда при трансляции возникает конфликт имен в функции `print()` класса `D` — компилятор не знает, какое из полей используется.

Аналогичная картина с методами. Например, пусть в базовых классах `B1` и `B2` определен некоторый метод:

```
void f() { /*...*/ }
```

Класс `D` унаследует обе версии этого метода, и при попытке вызвать `f()` возникает ошибка неоднозначности.

Ситуация еще более усугубляется, если у нас в программе наблюдается так называемое «ромбовидное» наследование (рис. 10.1).

```
class A { /*...*/ };
class B1: public A { /*...*/ };
class B2: public A { /*...*/ };
class D: public B1, public B2 { /*...*/ };
```

В этом случае класс `D` получает по 2 экземпляра всех полей и всех методов общего предка — класса `A`. В этом легко убедиться с помощью функции `sizeof()`.

Но в таком виде проблема неоднозначности относительно легко разрешается с помощью префикса. Например, в функции вывода `print()` достаточно явно указать, какое поле мы собираемся выводить: `B1::s` или `B2::s`. Вызов функции `f()` тоже квалифицируется префиксом: `B1::f()` или `B2::f()`.

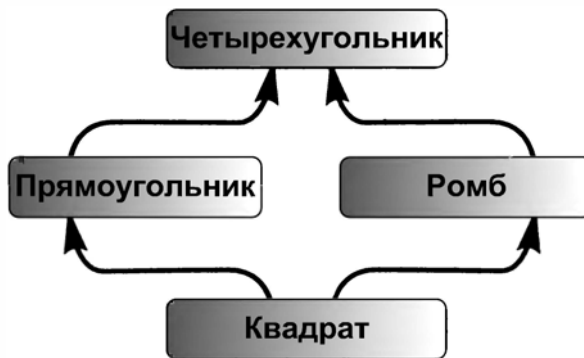


Рис. 10.1. Ромбовидное наследование

С виртуальными функциями происходит то же самое. Пусть в обоих базовых классах определена виртуальная функция `print()`. Класс-потомок унаследует обе. Если в наследнике не определена собственная виртуальная функция с таким же прототипом, то следующий вызов является неоднозначным:

```
D d(1, "1-1", 1.2);  
d.print();
```

То есть вызов виртуальной функции тоже должен быть квалифицирован именем класса: `B1::print()` или `B2::print()`. Однако в таком случае вызов перестает быть виртуальным!

Таким образом, мы видим, что множественное наследование создает достаточно много проблем, если его применять, не думая о последствиях. Однако «правильное» применение бывает чрезвычайно полезным (например, реализация паттерна Adapter). Многие авторы [2, 21, 23, 25] советуют применять множественное наследование только для классов-интерфейсов, не содержащих полей. Именно такое решение принято в языках Java и C#: для интерфейсов разрешено множественное наследование, а для классов — только одиночное.

Виртуальное наследование

Для устранения неоднозначности данных при ромбовидном наследовании в C++ реализован специальный механизм виртуального наследования. Показанное на рис. 10.1 «ромбовидное» наследование на C++ записывается так (см. п. п. 10.1/5 в [1]):

```
class A { /* ... */ };  
class B1: virtual public A { /* ... */ };  
class B2: virtual public A { /* ... */ };  
class D: public B1, public B2 { /* ... */ };
```

Классы B1 и B2 виртуально наследуют от базового класса A. Класс A является для них *виртуальным базовым классом*. Виртуальное наследование приводит к тому, что в классе D оказывается единственная копия полей класса A. Однако неоднозначности полей классов B1 и B2 это не устраняет, как и неоднозначности их методов.

И как мы уже неоднократно убеждались, за все приходится платить. Множественное наследование превращает древовидную иерархию наследования в ациклический граф наследования. При простом наследовании в дереве наследования от предка к наследнику всегда существует только один путь независимо от количества уровней наследования. При множественном наследовании путей в графе наследования может быть несколько (очевидно, при ромбовидном наследовании их два).

При обычном одиночном наследовании конструктор класса-потомка всегда вызывает конструктор непосредственного базового класса для инициализации полей базового класса. Этот порядок сохраняется на любом уровне иерархии. В обычном (не виртуальном) множественном наследовании работает то же правило. Если же мы рассмотрим с этой точки зрения тривиальную ромбовидную иерархию с виртуальным наследованием, то тут же возникает вопрос: какой из классов (B1 или B2) отвечает за инициализацию единственной копии общего базового класса? Опять неоднозначность!

Эта неоднозначность устраняется следующим правилом: виртуальный базовый класс инициализируется *последним производным классом* в иерархии. Данное правило поддерживается компилятором. Чтобы убедиться в том, что инициализацию виртуальной базы действительно обязан выполнять последний производный класс, напомним пример с виртуальным наследованием, но вызовы конструкторов пропишем так, как при обычном наследовании: в конструкторе наследника будет вызываться конструктор базового класса (листинг 10.2).

Листинг 10.2. Инициализация виртуального базового класса

```
class A
{ int a;
  public:
    A(const int &a):a(a) {}           // конструктор инициализации
};
class B1: virtual public A           // виртуальное наследование
{ int b;
  public:
    B1(const int &a, const int &b):A(a) // инициализация предка
    { this->b = b; }
};
class B2: virtual public A           // неоднозначность с B1
{ int b;
  public:
    B2(const int &a, const int &b):A(a) // инициализация предка
    { this->b = b; }
};
class D: public B1, public B2
{ double y;
  public:
    D(int b1, int b2, double y)
    :B1(0,b1), B2(0,b2)              // инициализация предков
    { this->y = y; }                 // ошибка трансляции!
};
```

Попытка откомпилировать эту иерархию классов в системе Visual C++.NET 2003 приводит к ошибке трансляции C2512 в конструкторе класса D:

```
A::A: no appropriate default constructor available
```

Сообщение говорит об отсутствии подходящего конструктора A по умолчанию. Причина в том, что мы не написали вызов конструктора виртуальной базы A в классе D — самом производном классе. Следовательно, транслятор предполагает, что в D необходимо «молча» вызвать конструктор A по умолчанию. Однако в классе A такого нет — отсюда и ошибка! Таким образом, класс D отвечает за инициализацию класса A. Правильная «конструкция» конструктора D должна быть такой, как показано в листинге 10.3.

Листинг 10.3. Правильный конструктор самого производного класса

```
D(int a, int b1, int b2, double y)
:A(a), B1(0,b1), B2(0,b2)      // инициализация виртуальной базы
{ this->y = y; }
```

Естественно, и «свое» поле конструктор может инициализировать в списке инициализации. Обратите внимание на то, что в конструкторах B1 и B2 первый аргумент (значение для поля b виртуальной базы) равен нулю. Этот аргумент в данном случае является фиктивным, поскольку у нас есть явный вызов конструктора виртуальной базы A(a). Транслятор правильно во всем разбирается, и поле a инициализируется значением первого аргумента конструктора D, а не нулями.

Принцип доминирования

В некоторых случаях, которые представляются неоднозначными, неоднозначности фактически нет. Рассмотрим следующую простую иерархию (листинг 10.4).

Листинг 10.4. Демонстрация принципа доминирования

```
class Up
{ public:
    virtual ~Up() {};
    virtual void Pttint() { cout << "Up!" << endl; }
};
class Left: virtual public Up
{ public:      // метод Print переопределен
    virtual void Pttint() { cout << "Left!" << endl; }
};
class Right: virtual public Up
{ // метод Print унаследован
};
class Down: public Left, public Right
{ };
```

Класс Down унаследовал метод Print() непосредственно от класса Left, а также метод Print() от класса Up «транзитом» через класс Right. Вроде бы налицо неоднозначность в ситуации:

```
Down a;
a.Print();
```

Однако в C++ определено правило *доминирования* (см. п. п. 10.2/6 в [1]), согласно которому в данной ситуации будет вызываться функция из класса `left`. В ситуациях, подобных данной, компилятор выбирает «наиболее производное» имя, которое считает доминирующим. Работает это правило только при виртуальном наследовании.

Финальный класс

Виртуальное наследование позволяет запретить наследование от нашего класса. О том, что это не просто интересная «фишка», а реально необходимый инструмент, говорит тот факт, что в Java введено специальное ключевое слово `final` (финальный). Если класс объявлен со спецификатором `final`, то пользователь не сможет от него наследовать. Класс `String` в библиотеке классов Java объявлен как финальный.

В C++ нет специальных ключевых слов, запрещающих наследование, поэтому объявление класса «финальным» основано на управлении доступом к специальным функциям класса: конструкторам и деструктору. Например, если деструктор в базовом классе объявить закрытым, то будет невозможно объявить переменную-объект ни базового, ни производного классов, например:

```
class OnlyDynamic
{ ~OnlyDynamic();
  public:
  // ...
};
class Derived: public OnlyDynamic
{ };
```

Наследование формально не запрещается (хотя Visual C++.NET 2003 выдает предупреждение), однако попытки объявить переменную следующим образом вызывают ошибку компиляции:

```
Derived Object; // ошибка!
```

Тем не менее такое определение базового класса не мешает нам создать наследника в динамической памяти, например:

```
Derived *p = new Derived;
delete p;
```

Как указано в [43], используя виртуальное наследование, можно вообще «запретить» наследование (а фактически — создание объектов класса-наследника). Обнаружил эту интересную особенность виртуального наследования Эндрю Кениг. Простейший пример показан в листинге 10.5.

Листинг 10.5. Финальный класс

```
class Lock // "запирающий" класс
{ Lock();
  Lock(const Lock&);
  friend class Usable;

public:
};

// наследник - "друг"
```

```
class Usable: public virtual Lock           // "финальный" класс
{ public:
    Usable() { cout << "Usable!"; }
};
class Derived: public Usable {};           // формально не запрещено
```

В этом случае мы имеем «внутреннюю» иерархию из двух классов:

- запирающий класс — виртуальная база, у которого конструкторы являются приватными;
- виртуальный наследник — финальный класс, от которого нельзя наследовать.

Хотя формально наследование не запрещено, попытки создать объект класса `Derived` оканчиваются неудачей на этапе трансляции:

```
Usable u;                               // нет ошибки-
Derived d;                               // ошибка трансляции
Derived *pd = new Derived;               // ошибка трансляции
```

Для объектов типа `Derived` выдается сообщение об отсутствии доступа к приватным конструкторам класса `Lock`.

Размеры классов при множественном наследовании

Интересно выяснить, какие размеры имеет класс-потомок при обычном и виртуальном множественном наследовании. Начнем с пустых классов и будем добавлять туда поля и придавать им виртуальность¹:

```
class A{ };
class B1: public A { };
class B2: public A { };
class D: public B1, public B2 { };
cout << sizeof(A) << endl;
cout << sizeof(B1) << endl;
cout << sizeof(B2) << endl;
cout << sizeof(D) << endl;
```

На экран, естественно, выводится четыре единицы — размеры пустых классов. Добавим поле `short` в класс `A`:

```
class A { short a; };
class B1: public A { };
class B2: public A { };
class D: public B1, public B2 { };
```

Получим цифры 2, 2, 2, 4 — это как раз говорит о том, что поле базового класса `A` класс-наследник `D` унаследовал дважды: от `B1` и от `B2`. Добавим в класс `A` поле типа `char`, чтобы его длина была равна трем, а наследование сделаем виртуальным:

```
class A { short a; char b; };
class B1: virtual public A { };
```

¹ Я думаю, мы достаточно много сравнивали Visual C++.NET 2003 и C++Builder 6, поэтому ограничусь только первой системой в режиме `#pragma pack(1)`.


```
class B2: virtual public A { };
class D: public B1, public B2 { };
```

Тогда получим размеры 3, 7, 7, 11. Очевидно, виртуальное наследование добавило в классы B1 и B2 по 4 байта. Можно предположить, что это — размер указателя (мы работаем на платформе Intel под Windows). Тогда класс D получил 3 байта от базового класса и 2 указателя (от B1 и от B2).

Теперь один класс унаследует виртуально, а другой — обычно:

```
class A { short a; char b; };
class B1: public A { };
class B2: virtual public A { };
class D: public B1, public B2 { };
```

В результате получаем 3, 3, 7, 10. Это означает следующее: класс A имеет размер 3 байта; класс B1 (простой наследник) унаследовал эти 3 байта; виртуальный наследник B2 имеет размер $3 + 4 = 7$ байт, а последний производный класс получил размер 3 (от A) + 3 (от B1) + 4 (указатель от B2) = 10 байт.

При виртуальном наследовании добавим в класс A виртуальный деструктор:

```
class A { short a; char b; virtual ~A() {} };
class B1: virtual public A { };
class B2: virtual public A { };
class D: public B1, public B2 { };
```

Размеры классов, естественно, увеличиваются на 4 байта — размер указателя на таблицу виртуальных функций: 7, 11, 11, 15.

Таким образом, мы наблюдаем, что на этапе выполнения программы виртуальное наследование увеличивает накладные расходы.

RTTI

В главе 9 при обсуждении виртуальных методов мы уже упоминали механизм RTTI, с помощью которого можно определить тип объекта во время выполнения, имея только указатель на него. Механизм динамической идентификации типа состоит из трех составляющих:

- оператора динамического преобразования типа `dynamic_cast<>` (см. п. п. 5.2.7 в [1]);
- оператора идентификации точного типа объекта `typeid()` (см. п. п. 5.2.8 в [1]);
- класса `type_info` (см. п. п. 18.5.1 в [1]);

Оператор `dynamic_cast<>` допускается применять к указателям только на полиморфные классы (содержащие хотя бы одну виртуальную функцию). Вообще-то любой класс легко сделать полиморфным, определив для него виртуальный деструктор. Оператор имеет следующий формат:

```
dynamic_cast<тип *>(указатель)
```

Результатом является преобразованный указатель. Если преобразование указателя к нужному типу выполнить не удастся, оператор возвращает нулевой указатель. Следовательно, результат преобразования нужно всегда проверять.

Преобразования допускаются только между «родственниками», то есть классами, входящими в одну иерархию наследования (тип указателя в круглых скобках должен быть родственным типу в угловых скобках). Преобразование может быть:

- повышающим — от производного класса к базовому;
- понижающим — от базового класса к производному;
- перекрестным — от одного производного класса к другому.

Как мы знаем, повышающее преобразование выполняется обычно по умолчанию посредством подстановки (см. главу 8). Тем не менее можно его выполнить и явно. Понижающее и перекрестное преобразования выполняются только с помощью оператора `dynamic_cast<>`.

Оператор `dynamic_cast<>` можно применять и к ссылкам. Формат оператора в этом случае такой:

```
dynamic_cast<тип &>(ссылка)
```

Все условия относительно «родственности» полиморфных классов должны выполняться и в этом случае, однако обработка аварийного случая происходит по-другому. Так как нулевых ссылок не бывает, при невозможности преобразования ссылки генерируется исключение `bad_cast` (см. п. п. 18.5.2 в [1]).

Если оператор `dynamic_cast<>` выполняет преобразование одного типа в другой, то оператор `typeid()` позволяет выяснить фактический тип класса. Для использования этого оператора необходимо включить в программу заголовок

```
#include <typeinfo>
```

В этом классе представлен класс `type_info`, который возвращается в качестве результата оператором `typeid()`. В стандарте (см. п. п. 18.5.1 в [1]) класс `type_info` определен так, как показано в листинге 10.6.

Листинг 10.6. Класс `type_info`

```
class type_info
{ public:
    virtual ?type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
};
```

Как видите, объекты типа `type_info` невозможно ни создать, ни скопировать — конструкторы и операция присваивания закрыты. Объекты типа `type_info` можно сравнивать между собой — и это основные операции, которые используются совместно с оператором `typeid()`; можно получить имя типа с помощью метода `name()`. Как написано в [2], метод `before()` «позволяет сортировать информацию о типе `type_info`. Нет никакой связи между отношениями упорядочения, определяемыми `before()`, и отношениями наследования».

Оператор `typeid()` имеет две синтаксически одинаковые формы — разница заключается в аргументе:

```
typeid(выражение)
typeid(имя_типа)
```

В качестве выражений наиболее часто применяются указатели. Таким образом, оператор `typeid()` позволяет определить точный тип объекта, на который указывает указатель. Если указатель нулевой, то возбуждается исключение `bad_typeid` (см. п. п. 18.5.3 в [1]). В качестве имени типа используются имена классов. Однако в отличие от `dynamic_cast<>`, оператор `typeid()` можно применять к встроенным типам. Полезного в этом мало, но ошибки не вызывает.

Так как оператор `typeid()` возвращает объект типа `type_info`, то можно сравнивать эти объекты и вызывать методы класса `type_info`. В справочной системе C++ Borland 6 есть пример, иллюстрирующий простейший случай использования оператора `typeid()`. Мы его модифицируем и дополним (листинг 10.7).

Листинг 10.7. Простое использование оператора `typeid()`

```
class D{ }; // базовый класс
class B: D { }; // наследник
int main()
{ char C;
  float X;
  if (typeid(C) == typeid(X)) // сравнили типы
      cout << "C и X — одного типа!" << endl;
  else cout << "C и X — разного типа! " << endl;
  cout << typeid(int).name(); // выводит int
  cout << typeid(double).name(); // выводит double
  cout << boolalpha <<
      (typeid(int).before(typeid(double))?true:false)
      << endl; // выводит false
  cout << typeid(D).name(); // выводит D
  cout << typeid(B).name(); // выводит B
  cout << boolalpha <<
      (typeid(D).before(typeid(B))?true:false)
      << endl; // выводит false
  int iobj;
  cout << typeid(iobj).name(); // выводит int
  out << typeid(8.16).name(); // выводит double
}
```

Сам механизм RTTI не является особенно сложным, однако не всегда понятно, когда и как его нужно применять, поскольку виртуальные функции позволяют в большинстве случаев обойтись без RTTI.

Рассмотрим пример. Предположим, мы разрабатываем файловую систему в составе некоей операционной системы с интерфейсом GUI¹ [55, 56, 59]. Для пользователя файлы представляются пиктограммами. Щелкнув на пиктограмме правой кнопкой мыши, мы получаем контекстное меню с командами открытия, закрытия,

¹ Реальным примером такой системы является Windows.

копирования и т. п. Реализация файловой системы основана на иерархии классов, корнем которой является абстрактный класс `File` примерно следующего вида:

```
class File
{ public:
    virtual void open() = 0;
    virtual void close() = 0;
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~File() = 0;
};
File::~File(){}           // чистый виртуальный деструктор требует реализации
```

Пусть в системе в первой версии существуют текстовые файлы и двоичные исполняемые файлы. В иерархии классов эти файлы представлены классами-наследниками от базового абстрактного класса, в которых реализованы чистые виртуальные функции. Например, класс `BinaryFile` определяет двоичный исполняемый файл:

```
class BinaryFile: public File
{ public:
    void open() { Execute(this); }
    // другие методы
};
```

Класс `TextFile` определяет тестовый файл:

```
class TextFile: public File
{ public:
    void open() { Activate_text_processor(this); }
    virtual void print();
    // другие методы
};
```

Двоичные файлы, естественно, отличаются от текстовых во многих деталях. Например, при открытии двоичного файла он запускается на выполнение, а при открытии текстового файла запускается текстовый редактор и ему передается текущий открываемый файл. Поскольку у нас функции открытия виртуальные, то проблем с правильным открытием не должно возникать. Однако различие проявляется и в том, что текстовые файлы можно печатать, а двоичные — нет. Следовательно, в классе `TextFile` существует специфический для этого типа файлов метод `print()`, которого нет в классе `BinaryFile`.

Щелчок правой кнопкой мыши на пиктограмме файла должна обрабатывать API-функция, которая открывает упомянутое контекстное меню. Заголовок функции может выглядеть так:

```
OnRightClick(File &file);
```

Функция принимает параметр базового класса, следовательно, в соответствии с принципом подстановки, может принимать в качестве аргумента как текстовый, так и двоичный файлы. Однако наши файлы не одинаковы: контекстное меню текстового файла должно отличаться от контекстного меню двоичного файла, по крайней мере, наличием команды печати. Следовательно, функция

`OnRightClick()` должна «уметь» различать типы файлов во время работы! Это можно обеспечить с помощью оператора `typeid()`. Структура функции может быть такой:

```
OnRightClick(File &file)
{ if (typeid(file) == typeid(TextFile))
  { // обработка текстового файла
  }
  else
  { // обработка двоичного файла
  }
}
```

Хотя статическим типом аргумента `file` является тип `File`, на место объекта базового класса может быть подставлен объект производного класса. Поэтому оператор `typeid()` возвращает объект `type_info`, который несет информацию о реальном типе передаваемого аргумента. Оператор `if` проверяет, совпадает ли реальный тип аргумента `file` с типом `TextFile`.

Однако одного оператора `typeid()` может оказаться недостаточно. Пусть у нас в системе появляется третий тип файла: HTML-файл. Так как HTML-файл является текстовым файлом, то, естественно, класс `HTMLFile` является наследником класса `TextFile`:

```
class HTMLFile: public TextFile
{ public:
    void open() { Activate_Brouser(this); }
    virtual void print();
    // другие методы
};
```

Реализация функции открытия, естественно, отличается от реализации функции открытия базового текстового файла: для HTML-файла вызывается программа-браузер¹. Функция печати, очевидно, тоже должна быть реализована по-другому.

В функции `OnRightClick()` у нас возникают проблемы, так как при передаче аргумента типа `HTMLFile` она будет работать по ветке двоичного файла. А все потому, что значение `typeid(HTMLFile)` не равно значению `typeid(TextFile)`, хотя типы `HTMLFile` и `TextFile` — ближайшие «родственники». В такой ситуации нам и может пригодиться оператор `dynamic_cast<>`. Вместо ссылки будем передавать в функцию `OnRightClick()` указатель. Тогда функция приобретает следующий вид:

```
OnRightClick(File *file)
{ TextFile *pFile = dynamic_cast<TextFile *>(file);
  if (pFile)
  { // обработка текстового файла и HTML-файла
  }
  else
  { // обработка двоичного файла
  }
}
```

¹ Не забывайте, что в базовом абстрактном классе функция открытия определена как чистая виртуальная, поэтому во всех производных классах она тоже является виртуальной, хотя это явно и не указано.

Если в функцию будет передан указатель на `HTMLFile`, то он успешно преобразуется к типу указателя на `TextFile`. В этом случае указатель `pFile` оказывается ненулевым. Если же при вызове функции будет передаваться аргумент-указатель на двоичный файл, то преобразование не выполнится и указатель `pFile` станет нулевым.

Немного другой вид функция приобретает при передаче параметра-ссылки:

```
OnRightClick(File &file)
{ try {
    TextFile pFile = dynamic_cast<TextFile &>(file);
    // обработка текстового файла и HTML-файла
}
catch(std::bad_cast &noTextFile)
{ // обработка двоичного файла
}
}
```

Если преобразование ссылки обошлось без генерации исключения, значит, имеем в наличии текстовый файл. Если же ссылку преобразовать не удалось, то файл, очевидно, не является текстовым — возникает исключение, и мы его перехватываем. В секции-ловушке выполняется обработка двоичного файла.

Перекрестные ссылки можно делать, например, так:

```
struct A
{ double d;
  virtual ~A(){};    // чтобы сделать класс полиморфным
};
struct B
{ double r;
  bool b;
};
struct D: public A, public B
{ int k;
  D() { b=true; d=r=0.0; k=1; }
};
A *pa = new D();
B *pb = dynamic_cast<B*>(pa);
```

Статический тип `pa` — указатель на `A`, динамический — указатель на `D`. Обычный `static_cast<>` не может преобразовать указатель на `A` в указатель на `B`, так как классы `A` и `B` независимы. Можно использовать оператор `reinterpret_cast<>` или преобразование в стиле `C (B*)` — трансляция пройдет без ошибок, однако результаты работы будут неправильными. В этом легко убедиться, если выполнить простой оператор вывода:

```
cout << pb->b << endl;
```

Так как поле `b` в конструкторе `D()` инициализируется как `true`, на экране должна появиться цифра 1. Однако при использовании оператора `reinterpret_cast<>` выводится нуль. В то же время при использовании оператора `dynamic_cast<>` все работает правильно. В этой проблеме легко разобраться, если проверить

значения указателей, в том и другом случаях: при использовании оператора `reinterpret_cast<>` адреса, содержащиеся в `ra` и `rb`, — одинаковы, а при использовании оператора `dynamic_cast<>` — разные. Это объясняется тем, что в объекте типа `D` объекты типа `A` и `B` занимают разные места. Оператор `dynamic_cast<>` правильно вычисляет адрес подобъекта типа `B` внутри `D`, а оператор `reinterpret_cast<>` этого не делает.

Понижающее приведение с помощью оператора `dynamic_cast<>` выполняется обычно, чтобы преобразовать указатель на виртуальную базу в указатель на один из дочерних классов, например:

```
struct V
{ virtual ~V() {}; // полиморфный класс - виртуальная база
};
struct A: virtual V {};
struct B: virtual V {};
struct D: public A, public B {};
V *pv = new D;
A *pa = dynamic_cast<A*>(pv); // понижающее приведение
```

Указатель `pv` имеет статический тип `V`, а указатель `pa` — статический тип `A`; преобразование выполняется от базы к наследнику. Это возможно вследствие того, что динамический тип указателя `pv` — тип `D`, а значит, в динамике выполняется повышающее приведение от `D` к `A`.

Мультиметоды

Мультиметод — это виртуальная функция, выбираемая при выполнении на основании типа нескольких аргументов [26]. Как сказано в главе 9, в C++ есть два вида полиморфизма: статический и динамический. Статический полиморфизм реализуется перегрузкой функций и шаблонами функций. Виртуальные функции являются воплощением механизма динамического полиморфизма.

К сожалению, два этих вида полиморфизма реализованы в C++ совсем не одинаково. Например, при перегрузке нужный вариант функции выбирается на основании всего списка параметров — они совершенно равноправны. Выбор же виртуальной функции производится по единственному — левому — аргументу. Даже синтаксис вызова виртуальной функции отдает предпочтение объекту `obj` перед другими аргументами:

```
obj.function(аргументы)
```

Говорят, что виртуальная функция способна выполнить *одиночную диспетчеризацию*. Мультиметоды — это инструмент *множественной диспетчеризации*. Наиболее распространенный вариант — двойная диспетчеризация, которая означает, что нужная функция выбирается на основании типов двух аргументов. Александреску в [20] называет двойную диспетчеризацию двойным переключением по типу.

Мультиметоды могут понадобиться в ситуациях, когда требуется выполнять некоторую операцию с различными сочетаниями типов параметров. Например,

Элджер в [25] приводит пример с операцией сложения различных чисел: целых, действительных, комплексных. Страуструп Б. в [43] описывает ситуацию, когда нужно определять пересечение различных геометрических фигур, Брюс Эккель в [11] рассматривает операцию умножения между матрицами, векторами и скалярами, а Скотт Мейерс в [24] — сталкивающиеся объекты космической игры.

Использование RTTI

Покажем несколько схем реализации мультиметодов. Обычно классы, которые участвуют в работе мультиметодов, являются «родственниками», однако в некоторых случаях возможна реализация и между неродственными классами. Пусть у нас есть базовый абстрактный класс `Base` и три наследника: `Derived_a`, `Derived_b` и `Derived_c`. Нам требуется выполнять некую операцию со всеми возможными сочетаниями параметров. В базовом классе определяется чистый абстрактный метод, который реализуется в производных классах. В общем случае для трех типов, `A`, `B` и `C`, число сочетаний любых двух типов равно девяти:

1. `Derived_a` — `Derived_a`.
2. `Derived_a` — `Derived_b`.
3. `Derived_a` — `Derived_c`.
4. `Derived_b` — `Derived_a`.
5. `Derived_b` — `Derived_b`.
6. `Derived_b` — `Derived_c`.
7. `Derived_c` — `Derived_a`.
8. `Derived_c` — `Derived_b`.
9. `Derived_c` — `Derived_c`.

В конкретных ситуациях сочетаний бывает меньше, так как операция может оказаться коммутативной для некоторых сочетаний. Например, операция умножения скаляра и вектора — коммутативна, так же как и операция умножения скаляра на матрицу. Метод, выполняющий двойное переключение по типу, реализуется с параметрами базового класса — используется принцип подстановки. Двойное переключение эмулируется при помощи цепочек `if...else` с использованием преобразования `dynamic_cast<>`. Покажем сначала реализацию для двух классов (листинг 10.8), а потом добавим третий.

Листинг 10.8. Двойной диспетчер для двух типов

```
// определения классов
class Base
{ public:
    virtual ~Base()=0;
    virtual bool Operator(const Base &R) = 0;    // требуемая операция
};
Base::~Base(){}                                // определение деструктора
class Derived_a: public Base
```

продолжение ➤

Листинг 10.8 (продолжение)

```

{ public:
    virtual bool Operator(const Base &R);
};
class Derived_b: public Base
{ public:
    virtual bool Operator(const Base &R);
};
// реализация методов двойной диспетчеризации
bool Derived_a::Operator(const Base &R)
{ if (const Derived_a *pra = dynamic_cast<const Derived_a*>(&R))
    { // правый аргумент типа A; обработка варианта "A-A"
        cout << "A-A" << endl;
        return true;
    }
    else if (const Derived_b *prb = dynamic_cast<const Derived_b*>(&R))
    { // правый аргумент типа B; обработка варианта "A-B"
        cout << "A-B" << endl;
        return true;
    }
    else throw exception ("Error! Incorrect type argument!");
}
bool Derived_b::Operator(const Base &R)
{ if (const Derived_a *pra = dynamic_cast<const Derived_a*>(&R))
    { // правый аргумент типа A; обработка варианта "B-A"
        cout << "B-A" << endl;
        return true;
    }
    else if (const Derived_b *prb = dynamic_cast<const Derived_b*>(&R))
    { // правый аргумент типа B; обработка варианта "B-B"
        cout << "B-B" << endl;
        return true;
    }
    else throw exception ("Error! Incorrect type argument!");
}

```

Мультиметоды устроены достаточно просто: проверяется правый операнд, и в зависимости от его типа выполняется обработка. Если же правый операнд неизвестного типа, то генерируется исключение. Добавим третий класс и покажем механизм использования оператора typeid() для распознавания типа (листинг 10.9).

Листинг 10.9. Мультиметод нового дополнительного класса

```

class Derived_c: public Base
{ public:
    virtual bool Operator(const Base &R);
};
bool Derived_c::Operator(const Base &R)
{ if (typeid(Derived_a) == typeid(R))
    { // правый аргумент типа A; обработка варианта "C-A"
        cout << "C-A" << endl;
        return true;
    }
}

```

```

else if (typeid(Derived_b) == typeid(R))
{ // правый аргумент типа B; обработка варианта "C-B"
  cout << "C-B" << endl;
  return true;
}
else if (typeid(Derived_c) == typeid(R))
{ // правый аргумент типа C; обработка варианта "C-C"
  cout << "C-C" << endl;
  return true;
}
else throw exception ("Error! Incorrect type argument!");
}

```

Если требуется, чтобы выполнялась обработка объектов типа C методами классов A и B, нужно модифицировать их, добавив проверку правого аргумента на совпадение с типом C. Но делать это в обязательном порядке нет необходимости — все определяется конкретной задачей. Таким образом, у нас реализовано семь сочетаний из девяти, в чем можно убедиться, выполнив такой фрагмент программы:

```

Derived_a A; Derived_b B; Derived_c C;
try {
  cout << A.Operator(A) << endl;
  cout << A.Operator(B) << endl;
  cout << B.Operator(A) << endl;
  cout << B.Operator(B) << endl;
  cout << C.Operator(A) << endl;
  cout << C.Operator(B) << endl;
  cout << C.Operator(C) << endl;
  cout << A.Operator(C) << endl; // вызывает исключение
}
catch(exception &e)
{ cout << e.what() << endl; }

```

Последний оператор вызывает исключение, так как сочетание `Derived_a` — `Derived_c` не реализовано.

Задачу мы решили — реализовали двойное переключение по типу. Однако даже начинающий программист понимает, что у этого решения есть недостатки. Например, задействован механизм RTTI, который не является самым эффективным в C++. Но это не самое важное. Гораздо хуже, что нам при добавлении нового класса приходится модифицировать тексты методов. Представьте, что классов у нас хотя бы 7–8, и методов, соответственно, тоже. То есть в каждом методе требуется выполнять до семи–восьми однотипных проверок. Добавление нового класса может превратиться в кошмар, поэтому поищем другие пути реализации двойной диспетчеризации.

Использование только виртуальных функций

Есть очень нетривиальный способ реализовать двойное переключение по типу с использованием только виртуальных функций. Как обычно, сначала реализуем мультиметоды для двух классов (листинг 10.10), а затем попытаемся добавить

третий. Основная идея состоит в том, чтобы реализовать двойную диспетчеризацию как две одиночные, выполняемые последовательно; первый виртуальный вызов определяет динамический тип первого объекта, а второй — второго объекта.

Листинг 10.10. Мультиметоды — только виртуальные функции

```
// предварительное объявление наследников
class Derived_a;
class Derived_b;
class Derived_c;
// абстрактный базовый класс
class Base
{ public:
    virtual ~Base()=0;
    virtual bool Operator(const Base &R)const = 0;
    virtual bool Operator(const Derived_a &R)const = 0;
    virtual bool Operator(const Derived_b &R)const = 0;
};
Base::~Base(){}
// определение классов-наследников
class Derived_a: public Base
{ public:
    virtual bool Operator(const Base &R) const
    { return R.Operator(*this); }; // перенаправление виртуальности
    virtual bool Operator(const Derived_a &R)const
    { cout << "A-A" << endl; return true; }
    virtual bool Operator(const Derived_b &R)const
    { cout << "A-B" << endl; return true; }
};
class Derived_b: public Base
{ public:
    virtual bool Operator(const Base &R)const
    { return R.Operator(*this); }; // перенаправление виртуальности
    virtual bool Operator(const Derived_a &R)const
    { cout << "B-A" << endl; return true; }
    virtual bool Operator(const Derived_b &R)const
    { cout << "B-B" << endl; return true; }
};
```

Обратите внимание на реализацию первой функции, которая абсолютно одинакова в обоих классах, — именно она реализует двойное переключение по типу:

```
virtual bool Operator(const Base &R) const
{ return R.Operator(*this); };
```

Чтобы было понятно, где и как она работает, рассмотрим фрагмент программы:

```
Derived_a A; Derived_b B; // нет необходимости в виртуальности
cout << A.Operator(A) << endl;
cout << A.Operator(B) << endl;
cout << B.Operator(A) << endl;
cout << B.Operator(B) << endl;
Base &D = A;
cout << A.Operator(D) << endl; // работают 2 виртуальных вызова
cout << B.Operator(D) << endl; // работают 2 виртуальных вызова
```

```
cout << D.Operator(D) << endl;    // работают 2 виртуальных вызова
cout << D.Operator(A) << endl;    // работает 1 виртуальный вызов
cout << D.Operator(B) << endl;    // работает 1 виртуальный вызов
```

В первых четырех вызовах нет необходимости в двойной диспетчеризации, так как нет подстановки ссылки на наследника вместо ссылки на базовый класс — вызываются сразу нужные функции. Эти функции и выводят на экран строки, соответствующие типам операндов:

```
A-A
A-B
B-A
B-B
```

Следующие три вызова как раз и выполняют двойную диспетчеризацию. В первом варианте мы сначала попадаем в первый метод класса `Derived_a`, так как тип левого аргумента именно этот. При этом на место параметра-ссылки `R` подставляется фактический аргумент-ссылка `D`, инициализированный ссылкой `A` типа `Derived_a`, а `*this` внутри метода статически имеет тип `Derived_a`. Таким образом, вызов превращается в `A.Operator(A)`, и на экран выводится строка «A-A». Второй вариант вызова сначала приводит нас к первой функции класса `B`, так как левый аргумент вызова имеет тип `Derived_b`. Снова на место ссылки `R` подставляется ссылка `D`, инициализированная ссылкой типа `Derived_a`. Однако мы находимся внутри класса `Derived_b`, поэтому `*this` имеет тип `Derived_b` и вызов превращается в `A.Operator(B)`, что мы и наблюдаем при выводе на экран строки «A-B». В третьем случае сначала виртуализируется левый аргумент, и вызов направляется в класс `Derived_a`, а дальше процесс продолжается по первому варианту, и на экран выводится строка «A-A».

Четвертый виртуальный вызов превращается в `A.Operator(A)`, а пятый — в `A.Operator(B)`. Виртуальность работает только один раз — для левого аргумента.

Теперь сделаем то же самое с указателями:

```
Base *pA = new Derived_a();
Base *pB = new Derived_b();
cout << pA->Operator(*pA) << endl;
cout << pA->Operator(*pB) << endl;
cout << pB->Operator(*pA) << endl;
cout << pB->Operator(*pB) << endl;
```

Первый вариант, естественно, сначала приводит нас в первый метод класса `Derived_a`, который вызывает второй метод того же класса с аргументом того же типа. Второй вызов тоже сначала сопоставляется с первым методом класса `Derived_a`, но поскольку в этом случае на место `R` подставляется ссылка на объект типа `Derived_b`, а `*this` имеет тип `Derived_a`, то вызывается метод с прототипом

```
virtual bool Operator(const Derived_a &R) const;
```

На экране, естественно, получаем строку «B-A». Третий и четвертые варианты, естественно, первый вызов осуществляют из класса `Derived_b`, а второй — в соответствии с аргументом `*this`.

Таким образом, методы с параметром типа `Base` осуществляют то самое двойное переключение по типу.

Этот способ реализации мультиметодов имеет довольно значительные преимущества по сравнению с предыдущим. Во-первых, нет необходимости генерировать исключение при неправильных типах аргументов, поскольку все проверки осуществляет компилятор. Во-вторых, обратите внимание на значительное упрощение текста методов — реализуется только фактически необходимая работа без всяких лишних проверок.

Теперь добавим третий класс и посмотрим, какие изменения придется внести в исходные классы:

```
class Derived_c: public Base
{ public:
    virtual bool Operator(const Base &R)const
    { return R.Operator(*this); }
    virtual bool Operator(const Derived_a &R)const
    { cout << "C-A" << endl; return true; }
    virtual bool Operator(const Derived_b &R)const
    { cout << "C-B" << endl; return true; }
    virtual bool Operator(const Derived_c &R)const
    { cout << "C-C" << endl; return true; }
};
```

Этот класс реализует нам операцию с левым операндом типа `Derived_c`. Отметим, что ни класс `Derived_a`, ни класс `Derived_b` не работают с аргументом типа `Derived_c`. Не будем вносить никаких изменений в эти классы и выполним небольшой фрагмент программы:

```
Derived_a A; Derived_b B; Derived_c C;
Base &rA = A;
Base &rB = B;
Base &rC = C;
cout << A.Operator(C) << endl;
cout << B.Operator(C) << endl;
cout << rA.Operator(rC) << endl;
cout << rB.Operator(rC) << endl;
```

Никаких ошибок компиляции нет! Несмотря на то, что мы не определяли в классах `Derived_a` и `Derived_b` операции с аргументом типа `Derived_c`, следующие вызовы не вызывают ошибок компиляции:

```
cout << A.Operator(C) << endl;
cout << B.Operator(C) << endl;
```

Причина в том, что в данном случае работает принцип подстановки. Ввиду отсутствия метода с аргументом типа `Derived_c` выбирается метод класса с параметром-ссылкой базового типа и на его место подставляется ссылка на наследника. Далее выполняется механизм двойной диспетчеризации. Точно так же работают вызовы:

```
cout << rA.Operator(rC) << endl;
cout << rB.Operator(rC) << endl;
```

Если нас это устраивает, никаких изменений в классы `Derived_a` и `Derived_b` вносить нет необходимости. Если же требуется специальный вид операции `A.Operator(C)`, то нужно добавить соответствующий виртуальный метод в класс `Derived_a`.

Как уже было сказано, подобная методика работает и для классов, не являющихся родственниками. Построим следующую иерархию классов (листинг 10.11).

Листинг 10.11. Мультиметоды для классов, не являющихся родственниками

```
class B;                                // объявление корня второй иерархии
class A                                // корень первой иерархии
{ public:
    virtual ~A(){}
    virtual void Operator(B*)=0;
};
class D1A;                             // объявления классов первой иерархии
class D2A;
class B                                // корень второй иерархии
{ public:
    virtual void Operator(D1A& rA) = 0;
    virtual void Operator(D2A& rA) = 0;
};
// классы иерархии A
class D1A: public A
{ public:
    virtual void Operator(B* R) { R->Operator(*this); }
};
class D2A: public A
{ public:
    virtual void Operator(B* R) { R->Operator(*this); }
};
// классы иерархии B
class D1B: public B
{ public:
    virtual void Operator(D1A& rA) { cout << "D1A!" << endl; };
    virtual void Operator(D2A& rA) { cout << "D2A!" << endl; };
};
class D2B: public B
{ public:
    virtual void Operator(D1A& rA) { cout << "D1A!" << endl; };
    virtual void Operator(D2A& rA) { cout << "D2A!" << endl; };
};
```

Две иерархии — A и B — не имеют общих родственников. Тем не менее методика последовательных виртуальных вызовов, описанная ранее для одной иерархии, работает точно так же. В данном случае все требуемые действия реализованы во второй иерархии, а первая только осуществляет первый шаг двойного переключения по типу. Небольшой фрагмент программы иллюстрирует необходимые переключения:

```
D1A a1; D2A a2;
B *pb = new D1B();           // указатель на базовый класс <- адрес наследника
a1.Operator(pb);             // работают 2 виртуальных вызова
a2.Operator(pb);             // работают 2 виртуальных вызова
```

```
pb->Operator(a1);          // работает 1 виртуальный вызов
pb = new D2B();
a1.Operator(pb);           // работают 2 виртуальных вызова
a2.Operator(pb);           // работают 2 виртуальных вызова
pb->Operator(a1);          // работает 1 виртуальный вызов
```

Двойное переключение по типу — это основа паттерна Visitor (посетитель). Это — один из наиболее трудных для понимания паттернов, недаром в [17] он описан последним.

Резюме

Одной из мощных концепций C++ является множественное наследование. При аккуратном и правильном использовании — это чрезвычайно полезный механизм. Множественное наследование часто используется в больших библиотеках классов.

В том виде, в котором оно реализовано в C++, множественное наследование часто является источником проблем. В первую очередь, это проблема дублирования и неоднозначности, для решения которой в C++ реализован механизм виртуального наследования. С другой стороны, для решения той же проблемы применяется принцип доминирования. Виртуальное наследование «добавляет неэффективности» — размеры класса-наследника увеличиваются.

Еще одним «необязательным» механизмом C++ стал механизм RTTI, который в некотором роде является альтернативой виртуальным функциям, причем значительно более понятной новичкам, чем механизм виртуальных функций. RTTI позволяет узнать истинный тип объекта во время исполнения программы и выполнить динамическое преобразование типа среди родственных полиморфных классов «во все стороны». Это свойство можно задействовать для реализации мультиметодов, однако использование для тех же целей виртуальных функций предпочтительней.

Контрольные вопросы

1. Сколько классов может быть использовано в качестве базовых?
2. Выполняется ли при множественном наследовании и принцип подстановки?
3. Можно ли наследовать открыто от одного класса, а закрыто от другого?
4. В чем проблемы множественного наследования?
5. Объясните смысл виртуального наследования.
6. Как формулируется принцип доминирования?
7. Какие функции отвечают за инициализацию виртуального базового класса?
8. Объясните принцип реализации «финального» класса.
9. Какие конструкции входят в состав механизма RTTI?

10. Можно ли выполнить понижающее приведение типа с помощью оператора `dynamic_cast<>`? А повышающее или перекрестное?
11. Чем различается работа оператора `dynamic_cast<>` при преобразовании указателей и ссылок?
12. Какое исключение генерирует оператор `dynamic_cast<>` и в каких случаях?
13. В каких случаях генерируется исключение `bad_typeid`?
14. Дайте определение мультиметодов.
15. Объясните, каким образом механизм RTTI может использоваться при реализации мультиметодов.

Упражнения

1. Реализовать класс `Money` (см. упражнение 4 в главе 4), используя множественное наследование. Один базовый класс — класс с подсчетом объектов (см. листинг 4.23), второй — класс `Array` из упражнения 6 в главе 8.
2. Реализовать шаблон стека как наследник от двух классов-шаблонов. Один шаблон определяет абстрактный интерфейс стека, второй шаблон — это шаблон `TDeque` (см. главу 6). Реализация методов стека делегируется классу `TDeque`.
3. Определить абстрактный базовый класс `TObject` с подсчетом объектов. Реализовать контейнер-список с элементами типа `TObject`. Использовать множественное наследование для обеспечения возможности хранения в этом контейнере объектов типа `Function` (см. упражнение 5 в главе 9).
4. Реализовать мультиметод сложения с использованием оператора `dynamic_cast<>` для трех видов чисел (см. упражнение 5 в главе 9): действительных (`Real`), комплексных (`Complex`) и дробных (`Fraction`).
5. Реализовать мультиметод умножения для тех же видов чисел, используя оператор `typeid()`.
6. Использовать виртуальные функции при реализации мультиметодов сложения и умножения для трех видов чисел.

Глава 11

Шаблоны классов

Такие контейнеры, как стек, должны работать с элементами любых типов. Однако написать такой универсальный стек не так-то просто. Реализация с помощью нетипизированных указателей `void*` (см. главу 6), хотя и работоспособна, требует предельной аккуратности, иначе можно нарваться на очень неприятные ошибки. Кроме того, динамическое выделение памяти и косвенное обращение сказывается на эффективности, да и преобразования типов, которые нужно выполнять, не украшают текст программы. Наследование немного скрашивает эту безрадостную картину — можно «упрятать внутрь» и косвенное обращение, и преобразование типов (см. листинг 8.11). Однако наследование «добавляет неэффективности» — из-за наследования в программе должны присутствовать два класса вместо одного: базовый с указателями типа `void*` и наследник от него.

Все было бы гораздо проще, если бы тип можно было сделать параметром. Тогда при объявлении объектов-контейнеров можно было бы указывать, элементы какого типа будут размещаться в контейнере. Именно с этой целью в C++ первоначально были добавлены *шаблоны* (см. п. 14 в [1]). В C++ реализованы два вида шаблонов: шаблоны классов и шаблоны функций¹ (см. п. п. 14.5.5 в [1]). Шаблоны классов, очевидно, в первую очередь предназначены для реализации обобщенных контейнеров и позволяют задавать типы элементов в качестве параметров контейнера. Конкретный тип элементов указывается при объявлении объекта. Часто шаблон класса называют *параметризованным типом*. Мы будем употреблять термины «параметризованный тип», «шаблонный класс», «шаблон класса» как эквивалентные. В этом же смысле будет использоваться слово «шаблон».

Концепция шаблонов оказала колоссальное влияние на стиль программирования на C++. Достаточно сказать, что в стандарт была включена стандартная библиотека шаблонов (STL), содержащая контейнеры, итераторы и алгоритмы (см. п. 23 в [1]). Появился даже новый стиль программирования — обобщенное программирование.

¹ Шаблоны функций рассматриваются в следующей главе.

Первое знакомство

Для знакомства с шаблонами классов перепишем наш простой стек (см. листинг 6.9), который верой и правдой послужил нам в главе 8 (см. листинги 8.9–8.12), превратив его в шаблон (листинг 11.1).

Листинг 11.1. Реализация стека в виде шаблона

```
#include <exception>           // необходимо для исключений
#include <string>
#include <iostream>
using namespace std;
//-----класс-шаблон-----
template <class T>             // шаблон; T - параметр шаблона
class TStack
{ struct Elem
    { T data;                  // тип элементов неизвестен
      Elem *next;
      Elem (const T& d, Elem *p) // тип элементов - параметр шаблона
      :data(d), next(p)
      { }
    };
    Elem * Head;              // указатель на вершину
    int count;                 // счетчик элементов
    TStack(const TStack &);    // закрыли копирование
    TStack& operator=(const TStack &); // закрыли присваивание
public:
    class Error: public std::exception // исключение - пустой стек
    { };
    TStack(): Head(0), count(0) // конструктор без аргументов
    { }
    ~TStack()                  // деструктор!
    { while(!empty()) pop(); }
    void push(const T& d)       // тип элементов - параметр шаблона
    { Head = new Elem(d, Head); // новый элемент в стек
      ++count;                  // увеличиваем счетчик
    }
    T top() const               // тип возвращаемого элемента - параметр

    { if (!empty()) return Head->data;
      else throw Error();
    }
    void pop()                  // удаление элемента с вершины
    { if (empty()) throw Error();
      T top = Head->data;        // T - параметр шаблона
      Elem *oldHead = Head; Head = Head->next; delete oldHead;
      --count;                  // уменьшаем счетчик
    }
    bool empty() const          // есть ли элементы в стеке
    { return Head==0; }
    int count() const           // количество элементов в стеке
    { return count; }
};
```

Листинг 11.1 (продолжение)

```
//-----программа-клиент, использующая шаблон стека
int main()
{ //-----стек с числами
  TStack<double> t;           // стек с числами типа double
  t.push(11);                // помещаем в стек числа
  t.push(21);
  t.push(31);
  cout << t.count() << endl;   // в стеке должно быть 3 элемента
  while (!t.empty())         // пока стек не пустой
  { cout << t.top() << endl;   // выводим число с вершины
    double p = t.pop();       // удаляем элемент из стека
  }
  cout << t.count() << endl;   // в стеке должно быть 0 элементов
//-----стек со строками
TStack<string> S;            // стек со строками
S.push("one");              // помещаем в стек строки
S.push("two");
S.push("three");
cout << S.count() << endl;
while (!S.empty())          // пока стек не пустой
{ string p = S.pop();        // удаляем элемент из стека
  cout << p << endl;         // выводим строку
}
cout << S.count() << endl;    // в стеке не должно быть элементов
try { string p = S.pop(); }  // проверка исключения
catch(const exception &e) { cout << e.what() << endl; }
return 0;
}
```

Как можно видеть, шаблон класса начинается ключевым словом `template`. В угловых скобках после него записан параметр шаблона в виде `<class T>`. Имя `T` является параметром-типом. В качестве имени параметра можно использовать, конечно, любой допустимый идентификатор, однако по негласному соглашению в качестве имен параметров-типов указываются имена, начинающиеся с префикса `T`. Внутри класса такой параметр может появляться на тех местах, где разрешается писать конкретный тип. В данном случае тип `T` использовался, чтобы задать тип поля `data` в структуре `Elem`, как тип параметра в методе `push()` и как тип возвращаемого значения в методах `top()` и `pop()`. Сам шаблонный класс имеет тип `TStack<T>`.

Реализация стека в виде шаблона потребовала некоторых изменений. Во-первых, в стеке с указателями (см. листинг 6.9) методы `top()` и `pop()` возвращали нуль (недопустимое значение для указателей), если стек был пуст. Такое решение подходит для указателей, но совершенно не годится для шаблонов — мы не можем знать, какие значения типа `T` являются допустимыми, а какие — нет. Поэтому в шаблонном классе вместо возврата значения генерируется исключение `Error`.

Во-вторых, написан деструктор. И при реализации деструктора, и при реализации методов `top()` и `pop()` неявно предполагалось, что класс `T` имеет конструктор

копирования, так как этот конструктор будет вызываться в деструкторе стека, при возврате значения из методов `top()` и `pop()`, а также в теле цикла:

```
while(!empty())
{ T t = pop(); }
```

Кроме того, класс `T` должен иметь конструктор по умолчанию, который будет вызываться в методе `push()` при создании нового элемента стека. Неявные предположения такого рода при создании шаблонных классов программист обязательно должен иметь в виду.

Использование шаблонного стека демонстрирует программа-клиент. Сначала создается стек с числами типа `double`:

```
TStack<double> t;
```

Обнаружив такую запись, компилятор создает из шаблонного класса конкретный класс, подставив на место `T` тип `double` (листинг 11.2). Процесс создания конкретного класса из шаблона путем подстановки аргументов называется *инстанцированием шаблона* (см. п. 14.7 в [1]).

ПРИМЕЧАНИЕ

- Термин «инстанцирование» (instantiation) не слишком понятен, хотя стал уже практически общепринятым. К сожалению, именно этот термин использовался при переводе самой важной книги о шаблонах [28] и при переводе «библии по C++» [2]. В русском языке наиболее близко передает смысл этого понятия термин «конкретизация».

Листинг 11.2. Инстанцированный шаблон

```
class TStack // имя корректируется компилятором
{ struct Elem
{ double data; // подставлен тип double
  Elem *next;
  Elem (const double& d, Elem *p) // подставлен тип double
  :data(d), next(p) { }
};
Elem * Head; // указатель на вершину
int count; // счетчик элементов
TStack(const TStack &); // закрыли копирование
TStack& operator=(const TStack &); // закрыли присваивание
public:
class Error: public std::exception // исключение - пустой стек
{ };
TStack(): Head(0), count(0) // конструктор
{ }
~TStack() // деструктор!
{ while(!empty()) pop(); }
void push(const double& d) // подставлен тип double
{ Head = new Elem(d, Head); // новый элемент в стек
  ++count; // увеличиваем счетчик
}
double top() const // подставлен тип double
{ if (!empty()) return Head->data;
  else throw Error();
}
```

Листинг 11.2 (продолжение)

```

void pop()
{ if (empty()) throw Error();
  double top = Head->data;          // подставлен тип double
  Elem *oldHead = Head;
  Head = Head->next; delete oldHead;
  --count;                          // уменьшаем счетчик
}
bool empty() const                  // есть ли элементы в стеке
{ return Head==0; }
int count() const                   // количество элементов в стеке
{ return count; }
};

```

Именно этот класс после подстановки транслируется и попадает в работающую программу. Именно он и используется при объявлении объекта-стека *t* в программе-клиенте (см. листинг 11.1). Далее выполняется обычная работа с этим стеком, в частности аргументы-числа метода *push()* переводятся по умолчанию в *double*.

Обратите внимание, что имя инстанцированного класса выделено курсивом. Дело в том, что компилятор корректирует имя инстанцированного класса аналогично тому, как он поступает с именами функций при перегрузке. Зачем же это нужно? Когда компилятор встречает другое объявление, то снова инстанцируется конкретный класс-стек, например, для работы со строками используется такое объявление:

```
TStack<string> S;
```

Очевидно, имена у классов *TStack<double>* и *TStack<string>* должны различаться, иначе возникает ошибка повторного определения.

Таким образом, каждое объявление объекта-стека с аргументом-типом приводит к появлению в программе соответствующего инстанцированного класса. Если наследование и композиция служат для многократного использования объектного кода, то шаблоны, как мы видим, позволяют многократно использовать исходные тексты.

В качестве шаблонного типа мы можем задавать и указатели. Например, мы вполне можем написать в программе следующее объявление:

```
TStack<void *> st;
```

По этому объявлению создается класс, практически совпадающий с первым универсальным стеком (см. листинг 6.9) — добавлен только деструктор. И работать с таким стеком нужно точно так же, как и с определенным вручную:

```

st.push(new double(21));          // помещаем в стек числа
st.push(new double(22));
st.push(new double(23));
while (!st.empty())               // пока стек не пустой
{ cout << *(double *)st.top() << endl; // выводим число с вершины
  double *p = (double *)st.pop();    // удаляем элемент из стека
  delete p;                          // возвращаем память
}

```

Однако помните о проблемах, возникающих при использовании указателей в качестве элементов стека, — эти проблемы никуда не исчезают и в случае шаблонного класса.

Определение шаблона класса

Шаблон класса объявляется следующим образом (см. п. п. 14.5.1 в [1]):

```
template <параметры>          // объявление шаблона
class имя_класса
{ // определение класса
};
```

Имя класса не должно совпадать с другими именами в той же области видимости. Определение объектов в общем виде выглядит так:

```
имя_класса<аргументы> имя_объекта;          // одиночный объект
имя_класса<аргументы> имя_объекта[количество]; // массив
имя_класса<аргументы> * имя_объекта;        // указатель
```

Параметр функции — константную ссылку — нужно объявлять так:

```
const имя_класса<аргументы> &имя_объекта;
```

В шаблоне стека у нас был один параметр, но обычно параметров в угловых скобках бывает несколько и они пишутся через запятую. Вспомним, что ассоциативный контейнер содержит множество пар «ключ-значение». Как раз для такого случая можно определить шаблон структуры (которая тоже является классом) с двумя элементами разного типа, например:

```
template <class T1, class T2 >
struct Pair { T1 first; T2 second; };
```

Имена параметров видимы в пределах шаблона, поэтому все имена в одном шаблоне должны быть разными. Объявлять объекты с помощью указанного шаблона можно так:

```
Pair<int, int> a;
Pair<int, double> b;
```

Такие переменные можно даже инициализировать точно так же, как и обычные структуры, например:

```
Pair<int, double> b = {1, 2.1};
```

В качестве аргументов при объявлении можно использовать и реализованные классы:

```
Pair<string, Date> d;
```

Естественно, имена классов должны быть видимы в точке определения переменной.

Ключевое слово `class` требуется писать перед каждым параметром. Вспомним, что в заголовке функции каждый параметр тоже нужно писать с типом — список

параметров шаблона должен удовлетворять такому же требованию. Например, следующее объявление ошибочно:

```
template <class T1, T2>                                // так писать нельзя!
struct Pair { T1 first; T2 second; };
```

При объявлении параметра шаблона вместо слова `class` стандарт разрешает использовать слово `typename` (см. п. п. 14.1/2 в [1]), например:

```
template <typename T1, typename T2>
struct Pair { T1 first; T2 second; };
```

Это определение ничем не отличается от приведенного ранее. Можно для одного параметра писать слово `class`, для другого — `typename`, например:

```
template <typename T1, class T2>
struct Pair { T1 first; T2 second; };
```

Наличие двух разных ключевых слов, используемых в одном и том же смысле, выглядит странным. Однако так сложилось исторически: слово `typename` появилось позже в связи с некоторыми тонкостями объявлений типов внутри шаблонного класса, которые мы рассмотрим позже. Так как по смыслу это слово как раз означает «имя типа», его использование в списке параметров шаблона выглядит естественным. Слово `class` оставлено для обратной совместимости.

Так же как и обычный класс, шаблон можно объявить (см. п. 14.5 в [1]). Объявление состоит из заголовка шаблона и не включает тело класса, например:

```
template <typename T> class TStack;
template <typename T1, typename T2> struct Pair;
template <typename T> class TSimpleArray;
```

Объявления шаблона служат той же цели, что и объявления обычных классов: ввести имя шаблона в область видимости, чтобы на него можно было ссылаться.

Внешнее определение методов

Мы написали шаблон стека, определив все методы в классе как подставляемые. Однако чаще методы определяются вне шаблонного класса. Перепишем опять наш шаблон стека (листинг 11.3).

Листинг 11.3. Внешнее определение методов шаблона

```
//-----определение шаблона класса-----
template <class T>
class TStack
{ struct Elem
  { T data;
    Elem *next;
    Elem (const T& d, Elem *p)
      :data(d), next(p) { }
  };
```

```

    Elem * Head;
    int count;
TStack(const TStack &);           // закрыли копирование
TStack& operator=(const TStack &); // закрыли присваивание
public:
    class Error: public std::exception { };
    TStack();
    ~TStack();
    void push(const T& d);
    T top() const;
    T pop();
    bool empty() const;
    int count() const;
};
//-----определение методов-----
template <class T>
TStack<T>::TStack(): Head(0), count(0)
{
}
template <class T>           // шаблон
TStack<T>::~~TStack()        // прототип
{ while(!empty())           // тело
    { T t = pop(); }
}
template <class T>
void TStack<T>::push(const T& d)
{ Head = new Elem(d, Head);
  ++count;
}
template <class T>
T TStack<T>::top() const
{ if(!empty()) return Head->data;
  else throw Error();
}
template <class T>
T TStack<T>::pop()
{ if (empty()) throw Error();
  T top = Head->data;
  Elem *oldHead = Head;   Head = Head->next;
  delete oldHead;
  --count;
  return top;
}
template <class T>
bool TStack<T>::empty() const
{ return Head==0; }
template <class T>
int TStack<T>::count() const
{ return count; }

```

В этом случае определение метода тоже должно начинаться со слова `template` с параметрами шаблона. Тип шаблонного класса `TStack<T>` задается в качестве префикса. В теле метода, как обычно, все имена класса можно употреблять без префикса.

Параметры шаблона — не типы

Стандарт разрешает в качестве параметров шаблона задавать параметры, не являющиеся типами (см. п. п. 14.1/4 в [1]). Такими параметрами могут быть:

- объект интегрального (integral) или перечислимого типа;
- указатель на объект или указатель на функцию;
- ссылка на объект или ссылка на функцию;
- указатель на член класса.

К интегральным типам относятся все целочисленные и символьные типы, а также булев тип. Как видите, нельзя использовать в качестве параметра ни один из встроенных дробных типов: ни `float`, ни `double`, ни `long double`.

ПРИМЕЧАНИЕ

В [28] высказано предположение двух членов Комитета по стандартизации (Н. Джосаттиса и Д. Вандевурда), что в будущих версиях стандарта это недоразумение будет исправлено.

Наиболее часто применяются целочисленные параметры, с помощью которых обычно задают некоторое количественное значение, например количество элементов контейнера-массива.

В библиотеке Boost (и в [29]) приводится пример шаблонного класса `array`, который предлагается использовать вместо встроенных массивов. Напишем упрощенную версию `TSimpleArray` этого шаблона (листинг 11.4). Пока не будем обращать внимание на эффективность расходования памяти и зададим массив как поле класса. Это позволит нам не определять ни конструктор копирования, ни оператор присваивания. Однако конструктор инициализации определим, чтобы не заполнять массив путем присваивания. Из операций достаточно определить операцию индексирования `operator[]` для доступа к элементам массива (как обычно, в двух вариантах).

Очевидно, тип элементов можно сделать параметром шаблона. Вторым претендентом в параметры является размер массива. Таким образом, заголовок шаблона может выглядеть так:

```
template <typename T, std::size_t N>
```

Точно такие же заголовки нужно будет писать и во внешних определениях методов. Тип `size_t` определен в библиотеке `<cstdint>` как `unsigned int`, поэтому нам нет нужды использовать оператор `typedef` для сокращения записи.

Листинг 11.4. Применение шаблона для реализации массива

```
#include <stdint>
template<typename T, std::size_t N>           // параметры шаблона
class TSimpleArray {
public:
    // типы
    typedef T                               value_type;
```

```

typedef T&                reference;
typedef const T&          const_reference;
typedef std::size_t       size_type;
static const size_type static_size = N;           // размер массива
TSimpleArray(const T &t);                          // конструктор
size_type size() const                          // получение размера
{ return static_size; }
reference operator[](const size_type& i)          // доступ к элементам
{ rangecheck(i); return elem[i]; }
const_reference operator[](const size_type& i) const
{ rangecheck(i); return elem[i]; };
private:
void rangecheck (const size_type& i) const        // проверка индекса
{ if (i >= size())
  { throw std::range_error("TSimpleArray - range!"); }
}
T elem[N];                                       // поле-массив
};
template<typename T, std::size_t N>              // реализация конструктора
TSimpleArray<T,N>::TSimpleArray(const T &t)
{ for (int i = 0; i<N; i++) elem[i] = t; }

```

В самом начале шаблонного класса определены несколько типов-синонимов с помощью оператора `typedef`, которые кажутся совершенно лишними. Однако такая практика улучшает читаемость класса и особенно полезна, когда пишется не единственный шаблон, а целая библиотека шаблонов — тогда во всех шаблонах можно определить одинаковые внутренние имена. Именно так сделано во всех классах-шаблонах в стандартной библиотеке шаблонов (STL).

Проверка индекса в операциях индексирования `operator[]` выполняется приватной функцией `rangecheck()`. Функция просто генерирует стандартное исключение при «вылете за границу» массива.

Использовать такой массив можно следующим образом:

```

TSimpleArray<int, 10> t(0);           // инициализируем нулем
TSimpleArray<int, 10> r(t);           // конструктор копирования
TSimpleArray<int, 10> p(2);           // все элементы = 2
for (int i = 0; i < p.size(); i++)
    cout << p[i] << ' ';           // доступ "справа"
cout << endl;
p = r;                                // присваиваем - типы одинаковы
for (int i = 0; i < p.size(); i++)
    p[i] = i;                         // доступ слева

```

Конструктор по умолчанию отсутствует¹, поэтому невозможно объявлять массив без инициализации:

```

TSimpleArray<int, 10> t;               // нельзя - нет конструктора по умолчанию

```

Копирование и присваивание работают только для массивов с одинаковым типом элементов и размером. Изменяя тип элементов и/или размер, получаем массивы

¹ Напомню, что при наличии хотя бы одного явно определенного конструктора конструктор по умолчанию (без аргументов) не создается.

разных типов. При попытках использовать оператор присваивания или конструктор копирования с такими разнотипными массивами компилятор выдает сообщение о невозможности преобразования типов. Объявим несколько массивов, отличающихся от массивов `p` и `t`, например:

```
TSimpleArray<int, 15> u(t);           // не совпадает размер
TSimpleArray<double, 10> w(t);       // не совпадает тип
TSimpleArray<int, 5> u1(3);
p = u1;                             // размеры не совпадают
TSimpleArray<double, 10> w1(4);
w1 = t;                             // типы элементов не совпадают
```

Во всех этих случаях компилятор выдает сообщение об ошибке. Обратите внимание, что ошибка возникает даже в том случае, если принимающий массив «больше» как по размеру, так и по типу — преобразование типа элементов для шаблонов автоматически не выполняется.

Инициализация нулем

Как и наш динамический массив `TArray` (см. листинг 6.10), класс `TSimpleArray` значительно «умнее» встроенных массивов. Массивы можно присваивать без написания циклов, перехватывается типичнейшая (и наиболее распространенная) ошибка «вылета за границу» массива, что здорово облегчает жизнь программисту. Однако в классе отсутствует конструктор без аргументов, что создает некоторое неудобство.

Как мы знаем, в обычном классе можно не определять конструктор без аргументов, если присвоить значения по умолчанию параметрам в конструкторе инициализации. Для каждого конкретного типа это легко можно сделать. Однако для шаблонного класса тут возникает проблема, так как конкретный тип присваиваемого выражения неизвестен. Тем не менее в стандарте такая ситуация учтена, и мы можем реализовать конструктор инициализации таким образом:

```
TSimpleArray(const T &t = T());           // прототип
template<typename T, std::size_t N>       // реализация
TSimpleArray<T,N>::TSimpleArray(const T &t)
{ for (int i = 0; i<N; i++) elem[i] = t; }
```

Следующая конструкция может быть задана, естественно, не в прототипе, а в заголовке реализации конструктора:

```
const T &t = T()
```

Для любого класса `T` она означает явный вызов конструктора без аргументов — подставляемый вместо `T` класс должен такой конструктор иметь. Однако для встроенных типов конструкторов не существует. Тем не менее для встроенных типов стандарт определяет смысл присваивания как инициализацию нулем:

```
T t = T()
```

Например:

```
int t = int();
```

Этот оператор означает

```
int t = 0;
```

Еще пример:

```
double t = double();
```

А этот оператор означает

```
double t = 0.0;
```

И подобные присваивания для встроенных типов можно писать явно.

Представленную конструкцию можно использовать не только в списке параметров, но и в любом месте шаблона, где допускается объявить переменную и инициализировать ее. Например, в любом методе шаблонного класса разрешается писать

```
T x = T();
```

Это присваивание превращается в корректную конструкцию при подстановке любого типа, как встроенного, так и реализованного.

Как мы помним, инициализация нулем разрешена и в списке инициализации конструктора. Аналогичную конструкцию разрешается использовать также в шаблонном классе, например:

```
template <class T>
class Type
{ public:
    Type(): x()                // инициализация нулем
    { }
    // ...
private:
    T x;                       // инициализируемое поле
};
```

В данном случае стандарт гарантирует, что, если вместо *T* подставить встроенный тип, поле *x* будет проинициализировано нулем. Для классов, естественно, вызывается конструктор без аргументов, хотя мы его вызов и не прописываем.

Параметры шаблона по умолчанию

Параметрам шаблонного класса можно присваивать значения по умолчанию (см. п. п. 14.1/9 в [1]). Разрешается присваивать значения по умолчанию и для параметров-типов. Например, для шаблона `TSimpleArray` можно присвоить по умолчанию тип `double` и размер массива равный 100. Заголовок шаблона пишется привычным способом:

```
template<typename T = double, std::size_t N = 100>
```

В реализации методов ничего изменять не требуется. Тогда допускаются объявления массивов такого вида:

```
TSimpleArray<int, 20> t;           // полное объявление
TSimpleArray<Date> d;             // количество по умолчанию
TSimpleArray<> p;                  // тип и количество по умолчанию
```

Пустые скобки писать обязательно, иначе компилятор будет искать обычный класс `TSimpleArray` (и, естественно, не найдет). Во всех объявлениях инициализация выполняется по умолчанию. Класс `Date` должен иметь конструктор без аргументов, или конструктор инициализации с параметрами по умолчанию.

Как и для функций с параметрами по умолчанию, при определении объекта нельзя пропускать левые параметры, например:

```
TSimpleArray<10> t;           // ошибка трансляции
```

Такое объявление приведет к ошибке трансляции — компилятор не обнаружит определения шаблона с одним целочисленным параметром.

Для целочисленных параметров допускается задавать в качестве значения константное выражение, которое компилятор способен вычислить на этапе трансляции. Для параметров-типов, естественно, можно задать либо встроенный тип, либо любое имя типа, видимое в точке определения шаблона.

Как и для обычных функций, присваивать значения по умолчанию нужно правым параметрам. Например, можно написать заголовок шаблона `TSimpleArray`:

```
template<typename T, std::size_t N = 100>
```

При этом не разрешается писать так:

```
template<typename T = double, std::size_t N>
```

Специализация

Наряду с общим шаблоном часто бывает необходима некоторая специализированная версия того же шаблона. Поэтому изначально в шаблонах был реализован механизм *специализации* (см. п. 14.7 в [1]). Специализация заключается в том, что на основе исходного *первичного* шаблона реализуется его специализированная версия для некоторых конкретных значений параметров. Специализация шаблона называется *полной*, если конкретизированы все параметры первичного шаблона. Если определена только часть параметров, то специализация называется *частичной* (см. п. п. 14.5.4 в [1]).

Специализация шаблона — это не присвоение параметров по умолчанию. Шаблон с параметрами по умолчанию является первичным шаблоном, который тоже можно специализировать. Специализация шаблона — это «перегрузка» для классов.

Специализация шаблона — это не инстанцирование. Инстанцирование выполняет компилятор при трансляции программы, когда встречает объявление объекта с конкретными значениями параметров шаблона. Специализацию первичного шаблона реализует программист как отдельный класс-шаблон, в котором некоторые параметры первичного шаблона имеют известные значения.

Специализация применяется чаще всего для параметров-типов. Первичный шаблон определяет общий вариант, а специализированная версия — частный случай для конкретных типов. Полная специализация представляет собой конкретный класс, реализованный для конкретных значений параметров первичного шаблона.

Обычно в специализированных версиях переопределяются отдельные (или даже все) методы, которые для заданного конкретного типа должны работать не так, как в общем случае. Например, пусть в первичном шаблоне определена операция присваивания `operator=`. Для символьных массивов эта операция обычно работает совершенно не так, как для всех остальных типов. Тогда нам нужно определить специализированную версию первичного шаблона для `const char[]` и переопределить в ней операцию присваивания.

При определении шаблона и его специализированных версий должен соблюдаться порядок следования: сначала должен быть определен первичный шаблон, и только после него разрешается определять специализированные версии (листинг 11.5).

Листинг 11.5. Специализации шаблона

```
template<class T, class U>           // первичный шаблон
class S
{ public: void print(); };           // переопределяемый метод
// частичные специализации шаблона
template<class T >                  // U = int
class S<T, int>
{ public: void print(); };
template<class U>                   // T = int
class S<int, T>
{ public: void print(); };
template<class T>                   // T = U
class S<T, T>
{ public: void print(); };
template<class T, class U>           // частичная специализация для указателей
class S<T*, U*>
{ public: void print(); };
template<class T>                   // T = T*, U = void*
class S<T*, void*>
{ public: void print(); };
template<>                           // полная специализация
class S<int*, double>
{ public: void print(); };
```

Специализации определяются наличием аргументов в угловых скобках после имени класса. Полную специализацию компилятор определяет по пустым угловым скобкам `<>` после слова `template`.

Наличие специализаций класса `S` ставит вопрос о том, какая конкретная версия будет использоваться при вызове метода `print()`. Ответ почти очевиден: из нескольких подходящих специализаций выбирается «наиболее специализированная». Например:

```
S<float, double>().print();           // первичный шаблон
S<char, char>().print();               // T = U
S<int, char*>().print();               // T = int
```

Здесь в первом вызове используется первичный шаблон, второй вызов соответствует шаблону со специализацией `T = U`, третий — сопоставляется со специализацией `T = int`.

Однако со специализациями шаблонов нужно быть осторожным. Например:

```
S<int, int>().print();
```

Этот вызов является неоднозначным — ему соответствует аж три возможных специализации: $T = \text{int}$, $U = \text{int}$ и $T = U$. Аналогично, неоднозначными будут, например, вызовы

```
S<int, void*>().print();
S<int*, void*>().print();
S<int*, int*>().print();
```

ВНИМАНИЕ

Круглые скобки перед точкой означают вызов конструктора по умолчанию. В данном случае каждый вызов выполняется для анонимного временного объекта.

Наиболее типичным видом специализации является специализация шаблона для указателей. *Образец* (см. с. 393 в [2]) специализации $\langle T^*, U^* \rangle$ после имени означает, что эта специализация должна использоваться во всех случаях, когда аргументом шаблона является указатель любого типа, кроме void^* , для которого реализована более «специализированная» версия $\langle T^*, \text{void}^* \rangle$, например:

```
S<int*, char*>().print();
S<int*, void*>().print();
```

В первом случае используется специализация с аргументами $\langle T^*, U^* \rangle$, а во втором — более специализированная версия $\langle T^*, \text{void}^* \rangle$.

Чтобы специализировать шаблон, необязательно иметь полное определение первичного шаблона — достаточно объявления, например:

```
template <class T, class U> class S;           // объявление первичного шаблона
template <typename T> class S<T*, void*>      // частичная специализация
{ // ...
};
template<> class S<int, void*>                 // полная специализация
{ // ...
};
```

Все будет корректно работать, если мы не будем пытаться инстанцировать первичный шаблон.

Специализировать шаблон можно и по параметрам, не являющимся типами, например:

```
template<int n, int m> struct A {};           // первичный шаблон
template<int n> struct A<n,n> {};            // специализация n = m
```

Нужно подчеркнуть, что специализация — это не наследование. Первичный шаблон и его специализированная версия — это *два разных шаблонных класса*. Полная специализация шаблона вообще представляет собой реализованный на основе первичного шаблона независимый класс. Поэтому полагаться на то,

что в специализированной версии «окажутся» методы, реализованные в первичном шаблоне, нельзя. Пусть в первичном шаблоне определен некоторый метод:

```
void f(void);
```

Если в специализированной версии этот метод не определен, то попытки вызвать метод первичного шаблона для класса специализированной версии приведут к ошибкам трансляции.

Шаблоны и... шаблоны

Включение аппарата шаблонов в C++ потребовало решения многих проблем, касающихся взаимодействия обычных и шаблонных конструкций. Даже в стандарте C++, который является очень лаконичным документом, раздел о шаблонах (см. п. 14 в [1]) занимает 60 страниц (235–297). Одна из таких общих проблем — вложенные конструкции, например:

- поле-шаблон в классе или в шаблоне;
- метод-шаблон, в том числе конструктор, в классе или шаблоне;
- вложенный шаблонный класс.

Ограничений на вложенные шаблонные классы нет: и класс может быть объявлен внутри шаблона, и шаблон — внутри как класса, так и шаблона. Единственное ограничение — шаблонный класс нельзя объявлять внутри функции. Множество прекрасных примеров вложенных классов можно обнаружить в различных файлах стандартной библиотеки шаблонов (STL) [29–35]. Например, любой шаблонный класс, реализующий один из контейнеров, содержит вложенные классы итераторов.

Кроме того, есть еще множество нюансов, связанных с параметрами шаблона, не являющимися типами, например параметрами-шаблонами, параметрами-указателями и параметрами-ссылками. Рассмотрим наиболее часто используемые из этих конструкций.

Поле-шаблон

В классе вполне можно объявить поле, инстанцировав некоторый шаблон. Например, мы можем реализовать простую очередь ограниченного размера на основе шаблонного класса `TSimpleArray` (см. листинг 11.4). Это может выглядеть так, как показано в листинге 11.6.

Листинг 11.6. Очередь с полем-шаблоном

```
class TQueue
{
    TSimpleArray<double, 1000> t;           // поле-шаблон
public:
    // методы
};
```


Как видите, объявление поля-шаблона ничем не отличается от объявления обычного объекта. Такая очередь практически ничем не отличается от очереди, реализованной на основе обычного массива. К сожалению, при использовании этого класса не видно, сколько элементов и какого типа можно помещать в очередь:

```
TQueue q;
```

Поэтому более интересный вариант — объявить поле-шаблон в шаблоне. Перепишем нашу простую очередь в виде шаблона (листинг 11.7).

Листинг 11.7. Поле-шаблон в шаблоне

```
template <typename T, std::size_t N = 1000>           // параметры очереди
class TQueue
{ TSimpleArray<T, N> t;                               // зависимое имя
public:
    // методы
};
```

Тогда при вызове можно видеть тип элементов и количество элементов:

```
TQueue<double> qt;
TQueue<double, 500> queue;
```

Параметр-шаблон

Еще один вариант реализации — использовать в качестве параметра шаблон (см. п. п. 14.3.3 в [1]). Это дает нам возможность иметь разные реализации очередей в зависимости от контейнера, который передается классу-шаблону `TQueue` в качестве аргумента. Параметру-шаблону, как и любому другому виду параметров шаблонного класса, можно присвоить значение по умолчанию — пусть это будет наш класс `TSimpleArray`. Посмотрите, как это делается (листинг 11.8).

Листинг 11.8. Шаблонный параметр шаблона

```
template <typename T,                                // тип элементов
        std::size_t N = 1000,                        // количество
        template <class, std::size_t>                // параметр-шаблон
        class Container = TSimpleArray>              // значение по умолчанию
class TQueue                                          // конец списка
{ Container<T, N> t;                                  // поле-контейнер
public:
    // методы
};
```

Параметр-шаблон объявлен последним:

```
template <class, std::size_t> class Container = TSimpleArray
```

Так же как обычный шаблон, параметр начинается с ключевого слова `template`, за которым следует список его параметров. Однако в списке параметров параметра-шаблона не задаются имена параметров¹ — они просто не используются.

¹ Не запутались? Можно сказать по-другому: имена параметров параметра-шаблона.

Можно провести аналогию с неиспользуемым параметром при реализации постфиксной операции инкремента `operation++(int)`.

Объявление такого контейнера в программе может выглядеть так:

```
TQueue<double> q1; // размер и контейнер по умолчанию
TQueue<double, 5000> q2; // контейнер по умолчанию
TQueue<double, 500, TSimpleArray> q3; // все задано явно
```

Обратите внимание, что в третьем варианте задается только имя контейнера без аргументов. Вместо нашего массива подойдет любой контейнер, имеющий два аналогичных параметра: тип элементов и беззнаковое целое. Вспомним «умный» массив `TArray` (см. листинг 6.10) и преобразуем его в шаблон. Интерфейс нового шаблонного класса практически не отличается от интерфейса `TSimpleArray` (листинг 11.9).

Листинг 11.9. Шаблонный класс `TArray`

```
template <typename T = double, std::size_t n = 100>
class TArray
{ public:
    // определение типов
    typedef std::size_t      size_type;
    typedef std::size_t      index_type;
    typedef T                value_type;
    typedef T&               reference;
    typedef T*               iterator;
    typedef T*               pointer;

    TArray(size_type size, value_type k=0.0);
    TArray(const TArray &a);
    TArray(const TArray &a, index_type begin, size_type k);
    TArray(const iterator begin, const iterator end);
    ~TArray();

    reference operator[](index_type index);
    const reference operator[](index_type index) const;
    TArray& operator=(const TArray &a);
    TArray& assign(const TArray &a, index_type l, index_type r);
    TArray& assign(const iterator begin, const iterator end);
    size_type size()const { return size_array; };
    iterator find(const value_type &a);
    friend ostream& operator <<(ostream& to, const TArray &a);
    friend istream& operator >>(istream& to, TArray &a);
    private:
        size_type size_array;
        pointer data;
};
```

Обратите внимание, как мало нам пришлось изменить для преобразования класса `TArray` в шаблон: фактически добавился только заголовок шаблона и в определении типов слово `double` заменено параметром шаблона `T`. При правильном подходе возможные изменения инкапсулируются в небольшой области текста и минимизируются.

Параметры шаблона сделаны точно такими, как в шаблоне `TSimpleArray`. Поэтому этот класс можно использовать в качестве аргумента при объявлении объекта-очереди:

```
TQueue<double, 500, TArray> Q4; // все задано явно
```

Отметим некоторые особенности объявления параметра-шаблона. Во-первых, количество, порядок и типы параметров параметра-шаблона должны в точности соответствовать параметрам подставляемого аргумента-шаблона (в данном случае — `TSimpleArray`). Нельзя, например, переставить их местами:

```
template <std::size_t, class> class Container = TSimpleArray
```

Точно такие же параметры имеет и второй шаблонный класс — `TArray`.

Во-вторых, в списке параметров параметра-шаблона разрешается вместо слова `class` использовать слово `typename`, например:

```
template <typename, std::size_t> class Container = TSimpleArray
```

Однако перед именем самого параметра-шаблона можно писать только слово `class`:

```
class Contaner
```

Не разрешается писать ни слово `typename`, ни слово `struct`, ни слово `union`.

В-третьих, некоторые проблемы доставляют параметры по умолчанию. Наш контейнер `TSimpleArray` такой параметр имеет — это количество элементов массива `N`. В шаблоне `TQueue` задан собственный параметр `N` и ему присвоено значение по умолчанию 1000. Однако очередь, вообще говоря, — структура данных потенциально бесконечная, поэтому количество элементов внутреннего контейнера должно быть его «внутренним» делом; предполагается, что контейнер самостоятельно управляется со своей памятью. Но задавать значения по умолчанию в шаблоне-параметре нельзя — его имена компилятором не рассматриваются в качестве аргументов! Например:

```
template <typename T,
        template <class, std::size_t N = 100> class Container=TSimpleArray
>
```

Этот заголовок шаблона `TQueue` в Visual C++.NET 2003 при объявлении поля-контейнера `Container<T, N> t`; вызывает ошибку компиляции о неопределенном имени `N`:

```
error C2065: 'N' : undeclared identifier
```

Поэтому нам пришлось вынести этот параметр на верхний уровень.

Метод-шаблон

В любом классе, как в обычном, так и в шаблоне, можно объявить метод-шаблон (см. п. п. 14.5.2 в [1]). Наш шаблонный класс `TSimpleArray` не позволяет присваивать массивы разной длины и (или) массивы с разными типами элементов.

Между тем вполне можно было бы разрешить присвоение «меньшего» массива «большему» — как по размеру, так и по типу. Например, операцию присваивания можно разрешить для следующих массивов:

```
TSimpleArray<int, 10> t;
TSimpleArray<int, 5> r;
t = r; // больший размер = меньший размер
TSimpleArray<double, 10> m;
TSimpleArray<int, 10> n;
m = n; // больший тип = меньший тип
TSimpleArray<double, 10> y;
TSimpleArray<int, 5> x;
y = x; // больший = меньший
```

В первом случае массив меньшего размера *r* копируется в массив большего размера *t*. Массивы *TSimpleArray* — не динамические, количество элементов во время выполнения не изменяется, поэтому «лишние» элементы *t* можно либо заполнять нулями, либо оставлять без изменения. Остановимся на втором варианте. Во втором присваивании размеры одинаковы, а тип принимающего массива «больше». В этом случае просто выполняется преобразование типа. И третий вариант объединяет оба предыдущих.

Таким образом, нам требуется реализовать операцию присваивания, в которой аргументы правого и левого типов представляют собой переменные *TSimpleArray* разных типов и размеров. Это можно сделать, определив *метод-шаблон*. Поскольку слева и справа в операции участвуют разные типы *TSimpleArray*, принимающий массив, очевидно, должен быть нашим основным типом, параметры которого указаны в шаблоне класса, а правый аргумент — массив, параметры которого задаются в методе. Прототип метода тогда выглядит так:

```
template<typename TI, std::size_t NI>
TSimpleArray<T,NI>& operator=(const TSimpleArray<TI, NI>& rhs);
```

Модифицированный шаблон *TSimpleArray* представлен в листинге 11.10.

Листинг 11.10. Метод-шаблон в шаблонном классе

```
#include <cstdlib> // для size_t
template<typename T = double, std::size_t N = 10>
class TSimpleArray {
public:
    // типы
    typedef T value_type;
    typedef T& reference;
    typedef const T& const_reference;
    typedef std::size_t size_type;
    // конструкторы и присваивание
    template<typename TI, std::size_t NI>
    TSimpleArray<T,NI>& operator=(const TSimpleArray<TI, NI>& rhs);
    TSimpleArray(const T &t = T());
    // получение размера
    size_type size() const { return static_size; }
```

продолжение ➤

Листинг 11.10 (продолжение)

```

    // доступ к элементам
    reference operator[](const size_type& i)
    { rangecheck(i); return elem[i]; }
    const_reference operator[](const size_type& i) const
    { rangecheck(i); return elem[i]; }
private:
    static const size_type static_size = N;    // размер массива
    // проверка индекса
    void rangecheck (const size_type& i) const
    { if (i >= size()) { throw std::range_error("TSimpleArray - range!"); } }
    T elem[N];
};
template<typename T, std::size_t N>
TSimpleArray<T,N>::TSimpleArray(const T &t = T())
{ for (int i = 0; i<N; i++) elem[i] = t; }
// Определение метода-шаблона
template<typename T, std::size_t N>                // внешний шаблон (класса)
template<typename TI, std::size_t NI>              // внутренний шаблон (метода)
TSimpleArray<T,N>&                                // возвращаемый тип
TSimpleArray<T,N>::operator=(const TSimpleArray<TI, NI>& rhs)
{ if ((void *)this!=(void *)&rhs)                // проверка самоприсваивания
  { if (N>=NI)                                     // проверка размера
    for(int i = 0; i<NI; ++i)
      elem[i]=static_cast<T>(rhs[i]);             // преобразование типа
  }
  return *this;
}

```

Обратите внимание, что определение метода-шаблона начинается с заголовка `template` шаблонного класса, а затем уже задается его собственный заголовок `template`. Так как нам неизвестен тип в шаблоне-методе, то приходится адреса основного типа и присваиваемого типа приводить к `void*`.

Заметим, что определенный нами шаблон операции присваивания не замещает оператор присваивания, генерируемый по умолчанию (см. сноску 109 на с. 209 п. 12.8 в [1]). Для присвоения массивов одного типа по-прежнему будет вызываться стандартный оператор присваивания. Это можно увидеть, выполнив в отладчике программу из листинга 11.11. Для сокращения текста шаблонный класс не указан.

Листинг 11.11. Проверка метода-шаблона

```

#include <iostream>
// здесь должен быть шаблонный класс
int main()
{ TSimpleArray<int, 10> t;
  for(int i = 1; i<10; ++i) t[i] = i;
  for(int i = 0; i<10; ++i) std::cout << t[i]<< ' ';
  std::cout << std::endl;
  TSimpleArray<int, 10> tt;                // тип tt = типу t
  tt = t;                                  // встроенная
  TSimpleArray<int, 3> p;                  // p.size() < t.size()
}

```

```

for(int i = 0; i<3; ++i) p[i] = i+30;
t = p;                                     // метод-шаблон
for(int i = 0; i<10; ++i) std::cout << t[i]<< ' ';
std::cout << std::endl;
TSimpleArray<double, 12> w;
w = p;                                     // метод-шаблон
for(int i = 0; i<12; ++i) std::cout << w[i]<< ' ';
std::cout << std::endl;
w = tt;                                    // метод-шаблон
for(int i = 0; i<12; ++i) std::cout << w[i]<< ' ';
std::cout << std::endl;
return 0;
}

```

При выполнении следующего присваивания выполняется встроенная операция, генерируемая по умолчанию:

```
tt = t;
```

В остальных случаях выполняется реализованная операция-шаблон. Аналогично шаблон конструктора копирования никогда не замещает генерируемый по умолчанию конструктор копирования (см. сноску 106 на с. 207 п. 12.8 в [1]). Методы-шаблоны не могут быть виртуальными, хотя для обычных методов шаблонного класса такого ограничения нет. Кроме того, метод-шаблон может быть определен и в обычном классе. В этом случае его определение похоже на определение шаблонной функции¹, так как заголовок шаблона `template` только один.

Шаблоны и наследование

Ограничений на наследование нет — разрешаются все разновидности:

- ☐ шаблон наследует от обычного класса;
- ☐ класс наследует от шаблона;
- ☐ шаблон наследует от шаблона.

Необходимо также разобраться в отношениях дружественности при использовании шаблонов:

- ☐ дружественная функция для шаблонного класса;
- ☐ дружественная функция-шаблон для шаблонного или обычного класса;
- ☐ дружественный класс для шаблонного класса.

Шаблон вполне может наследовать от простого класса. Например, пусть мы хотим, чтобы все классы, сгенерированные из некоторого шаблона, имели общие статические поля и (или) методы. Это можно реализовать как раз путем наследования шаблона от простого класса — такой класс называется независимым базовым классом. Мы должны вынести все общие статические члены в нешаблонный базовый класс, от которого и должен наследовать наш шаблон (листинг 11.12).

¹ Шаботонные функции (шаботоны функций) рассматриваются в следующей главе.

Листинг 11.12. Наследование шаблона от класса

```
struct Common
{ static unsigned long Number;
};
template <typename T>
class X: public Common
{ // ...
};
unsigned long Common::Number = 0;
```

Статическое поле `Number` будет общим для всех классов, инстанцированных из шаблона `X`.

Наследование шаблона от нешаблонного класса можно использовать в некоторых случаях для решения проблемы «разбухания» кода при инстанцировании шаблонов. Например, нам потребовалась частичная специализация шаблона `TStack` (см. листинг 11.1) для указателей:

```
template <class T>
class TStack<T*>
{ //...
};
```

При инстанцировании этого шаблона в программе образуется несколько практически одинаковых классов, например:

```
TStack <int *> i;
TStack <double*> d;
TStack <unsigned long *> ul;
```

Однако можно существенно сократить объем кода, используя полную специализацию шаблона и делегирование:

```
template <> class TStack <void*> // полная специализация
{ //...
    typedef std::size_t size_type;
    void push(const void* &t);
    void* top();
    void pop();
    bool empty() const;
    int count() const
};
```

Эта полная специализация является обычным классом. Частично специализированный шаблон может наследовать от полной специализации, например:

```
template <class T>
class TStack<T*>: private TStack<void*>
{ //...
    typedef TStack<void*> TBase;
public:
    void push(const T* &t) { TBase::push(t); }
    void pop() { TBase::pop(t); }
    T* top() { return static_cast<T*>(TBase::top()); }
    //...
};
```

В этом шаблоне реализация методов чрезвычайно экономная: метод шаблонного класса вызывает базовый (делегирует работу). Таким образом, имеем существенную экономию объема кода, так как основная реализация выполнена в единственном экземпляре в классе `TStack<void*>`. Такой подход еще и сокращает время трансляции, так как основной «длинный» класс транслируется только один раз, а наследники — короткие.

Но гораздо интереснее наследование простого класса от шаблона. Вспомним использование статических полей для подсчета объектов класса (см. листинг 4.23). Вставлять в каждый класс поле-счетчик и прописывать код его изменения в конструкторе и деструкторе быстро надоедает — хочется написать код один раз и использовать постоянно. Наследование от общего базового класса, в котором реализован подсчет объектов, не решает проблемы, так как унаследованное статическое поле единственное для всех наследников. Таким образом, выполняется подсчет объектов не каждого класса, а всех объектов иерархии.

Наследование от шаблона, которое на первый взгляд выглядит хитрым трюком, позволяет получить именно то, что требуется: написать код подсчета объектов единственный раз и использовать эту функциональность в каждом классе (листинг 11.13).

Листинг 11.13. Наследование обычного класса от шаблона

```
#include <iostream>
using namespace std;
template<class T>                                // шаблон
class CountedObject
{
    static unsigned int counter;                // счетчик
public:
    // функциональность подсчета объектов
    CountedObject() { ++counter; }
    CountedObject(const CountedObject<T>& t) { ++counter; }
    ~CountedObject() { --counter; }
    static unsigned int getCounter() { return counter; }
};
template <class T>                                // инициализация счетчика
unsigned int CountedObject<T>::counter = 0;
// наследование от шаблона
class Count01: public CountedObject<Count01>
{ // дополнительная нужная функциональность
};
class Count02: public CountedObject<Count02>
{ // дополнительная нужная функциональность
};
int main()
{
    Count01 a,b;
    cout << Count01::getCounter() << endl;    // выводит 2!
    Count02 a2,b3,c2;
    cout << Count02::getCounter() << endl;    // выводит 3!
    return 0;
};
```


Программа считает объекты совершенно правильно — каждый класс отдельно. Следовательно, и класс Count01, и класс Count02 имеют собственные копии поля-счетчика и унаследовали функциональность шаблонного класса. Однако наследование очень напоминает циклическое самоопределение, так как в качестве аргумента шаблона используется сам наследуемый класс! Тем не менее компилятор сумел правильно разобраться в этой головоломке, поскольку в шаблоне ни одно поле (и ни один метод) не зависит от параметра шаблона T. Именно это обстоятельство позволяет вычислить размер класса CountedObject (он равен 0, так как статические поля в размер класса не включаются) без подстановки фактического аргумента! Поэтому любое наследование *от специализации* CountedObject полностью определяется на момент обработки, и никакой рекурсии не возникает. Впервые этот удивительный прием показал Джеймс Коплиен в [26].

Шаблоны и дружелюбность

Естественно, у шаблона могут быть друзья, и шаблоны тоже могут быть друзьями, но тут тоже имеют место некоторые нюансы (см. п. п. 14.5.3 в [1]). Напишем простейший шаблонный класс и определим для него дружелюбную операцию вывода <<. Во-первых, выясняется, что дружелюбная функция должна быть шаблоном¹, так как аргумент для вывода у нее, очевидно, зависит от параметра шаблонного класса. Во-вторых, шаблонность операции надо как-то указать в шаблонном классе. Как это делается, показано в листинге 11.14.

Листинг 11.14. Дружелюбная функция-шаблон

```
#include <iostream>
template <typename T>
class TClass
{   T x;
    public:
        TClass(const T &t): x(t) {}           // конструктор инициализации
        friend std::ostream& operator<< <>(std::ostream &os, const TClass<T> &t);
};
// внешнее определение функции-шаблона
template<class T>
std::ostream& operator<<(std::ostream &os, const TClass<T>&t)
{ return os << '(' << t.x << ')'; }
```

Определение дружелюбной функции-шаблона похоже на определение метода-шаблона, но несколько проще — нет второго заголовка. На то, что дружелюбная операция является шаблоном, указывают пустые скобки <> после имени функции в прототипе, задаваемом в определении класса.

Внешние определения дружелюбных функций-шаблонов имеют одну существенную особенность: автоматические преобразования аргументов не выполняются. Включим в наш пример дружелюбную операцию сложения — тогда наш шаблон будет выглядеть так, как показано в листинге 11.15.

¹ Подробности определения шаблонов функций рассматриваются в следующей главе.

Листинг 11.15. Дружелюбная функция сложения

```
#include <iostream>
template <typename T>
class TClass
{ T x;
public:
    TClass(const T &t):x(t) {}
    friend std::ostream& operator<< <>(std::ostream &os, const TClass<T> &t);
    friend TClass<T> operator+ <>(const TClass<T>&a, const TClass<T>&b);
};
template<class T>
std::ostream& operator<<(std::ostream &os, const TClass<T>&t)
{ return os << '(' << t.x << ')'; }
template<class T>
TClass<T> operator+(const TClass<T>&a, const TClass<T>&b)
{ return TClass<T>(a.x + b.x); }
```

Попробуем использовать эти шаблоны:

```
TClass<int> a(1), d1(2), d2(3);
cout << a << endl;
cout << d1+d2 << endl;
cout << d1+1 << endl;           // ошибка компиляции!
```

Выражение `d1+1` вызывает ошибку компиляции — оказывается, для шаблонов конструктор автоматически не вызывается, как это делается для обычных классов. Решение состоит в том, чтобы определить дружелюбные функции непосредственно внутри класса (листинг 11.16).

Листинг 11.16. Внутреннее определение дружелюбных функций

```
#include <iostream>
template <typename T>
class TClass
{ T x;
public:
    TClass(const T &t):x(t) {}
    friend std::ostream& operator<<(std::ostream &os, const TClass<T> &t)
    { return os << '(' << t.x << ')'; }
    friend TClass<T> operator+(const TClass<T>&a, const TClass<T>&b)
    { return TClass<T>(a.x + b.x); }
};
```

После этого приведенная ранее программа работает правильно. Нужно подчеркнуть, что определенные внутри шаблонного класса `TClass` функции-друзья *не являются функциями-шаблонами* — это обычные функции, несмотря на то, что аргументы у них зависят от параметра шаблона. Кстати, именно это требование — зависимость параметров дружелюбной функции от аргумента шаблона — является обязательным при внутреннем определении дружелюбных функций. Дело в том, что при каждом инстанцировании шаблона `TClass` создается новый экземпляр таких дружелюбных функций. Поэтому, если функция не будет зависеть от параметров шаблонного класса, при нескольких инстанцированиях сгенерируются одинаковые определения. Например, определим в шаблоне `TClass` дружелюбную функцию:

```
friend void f(void)
{ std::cout << x << std::endl; }
```

Теперь попробуем сделать два объявления:

```
TClass<int> a(1), d1(2), d2(3);  
TClass<double> b(1);
```

При наличии этих объявлений возникает ошибка повторного определения функции `f()`.

Резюме

В C++ включено два вида шаблонов: шаблоны функций и шаблоны классов. Основное назначение шаблонов классов — реализация обобщенных контейнеров, не зависящих от типа элемента.

Основной вид параметров шаблона — имя типа. Однако шаблоны классов могут иметь параметры и других видов. В частности, одним из мощнейших средств программирования шаблонов является параметр-шаблон. Кроме того, параметры шаблона класса разрешается задавать по умолчанию. Однако существует и несколько не совсем понятных ограничений, которые, скорее всего, будут сняты в следующем стандарте C++.

Процесс подстановки аргументов на место параметров шаблона называется инстанцированием шаблона. Инстанцирование позволяет создавать из одной «заготовки» целое семейство конкретных реализаций. На основе первичного шаблона можно получать специализированные версии, в которых заданы часть или даже все параметры. Последний вариант называется полной специализацией.

Классы-шаблоны могут быть вложенными, могут быть друзьями, могут наследовать от шаблона или от класса. Обычная практика — определить полную специализацию шаблона для указателей и использовать ее как базовый класс для частично специализированного шаблона по указателям. Интересный прием наследования класса от полной специализации шаблона позволяет организовать счетчик объектов для конкретного класса, а не для всей иерархии наследования.

Дружественные функции могут быть определены как внешние функции-шаблоны или непосредственно внутри класса. В последнем случае дружественная функция является обычной функцией, и ее параметры должны зависеть от параметров шаблона, иначе при неоднократном инстанцировании возникнет ошибка повторного определения.

Контрольные вопросы

1. Для чего предназначены шаблоны?
2. Какие виды шаблонов в C++ вы знаете?
3. Объясните термин «инстанцирование шаблона».
4. В чем разница между определением и объявлением шаблона?
5. Объясните назначение ключевого слова `typename`.
6. Какие виды параметров разрешается задавать в шаблоне класса?

7. Можно ли параметрам шаблона присваивать значения по умолчанию?
8. Может ли параметром шаблона быть другой шаблон? Каковы особенности объявления параметра-шаблона?
9. Что такое специализация шаблона? Объясните разницу между полной и частичной специализацией.
10. Нужно ли задавать определение первичного шаблона для определения специализированных версий, или достаточно объявления?
11. Может ли шаблонный класс быть вложенным в другой шаблонный класс? А в обычный класс?
12. Можно ли объявить в классе шаблонный метод? А шаблонный конструктор?
13. Может ли шаблон класса быть наследником обычного класса? А обычный класс — наследником шаблона?
14. Может ли вложенный класс быть шаблоном?
15. Можно ли обычный класс сделать внутренним классом шаблона? Существуют ли какие-нибудь ограничения в этом случае?
16. Каким образом можно использовать возможность наследования обычного класса от шаблона?
17. Может ли шаблонный конструктор быть конструктором по умолчанию?
18. Может ли шаблонный класс иметь «друзей»?
19. Какие проблемы возникают при объявлении дружественной функции для шаблонного класса?
20. Разрешается ли определять в шаблонном классе статические поля? А статические методы?

Упражнения

1. Переработать шаблон массива `TSimpleArray` (см. листинг 11.4) в динамический массив (см. листинг 11.9).
2. Реализовать класс `TDeque` (см. главу 6) как шаблонный класс.
3. Взяв за основу шаблонный класс `TSimpleArray` (см. листинг 11.4), реализовать шаблон массива с задаваемыми границами индексов (см. упражнение 1 в главе 5).
4. Реализовать шаблон очереди, используя шаблон `TDeque` (см. упражнение 2) как параметр-шаблон.
5. Реализовать шаблон стека, задав поле-шаблон `TSimpleArray` (см. листинг 11.4).
6. Взяв за основу класс `TDeque` (см. главу 6), реализовать шаблон списка `List`, который обеспечивает вставку и удаление произвольного элемента списка по итератору, а также поиск элемента по значению.
7. Взяв за образец класс `TArray` (см. листинг 5.2), реализовать шаблон числового массива с полным набором арифметических операций.

8. Используя шаблон `TSimpleArray` как первичный, реализовать класс `BitString` (см. упражнение 4 в главе 3) как полную специализацию.
9. Реализовать класс `DoubleArray` (см. упражнение 2 в главе 5) как класс-наследник от шаблона `TSimpleArray`.
10. Реализовать класс `Set` (см. упражнение 7 в главе 5) в виде шаблона.
11. Реализовать динамический числовой массив (см. упражнение 7) как абстрактный шаблонный класс, объявив арифметические операции чистыми виртуальными функциями. Реализовать класс `DoubleArray` (см. упражнение 2 в главе 5) как класс-наследник, а класс `BitString` (см. упражнение 11 в главе 1) — как специализацию этого шаблона.

Глава 12

Шаблоны функций

Очевидно, что некоторые функции должны «уметь» работать с параметрами любых типов. Например, функция, сортирующая массив, должна «уметь» сортировать массив любого типа. Для решения этой проблемы в C++ можно применить перегрузку функций. Однако на практике этот механизм — не совсем то, что требуется. В самом деле, если мы попробуем написать полный набор перегруженных функций `swap()` для обмена двух значений элементарных типов, то потребуется реализовать 3 варианта для символов (обычных и широких), 6 вариантов для различных целых и 3 варианта для дробных. Итого — 12 вариантов. Тем не менее для каждого реализованного класса функцию обмена придется писать заново. И все эти функции, даже если они реально не используются, должны присутствовать в программе, чтобы программист не задумывался о типах обмениваемых объектов.

Функции-шаблоны позволяют сократить эту утомительную рутинную работу — достаточно написать один шаблон функции и вызывать его потом с нужными типами аргументов. В работающую программу попадут только те варианты функций, которые реально были вызваны для выполнения. Таким образом, шаблоны функций не только сокращают исходный текст, но и уменьшают объем выполняемого файла.

В предыдущей главе мы уже столкнулись с функциями-шаблонами: любой метод класса может быть сделан шаблонным (см. листинг 11.10), и дружественная функция класса может быть реализована как шаблонная (см. листинги 11.14 и 11.15). Здесь мы рассмотрим подробности и выясним детали использования функций-шаблонов.

Определение шаблона функции

Для знакомства с шаблонами функций (см. п. п. 14.5.5 в [1]) напомним шаблон функции суммирования последовательности элементов массива. Параметром шаблона можно сделать, очевидно, тип элементов массива, а параметрами функции

будут, как обычно, два указателя: указатель на первый суммируемый элемент последовательности и указатель на первый элемент, не включаемый в сумму. Напомню, этот подход принят в стандартной библиотеке шаблонов¹, и мы тоже уже применяли подобный подход при реализации конструкторов классов TString (см. листинг 4.2) и TArray (см. листинг 5.4). Шаблон функции суммирования представлен в листинге 12.1.

Листинг 12.1. Шаблон функции суммирования элементов массива

```
template <typename T>
T Summa(T const *begin, T const *end)
{ T total = T(); // инициализация нулем
  while (begin != end)
  { total += *begin;
    ++begin;
  }
  return total;
}
```

Как видно, шаблон функции так же, как и метод класса, должен начинаться с заголовка шаблона, в котором задаются его параметры. После заголовка следует обычное определение функции, в котором на месте типа стоит имя параметра-типа. Обнуление переменной для суммирования осуществляется с помощью уже известной нам конструкции инициализации нулем. Использовать данный шаблон можно следующим образом:

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
cout << Summa(a, a+10) << endl; // суммирование массива типа int
double b[10] = {1.1,2,3,4,5,6,7,8,9,10.1};
cout << Summa(b, b+10) << endl; // суммирование массива типа double
```

Сразу бросается в глаза, что при вызове шаблона функции не заданы аргументы шаблона в угловых скобках <> — вызов функции-шаблона не отличается от вызова обычной функции. В этом состоит принципиальное отличие шаблонов функций от шаблонов классов — для шаблонов классов аргументы нужно задавать всегда явно. В данном случае компилятор самостоятельно *выводит* подставляемый тип на основании информации о типе фактического аргумента при вызове.

В случае инстанцирования шаблона при первом вызове (для массива `int a[10]`) компилятор генерирует конкретный вариант функции:

```
int Summa(int const *begin, int const *end)
{ int total = int(); // инициализация нулем
  while (begin != end)
  { total += *begin;
    ++begin;
  }
  return total;
}
```

¹ В стандартной библиотеке параметры — это итераторы, частным случаем которых являются указатели.

При втором вызове (для массива `double b[10]`) инстанцируется функция

```
double Summa(double const *begin, double const *end)
{ double total = double();           // инициализация нулем
  while (begin != end)
  { total +=*begin;
    ++begin;
  }
  return total;
}
```

Далее компилятор «украшает» имена функций как при обычной перегрузке.

Наш шаблон был написан в предположении, что будут суммироваться элементы числовых массивов. Однако он получился более универсальным, чем задумывалось. Шаблон правильно работает для любого класса, в котором реализован конструктор без аргументов и операция `operator+` — это не обязательно операция сложения. Например, корректно работает вызов с аргументом типа `string`:

```
string d[4] = {"1+", "2+", "3+", "4"};
cout << Summa(d, d+4) << endl;
```

Результатом является строка `"1+2+3+4"`. Стандартный класс `string` имеет конструктор без аргументов, а операция «сложения» является операцией сцепления.

Несмотря на то что компилятор самостоятельно выводит тип аргумента шаблона, вполне можно задать его и в явном виде, например:

```
cout << Summa<double>(b, b+10) << endl;
```

Кроме того, вывод типов не работает для типа возвращаемого значения, поэтому такой аргумент при вызове функции-шаблона всегда нужно указывать явно. Рассмотрим этот вопрос подробнее. Наш шаблон содержит одну ошибку, которую можно продемонстрировать следующим образом:

```
short s[2]= { 20000,20000 };
cout << Summa(s, s+2) << endl;
```

Результат, выводимый на экран, отрицательный, и равен `-25 536`. Произошло обычное переполнение, так как результат операции $20\,000 + 20\,000 = 40\,000$ не помещается в переменную типа `short`. И такая ситуация возникает каждый раз, когда значения элементов массива находятся на границе диапазона. Нам нужно модифицировать шаблон таким образом, чтобы результат помещался в более «мощный» тип, чем тип аргументов. Однако в C++ отсутствуют стандартные средства, позволяющие выбрать «более мощный тип».

СОВЕТ

Проблему можно решить с помощью некоторых приемов программирования шаблонов [20, 21, 28].

Поэтому надо добавить в шаблон `Summa` еще один параметр — тип возвращаемого значения (листинг 12.2).

Листинг 12.2. Шаблон функции с параметром — типом возвращаемого значения

```
template <typename RT, typename T>
RT Summa(T const *begin, T const *end)
{ RT total = RT(); // инициализация нулем
  while (begin != end)
  { total +=*begin;
    ++begin;
  }
  return total;
}
```

Теперь вызовы нужно писать с явным указанием аргумента для типа возвращаемого значения, например:

```
int a[10]= {1,2,3,4,5,6,7,8,9,10};
cout << Summa<long>(a, a+10) << endl;
double b[10]= {1.1,2,3,4,5,6,7,8,9,10.1};
cout << Summa<double>(b, b+10) << endl;
string d[4] = {"1+", "2+", "3+", "4"};
cout << Summa<string>(d, d+4) << endl;
short s[2]= { 20000,20000 };
cout << Summa<int>(s, s+2) << endl;
```

Попытка вызвать функцию-шаблон без аргумента приводит к ошибке трансляции, например:

```
cout << Summa(d, d+4) << endl;
```

Дело в том, что тип возвращаемого значения *не выводится* компилятором автоматически, в отличие от типов параметров.

Обратите внимание, что параметр шаблона RT для возвращаемого значения функции указан первым. Это позволяет нам при вызове не задавать аргументы для остальных параметров — их вычисляет компилятор. Перепишем шаблон, переставив параметры:

```
template <typename T, typename RT>
RT Summa(T const *begin, T const *end)
{ RT total = RT(); // инициализация нулем
  while (begin != end)
  { total +=*begin;
    ++begin;
  }
  return total;
}
```

Теперь при вызове придется задавать *все* параметры явно, например:

```
int a[10]= {1,2,3,4,5,6,7,8,9,10};
cout << Summa<int, long>(a, a+10) << endl;
```

Попробуем написать вызов такого шаблона функции с одним аргументом:

```
cout << Summa<long>(a, a+10) << endl;
```

В этом случае выдается ошибка компиляции (Visual C++.NET 2003):

```
error C2783: 'RT Summa(const T *,const T *)':
could not deduce template argument for 'RT'
```

В некоторых случаях необходимость явным образом задавать аргумент шаблона для возвращаемого значения может быть полезной. Например, можно реализовать «оператор» неявного преобразования таким образом:

```
template <typename RT, typename T>
inline
implicit_cast(const T &t)
{ return RT(t); }
```

Вызов такого шаблона выглядит абсолютно так же, как и вызов стандартных операторов преобразования типа, например:

```
int a = 1;
float b = implicit_cast<float>(i);
```

Но правильно работает такое преобразование, естественно, только для разрешенных неявных преобразований.

Параметры по умолчанию

К сожалению, в шаблонах функций нельзя задавать параметры по умолчанию, хотя мы можем написать заголовок шаблона функции суммирования, например:

```
template <typename RT = double, typename T = float>
```

Visual C++.NET 2003 никак не «реагирует» на такое определение, однако при вызове без указания аргументов шаблона выдает ту же ошибку C2783. C++ Builder 6 «реагирует» на объявление шаблона сразу, не дожидаясь вызова. Сообщение об ошибке E2408 явно указывает на недопустимость присвоения значений по умолчанию параметрам шаблона функции:

Default values may be specified only in primary class template declarations

Сообщение гласит, что значения по умолчанию могут быть определены только в объявлении шаблона класса. Однако это ограничение легко обойти, используя класс-оболочку (листинг 12.3). Функция при этом объявляется статическим методом класса.

Листинг 12.3. Класс-шаблон — оболочка функции

```
template <typename RT= double, typename T = float>
class Function
{ public:
    static RT Summa(T const *begin, T const *end)
    { RT total = RT(); // инициализация нулем
      while (begin != end)
      { total +=*begin; ++begin; }
      return total;
    }
};
```

Вызов такой функции должен сопровождаться префиксом класса-шаблона:

```
float f[10]= {1.1,2.3,4.5,6.7,8.9,10.1};
cout << Function<>::Summa(f, f+2) << endl;
```

```
int L[10]= {1,2,3,4,5,6,7,8,9,10};
cout << Function<float,int>::Summa(L, L+10) << end
```

Параметры шаблона — не типы

Параметры, не являющиеся типами, можно использовать и в шаблонах функций. Например, можно написать шаблон функции, обнуляющий любой двумерный числовой массив:

```
template <std::size_t M, std::size_t N, typename T>
void InitArray(T array[M][N])           // передача массива
{ for(std::size_t i = 0; i<M; ++i)
  for(std::size_t j = 0; j<N; ++j)
    array[i][j] = T();
}
```

Вызов такой функции-шаблона может осуществляться, например, так:

```
const int m = 2; const int n = 3;
int g[m][n];
InitArray<m,n>(g);                       // явное задание аргументов
```

Однако такие параметры иногда тоже могут быть выведены компилятором, и их можно не задавать при вызове. Нужно немного изменить шаблон, задав передачу массива по ссылке:

```
template <std::size_t M, std::size_t N, typename T>
void InitArray(T (&array)[M][N])        // передача массива по ссылке
{ for(std::size_t i = 0; i<M; ++i)
  for(std::size_t j = 0; j<N; ++j)
    array[i][j] = T();
}
```

Тогда вызов с массивом `g` в качестве параметра ничем не отличается от вызова обычной функции:

```
InitArray(g);                            // аргументы вычисляются компилятором
```

Перегрузка шаблонов функций

Допускается перегрузка шаблонов функций как шаблонами, так и обычными функциями (см. п. п. 14.5.5.1 в [1]), например:

```
template<typename T>                               // шаблон 1
inline
T const& max(T const& x, T const& y)
{ return (x > y)?x:y; }
template<typename T>                               // шаблон 2
inline
T const& max(T const& x, T const& y, const T& z)
{ return max(max(x,y),z); }
char *max(const char *x, const char *y)           // функция
{ return (strcmp(x,y) > 0)?x:y; }
double& max(const double& x, const double& y)    // функция
{ return (x > y)?x:y; }
```

Как и при перегрузке обычных функций, шаблоны (и функции) должны различаться списками параметров. Компилятор при обработке конкретного вызова старается подобрать наиболее подходящий вариант, например:

```
char *s1 = "0123456789", *s2 = "1234567890";
float a = 1.0, b = 2.0;
cout << max(1,2) << endl;           // шаблон 1
cout << max(1.0,2.0) << endl;        // функция с аргументами типа double
cout << max(1.0,2.0, 3.0) << endl;    // шаблон 2
cout << max(1,2.0) << endl;          // функция с аргументами типа double
cout << max(a,b) << endl;            // шаблон 1
cout << max(s1,s2) << endl;          // функция с аргументами типа char*
```

Мы можем «заставить» компилятор вызвать шаблон с аргументами типа `char*`. Для этого просто нужно записать угловые скобки в вызове, например:

```
cout << max<>(s1,s2) << endl;        // шаблон 1 с аргументами типа char*
```

Однако следует отдавать себе отчет, что в этом случае сравниваются не строки, а указатели!

Специализация шаблонов функций

Для шаблонов функций, как известно, параметры по умолчанию присваивать не разрешается, однако специализировать шаблон можно (см. п. 14.8 в [1]). Но и в этом случае есть ограничение — частичная специализация функций запрещена, разрешается только полная специализация.

СОВЕТ

В этом случае тоже можно «обернуть» функцию в класс-шаблон и использовать частичную специализацию класса-шаблона.

Например, вместо рассмотренной ранее функции `max()` с параметрами-указателями на символы можно специализировать исходный первичный шаблон (листинг 12.4).

Листинг 12.4. Специализация шаблона функции

```
template<typename T>           // шаблон
inline
T const& max(T const& x, T const& y)
{ return (x > y)?x:y; }
template<>                     // специализация шаблона
inline
char* const& max(char* const& x, char* const& y)
{ return (strcmp(x,y) > 0)?x:y; }
```

Специализация начинается с того же ключевого слова `template` с пустыми угловыми скобками. Вообще-то для полной специализации требуется указывать специализирующие параметры после имени шаблона, например:

```
template<>                     // специализация шаблона
inline
```

```
char* const& max<char*>(char* const& x, char* const& y)
{ return (strcmp(x,y) > 0)?x:y;      // сравнение строк
}
```

Но для шаблонов функций этого практически всегда можно не делать, поскольку компилятор может вывести тип самостоятельно. Вызов специализированной версии демонстрирует следующий фрагмент программы:

```
char *s1 = "0123456789", *s2 = "1234567890";
float a = 1.0, b = 2.0;
cout << max(1,2) << endl;           // шаблон с типом int
cout << max(1.0,2.0) << endl;       // шаблон с типом double
cout << max(a,b) << endl;          // шаблон с типом float
cout << max<>(s1,s2) << endl;       // специализация с типом char*
```

Реализация обобщенных алгоритмов

Давайте напомним функцию-фильтр для решения следующей задачи: выбрать из последовательного контейнера положительные элементы и скопировать их в другой контейнер. Мы писали подобную функцию для «умного» массива (см. листинг 5.19), однако она была недостаточно обобщенная, так как предназначалась только для обработки объектов класса `TArray`.

Очевидно, что первым шагом обобщения являются шаблоны — мы должны написать шаблон функции-фильтра, чтобы не зависеть от типа элементов контейнера. Однако этого недостаточно. Если мы будем передавать контейнер в качестве параметра и возвращать новый контейнер как результат, наша функция-фильтр будет зависеть от типа контейнера. Например, напомним фильтр с таким заголовком:

```
template <class T>
TArray<T> copy_if(const TArray<T>& m, const T& t)
```

Мы сразу написали обобщенный вариант — сравнение элементов контейнера будет выполняться не с нулем, а с произвольным значением. Однако совершенно очевидно, что такая функция не в состоянии обрабатывать контейнеры другого типа. Кроме того, мы не сможем обрабатывать с помощью такой функции массивы.

К счастью, решение нам уже известно — итераторы. Входной контейнер, в котором должен осуществляться поиск, представляется в виде пары итераторов, задающей полуоткрытый интервал `[begin, end)`. Однако и выходной контейнер нельзя задавать как результат — зависимость от типа контейнера останется. Поэтому лучше и выходной контейнер задать итераторами. Нам достаточно одного итератора — начального. Тогда функция может либо вообще не возвращать результат, либо вернуть выходной итератор. Естественно, типы итераторов являются параметрами шаблона, как и тип элементов контейнера.

Такое решение сразу накладывает некоторые ограничения на использование функции-фильтра:

1. Чтобы задать итератор выходного контейнера при вызове нашей функции, выходной контейнер должен существовать на момент вызова.

2. В контейнере должны существовать элементы и их должно быть достаточное количество, чтобы вместить все удовлетворяющие критерию отбора элементы входного контейнера.
3. Выходной контейнер должен быть совместим по типу элементов с входным. Таким образом, в отношении выходного контейнера мы вынуждены полностью полагаться на программиста. Однако «овчинка стоит выделки» — мы получаем алгоритм, который действительно не зависит от входного и выходного контейнеров (листинг 12.5).

Листинг 12.5. Функция-фильтр копирования по условию

```
template < class InputIterator,           // входной контейнер
          class OutputIterator,         // выходной контейнер
          class T                       // тип элементов
        >
void copy_if(InputIterator first, InputIterator last,
             OutputIterator result,
             const T &v
            )
{ for ( ; first != last; ++first)
    if (*first > v)
        { *result = *first; ++result; }
    return;
}
```

Этот алгоритм совершенно правильно работает и с массивами, и с контейнерами, обеспечивающими последовательный доступ с помощью итераторов.

ПРИМЕЧАНИЕ

Алгоритм работает и с контейнерами стандартной библиотеки шаблонов.

Однако он все-таки не является универсальным — критерий отбора элементов задан жестко в теле функции в виде условия оператора `if`. Алгоритм стал бы значительно мощнее, если бы и критерий отбора можно было задавать в виде параметра.

Это можно сделать, определив еще один параметр — указатель на функцию. Очевидно, что эта функция должна вызываться в условии оператора `if`, а поэтому возвращать должна результат типа `bool`¹. Функция, возвращающая результат типа `bool`, называется *предикатом*. Функция-предикат с одним параметром называется *унарным* предикатом; функция-предикат с двумя параметрами — *бинарным* предикатом. Очевидно, наша функция — критерий отбора — является бинарным предикатом, так как вызов ее в условии оператора `if` может выглядеть так:

```
if(f(v, *first)) ...
```

¹ Конечно, это не обязательно, но мы сейчас говорим о «правильном» использовании `copy_if`.

Перепишем нашу функцию-фильтр и напомним заодно пару функций-предикатов для проверки ее работы (листинг 12.6).

Листинг 12.6. Функция-фильтр с параметром-функцией

```
template < class InputIterator,
           class OutputIterator,
           class T
         >
void copy_if( InputIterator first, InputIterator last,
              OutputIterator result,
              bool f(const T &a, const T &b),
              const T &v
            )
{ for ( ; first != last; ++first)
    if (f(v, *first))
    { *result = *first; ++result; }
  return;
}

// функции-предикаты
template <class T>
bool lessf(const T &a, const T &b)
{ return (a<b); }
bool greaterf(const int &a, const int &b)
{ return a > b; }
// главная функция
int main()
{ int a[10] = { 1,2,3,4,5,6,7,8,9,0};
  int b[10] = {0};
  copy_if(a, a+10, b, lessf<int>, 5);
  for(int i = 0; i < 10; ++i)
    cout << b[i] << ' ';
  cout << endl;
  copy_if(a, a+10, b, greaterf, 7);
  for(int i = 0; i < 10; ++i)
    cout << b[i] << ' ';
  cout << endl;
  getchar();
  return EXIT_SUCCESS;
}
```

Эта программа выводит на экран следующее:

```
6 7 8 9 0 0 0 0 0 0
1 2 3 4 5 6 0 0 0 0
```

Первая строка — это результат вызова:

```
copy_if(a, a+10, b, lessf<int>, 5);
```

Вторая строка соответствует вызову:

```
copy_if(a, a+10, b, greaterf, 7);
```

Как видите, наша обобщенная функция-фильтр прекрасно работает и с обычной функцией, и с шаблоном функции. Аргумент шаблона должен совпадать с типом элементов массива. Попробуем задать другой тип, например:

```
copy_if(a, a+10, b, lessf<double>, 5);
```

В результате возникает ошибка трансляции — вспомним, что автоматическое преобразование типов в шаблонах не работает.

Разберемся теперь в достоинствах и недостатках данного подхода. Очевидным достоинством является дальнейшее обобщение функции-фильтра: мы теперь можем задать ей в качестве параметра любой бинарный предикат.

Однако обобщения на этом не заканчиваются. На мой взгляд, тип передаваемой функции излишне «жестко» зафиксирован. Кроме того, последний параметр тоже выглядит лишним — именно он придает «жесткость» типу функции-предиката. Стоит подумать, чтобы сократить количество параметров до четырех — ведь нужен только входной контейнер, выходной контейнер и функция-предикат для отбора элементов. Рассмотрим, как это можно сделать. Параметр, записанный следующим образом, интерпретируется как указатель на функцию:

```
bool f(const T &a, const T &b)
```

Можно записать этот параметр явно, как указатель:

```
bool (*f)(const T &a, const T &b)
```

Этот вариант эквивалентен предыдущей записи. Последний вариант наводит на мысль о применении ключевого слова `typedef`:

```
typedef bool (*Predicate)(const T &a, const T &b);
```

Тип `Predicate` можно указать в качестве типа параметра-предиката. Однако значительно лучше написать аналогичный параметр шаблона:

```
template < class InputIterator,
          class OutputIterator,
          class Predicate,
          class T
        >
void copy_if( InputIterator first, InputIterator last,
             OutputIterator result,
             Predicate function,
             const T &v
           )
```

Такая запись разрешается, так как параметром шаблона может быть указатель на функцию (см. п. п. 14.1/4 в [1]).

Теперь подумаем, какой должна быть эта функция-предикат. Хотя это выглядит парадоксальным, но унарный предикат делает нашу функцию-фильтр более универсальной и позволяет избавиться от последнего параметра (листинг 12.7).

Листинг 12.7. Алгоритм с унарным предикатом

```
template < class InputIterator, class OutputIterator,
          class Predicate
        >
void copy_if( InputIterator first, InputIterator last,
             OutputIterator result,
             Predicate function
```


Листинг 12.7 (продолжение)

```

    )
{ for ( ; first != last; ++first)
    if (function(*first))
        { *result = *first; ++result; }
    return;
}

```

Наш предикат имеет один параметр — элемент контейнера. Все действия и другие необходимые величины предикат инкапсулирует внутри себя. Например:

```

int odd(const int &a)
{ return (a%2); }

```

Этот предикат определяет нечетность аргумента. Еще пример:

```

bool negative(const int &a)
{ return (a<0); }

```

Этот предикат служит для отбора отрицательных элементов контейнера. А если нам требуется, чтобы все элементы контейнера были больше пяти, мы напишем такой предикат:

```

bool gt5(const int &a)
{ return (a>5); }

```

Хотя подобное представление предиката несколько ухудшает читабельность кода, поскольку о назначении предиката можно судить только по названию функции, оно никак не ограничивает функциональность. Более того, указание предиката как параметра шаблона позволяет задавать практически любые функции с одним аргументом. Единственное ограничение — функция должна возвращать хоть какой-то результат, так как вызов стоит в условии оператора `if`. Например, мы можем с помощью передаваемой функции умножить все элементы массива на 2:

```

int twice(int &a)
{ return a*=2; }
//...
int a[10] = { 1,2,3,4,5,6,7,8,9,0};
int b[10] = {0};
copy_if(a, a+10, b, twice);
for(int i = 0; i < 10; ++i)
    cout << b[i] << ' ';
cout << endl;

```

Этот фрагмент выведет на экран следующее:

```
2 4 6 8 10 12 14 16 18 0
```

Указатели на функции и указатели на методы

Все выглядит великолепно. Однако мы знаем, что «бесплатного сыра не бывает». Расплачиваться приходится эффективностью: вызов предиката выполняется для каждого элемента контейнера — каждый раз формируется стек параметров,

функция вызывается, а потом выполняется возврат из нее. Для больших контейнеров это может стать серьезной проблемой. Для повышения эффективности вызовов в C++ используют ключевое слово `inline`. Тело функции, объявленной с этим ключевым словом, может быть подставлено на место вызова — экономится время на заполнение стека, вызов функции и возврат из нее. Как правило, предикаты — это простые функции, поэтому являются хорошими кандидатами на подстановку. Но в данном случае объявлять предикаты подставляемыми бесполезно, так как вызов является не прямым, а косвенным — через указатель. Компилятор не может подставить функцию, вызов которой является косвенным. Есть еще одна, более важная причина, по которой нельзя считать нашу функцию-фильтр `sort_if()` достаточно универсальной: мы не можем передать функции-фильтру предикат-метод. Дело в том, что указатель на метод существенно отличается по типу от обычного указателя на функцию, даже если прототипы функции и метода внешне идентичны. Вспомним, как определяется указатель на функцию:

```
тип (*имя-указателя)(список параметров);
```

Для сокращения записи применяют обычно оператор `typedef`. Следующее объявление вводит новое имя типа:

```
typedef тип (*тип-указателя)(список параметров);
```

После этого указатель на функцию можно объявлять так:

```
тип-указателя имя-указателя;
```

Указателю присваивается адрес функции. В C++ разрешается присваивать адрес функции, явно или неявно задавая операцию получения адреса:

```
имя-указателя = &имя-функции;
имя-указателя = имя-функции;
```

В последнем случае имя функции неявно преобразуется в адрес.

ПРИМЕЧАНИЕ

В системе Visual C++.NET 2003 для библиотечных функций работает только вариант с явной операцией адресации. Второй вариант вызвал фатальную ошибку компилятора `fatal error 1001: INTERNAL COMPILER ERROR`. Borland C++ Builder 6 оба вызова обрабатывает правильно.

Вызов функции через указатель выполняется так:

```
(*имя-указателя)(аргументы);
```

Тот же вызов можно написать в более простой форме:

```
имя-указателя(аргументы);
```

Рассмотрим несколько простых примеров. Допустим, у нас объявлен указатель на функцию:

```
int (*pf)(void);
```

Этому указателю можно присвоить адрес любой функции с таким же прототипом, в том числе и библиотечной. Например, такой прототип имеет функция-генератор случайного числа `rand()`, который объявлен в заголовке библиотеки `<cstdlib>`:

```
int f1(void) { return 1; }
int f2(void) { return 2; }
pf = f1;      cout << pf() << endl;      // вызов f1()
pf = f2;      cout << pf() << endl;      // вызов f2()
pf = &rand;   cout << pf() << endl;      // вызов rand()
```

Напишем теперь простой класс

```
class Constant
{
    int value;
public:
    Constant(const int &a): value(a) {}
    int get(void) { return value; }
};
```

Метод `get()` с виду имеет тот же прототип, что и функции в приведенном ранее фрагменте. Однако попытки присвоить адрес метода `get()` указателю `pf` компилятор пресекает «на корню». Да и не совсем понятно, как этот адрес задавать. Вариантов два.

1. Объявить объект и взять адрес метода в объекте:

```
Constant A(5); pf = &A.get();
```

2. Не объявлять объект, а приписать префикс класса:

```
pf = &Constant::get;
```

Первый вариант вообще неверный: и Visual C++.NET 2003, и Borland C++ Builder 6 сообщают, что операцию взятия адреса `&` так использовать нельзя. Второй вариант ближе к истине: оба компилятора сообщают только о невозможности выполнить преобразование типов. Попытки прописать преобразование явно не проходят:

```
pf = static_cast<int (*) (void)>(&Constant::get);
pf = reinterpret_cast<int (*) (void)>(&Constant::get);
pf = (int (*) (void))(&Constant::get);
```

Все эти варианты вызывают ту же ошибку компиляции — невозможность преобразования типов. Таким образом, тип указателя на метод класса кардинально отличается от типа указателя на функцию: адрес метода нельзя присвоить указателю на функцию, даже если внешне их прототипы совпадают. Это становится понятным, если мы вспомним, что нестатические методы получают дополнительный параметр — указатель `this`.

Аналогично, нельзя присвоить обычному указателю на функцию адрес виртуального метода — в этом случае дело усугубляется еще наличием в составе объекта указателя на таблицу виртуальных методов. А вот то статическими методами картина другая! Статический метод не получает никаких «лишних параметров», поэтому его адрес можно присваивать обычному указателю на функцию без всяких

преобразований. Добавим в класс `Constant` статическое поле и статический метод с нужным нам прототипом:

```
class Constant
{
    int value;
    static int d;
public:
    Constant(const int &a): value(a) {}
    int get(void) { return value; }
    static int getd(void) { return d; }           // статический метод
};
```

Тогда нашему указателю `pf` можно присвоить адрес статического метода согласно второму из приведенных ранее вариантов, например:

```
pf = Constant::getd;           // или pf = &Constant::getd;
```

Префикс, естественно, необходимо писать.

Указатель на метод объявляется по-другому (см. п. п. 8.3.3 в [1]) — нужно задать имя класса в качестве префикса:

```
int (Constant::*pm)(void);
```

Такому указателю можно присваивать адреса обычных и виртуальных методов согласно второму из приведенных ранее вариантов, например:

```
pm = &Constant::get;           // или pm = Constant::get;
```

Адрес статического метода, так же как и адрес обычной функции, такому указателю присвоить нельзя — возникает ошибка трансляции: компилятор сообщает о невозможности выполнить преобразование типов.

И косвенный вызов метода выполняется по-другому — с помощью операции выбора члена класса `.*` или `->*` (см. п. 5.5 в [1]). Несмотря на то что указатель на член класса является отдельной независимой от класса переменной, вызов метода по указателю возможен только при наличии объекта, например:

```
Constant A(5);
cout << (A.*pm)() << endl;
```

Выражение `(A.*pm)()` означает следующее: для объекта `A` вызвать метод, чей адрес записан в указателе `pm`. Слева от операции `.*` — объект, справа — указатель на метод. Скобки вокруг выражения `A.*pm` писать обязательно, так как приоритет операции вызова функции `()` выше, чем приоритет операции выбора члена класса `.*`.

В выражении (объект.`*указатель`) можно заменить часть объект. записью указатель-`>`, например:

```
Constant *pc = new Constant(7);
cout << (pc->*pm)() << endl;
```

Обратите внимание: в выражении `(pc->*pm)` слева — обычный указатель на динамический объект, а справа — указатель на метод этого объекта. Это два совершенно разных типа указателя.

Интересно, что в Visual C++.NET 2003 размер `sizeof(pf)` указателя на функцию совпадает с размером `sizeof(pm)` указателя на член класса и равен 4 байтам. А вот система Borland C++ Builder 6 выдает совершенно разные цифры: размер указателя на функцию равен 4 байтам, а размер указателя на метод — 12 байт!

Понятие функтора

Таким образом, поскольку в функции `copy_if()` вызов функции-предиката имеет вид вызова обычной функции, передавать методы в качестве аргумента нельзя. Ограничение достаточно существенное, поэтому хотелось бы от него избавиться.

Первое решение, которое приходит в голову, состоит в том, чтобы передавать в функцию-фильтр `copy_if()` не указатель на функцию, а объект, в котором реализована соответствующая функция в виде метода. Изменений в шаблоне функции-фильтра это потребует минимальных — достаточно задать вызов предиката как вызов метода (листинг 12.8).

Листинг 12.8. Предикат как объект

```
class D // класс-оболочка функции
{ public:
    bool function(const int &a) const
    { return a%2; }
};
template < class InputIterator,
           class OutputIterator,
           class Predicate
           >
void copy_if( InputIterator first, InputIterator last,
              OutputIterator result,
              const Predicate &Obj
              )
{ for ( ; first != last; ++first)
    if (Obj.function(*first)) // вызов метода
    { *result = *first; ++result; }
    return;
}
```

Это решение нас устроить не может, так как практически не отличается от предыдущего — мы только сменили один вид параметра на другой. Теперь в предикат невозможно передать указатель на функцию. Кроме того, нельзя забывать, что метод в классе должен называться всегда одинаково — тем именем, которое указано в вызове. И еще одно замечание: мы объявили метод `function()` константным, так как параметр передается по константной ссылке. Если нас это не устраивает, то нужно передавать параметр по значению, как в предыдущем варианте функции `copy_if()`.

Необходимо такое решение, которое действительно было бы универсальным и обеспечивало работу и с функциями, и с методами. Для этого нам нужно сначала разобраться с перегрузкой операции `operator()`, реализующей вызов функции в классе.

Мы уже перегружали эту операцию для индексирования динамических массивов (см. главу 5). Однако в данном случае мы перегрузим ее для использования «по прямому назначению» — как операцию вызова функции.

Традиционно класс, в котором перегружена операция `operator()`, называют *функциональным классом*, или *классом-функтором*. Объект такого класса называют *объектом-функцией*, *функциональным объектом* (см. п. 20.3 в [1]), или просто *функтором*. Функтор представляет собой объект, который ведет себя, как функция.

ПРИМЕЧАНИЕ

Как справедливо отмечено в [28], таковыми являются не только функциональные объекты, но и указатели на функции и методы. Однако мы будем придерживаться традиционного взгляда, считая функторами только объекты-функции.

Перепишем функцию определения нечетности как класс-функтор (листинг 12.9).

Листинг 12.9. Класс-функтор Odd

```
class Odd
{ public:
    int operator()(const int& d)
    { return (d%2); }
};
```

Как видите, по сравнению с функцией изменения выглядят чисто косметическими: имя функции заменено на `operator()`, и сама функция «обернута» в класс, как в листинге 12.8. Однако последствия таких минимальных изменений колоссальны! Так как мы имеем класс, то можем использовать разнообразнейшие возможности объектно-ориентированного подхода, например:

- ❑ реализовать в классе любые необходимые методы и операции, в том числе перегрузить операцию `operator()` несколько раз;
- ❑ объявить в классе любые поля для сохранения состояния функтора (в разные моменты времени функтор может иметь разные состояния, следовательно, перед использованием функтор можно инициализировать);
- ❑ если необходимо, то можно построить иерархию функторов, в том числе и с виртуальными функциями — операция `operator()` тоже может быть сделана виртуальной.

Функторы обладают еще одной важной особенностью по сравнению с указателями на функции. Тип обычной функции (и указателя на функцию) определяется только прототипом. Если прототипы одинаковы, то и тип один и тот же. Объекты функций могут быть разного типа даже при идентичных прототипах операции `operator()` — просто назовите классы по-разному. Это открывает возможности унифицированного программирования с применением шаблонов, так как позволяет передавать поведение как параметр шаблона. Именно эта особенность позволяет реализовать паттерн Strategy (стратегия) на этапе компиляции [17].

Синтаксис вызова функтора такой же, как и обычной функции, только вместо имени функции задается имя класса-функтора. Пусть у нас объявлен объект-функция:

```
Odd f;
```

Тогда вызов записывается так:

```
f(5);
```

Это сокращенная запись, полная выглядит более громоздко:

```
f.operator()(5);
```

Наша обобщенная функция-фильтр `copy_if()` с вызовом функтора показана в листинге 12.10.

Листинг 12.10. Вызов функтора в `copy_if()`

```
template < class InputIterator,
           class OutputIterator,
           class Predicate
         >
void copy_if( InputIterator first, InputIterator last,
              OutputIterator result,
              Predicate Functor
            )
{ for ( ; first != last; ++first)
    if (Functor(*first))                // вызов функтора
        { *result = *first; ++result; }
    return;
}
// проверка работы функтора
int main()
{   int a[10] = { 1,2,3,4,5,6,7,8,9,0};
    int b[10] = {0};
    copy_if(a, a+10, b, Odd());
    for(int i = 0; i < 10; ++i)
        cout << b[i] << ' ';
    cout << endl;
}
```

Как видим, определение функции-фильтра с точностью до имен совпадает с определением в листинге 12.7. Вызов функтора в теле функции `copy_if()` совпадает с вызовом функции в листинге 12.7:

```
Functor(*first)
```

Аргумент-функтор в списке передаваемых параметров задан в виде `Odd()`. Это — просто вызов конструктора и создание анонимного объекта-функции типа `Odd`. Можно объект создать явно и передать его в `copy_if()`, например:

```
Odd f;
copy_if(a, a+10, b, f);
```

Результат — тот же.

Обратите внимание на то, что в листингах 12.7 и 12.10 функции-фильтры `copy_if()` абсолютно идентичны. С практической точки зрения это означает, что мы можем передавать в качестве аргумента-предиката, как и раньше, обычную

функцию! Например, мы получим те же самые результаты, используя приведенную ранее функцию-предикат `odd()`:

```
copy_if(a, a+10, b, odd);
```

Таким образом, мы еще более обобщили нашу функцию-фильтр.

Первым приятным следствием такого обобщения является то, что теперь можно задавать предикаты с несколькими параметрами — просто в классе задается нужное количество полей соответствующих типов. Функтор, имеющий поля, называют *функтором с состоянием*. Заполнение этих полей выполняет конструктор, который может выполнять сколь угодно сложную обработку. Напишем класс-функтор, обеспечивающий сравнение элемента контейнера с любым значением на «больше» (листинг 12.11).

Листинг 12.11. Функтор с несколькими параметрами

```
class Greater
{ int value;
public:
    Greater(const int &v): value(v) {}
    bool operator()(const int& d)
    { return (d > value); }
};
```

Вызов выполняется так:

```
copy_if(a, a+10, b, Greater(3));
```

Здесь мы имеем существенный выигрыш в универсальности (и в читабельности) по сравнению с функциями — вспомните функцию `gt5()`. Ранее мы были вынуждены упрятывать все необходимые величины в функции-предикате, которая выполняла единственную операцию. Поэтому, чтобы сравнивать элементы контейнера с числом 5, нам нужно было писать функцию `gt5()`, а чтобы сравнить с тройкой, приходилось писать другую функцию, например, `gt3()`.

Естественно, можно объявить объект-функцию явным образом:

```
Greater four(4);
copy_if(a, a+10, b, four);
```

Класс-функтор вполне может быть и шаблоном, например:

```
template <class T>
class Greater
{ T value;
public:
    Greater(const T &v): value(v) {}
    bool operator()(const T& d)
    { return (d > value); }
};
```

Класс `T` должен иметь конструктор без аргументов и обеспечивать реализацию операции `operator >`.

Тогда вызов `copy_if()` может выглядеть, например, так:

```
copy_if(a, a+10, b, Greater<int>(4));
```


ВНИМАНИЕ

Следите за тем, чтобы имена ваших функторов-шаблонов не совпадали с именами функторов из стандартной библиотеки шаблонов. В противном случае можно «наравиться» на странные ошибки вплоть до `fatal error` (фатальная ошибка) компилятора.

Адаптеры функторов

Реализация функтора `Greater` не является универсальной в данном случае — он «адаптирован» под нашу функцию-фильтр `copy_if()`. «Адаптация» состоит в том, что мы вообще-то бинарный предикат «больше» превратили в унарный, задав один из аргументов как аргумент операции `operator()`, а другой — в виде поля класса. Причем мы фиксируем значение этого аргумента при создании объекта-функтора. Таким образом, при вызове функтора в теле `copy_if()` ему передается только один аргумент — элемент контейнера. Второй инкапсулирован в функторе, и его значение зафиксировано при конструировании функтора.

Можно пойти по другому пути: написать универсальный бинарный функтор-предикат `Greater` и попытаться использовать его в качестве аргумента функции `copy_if()`. Возможная реализация представлена в листинге 12.12.

Листинг 12.12. Универсальный функтор «больше»

```
template<class Type>
struct Greater
{
    bool operator()(const Type& Left, const Type& Right) const
    {
        return (Left > Right);
    }
};
```

Однако мы не сможем непосредственно использовать такой функтор в функции `copy_if()` — аргументов у него два. Тогда зачем нам «прелести» универсальности? Вообще-то в нашем конкретном случае дальнейшее обобщение выглядит лишним. Однако торопиться с выводами не следует. Этот предикат может пригодиться для других обобщенных алгоритмов (например, для сортировки), так как способен работать с любым классом, в котором реализован конструктор по умолчанию и операция «больше». Например, его вполне можно использовать для сравнения строк, если в классе строк перегружена операция `operator >`.

Поэтому мы все-таки попробуем задействовать этот универсальный предикат. Нам нужно каким-то образом «избавиться» от одного из аргументов, оставив только второй. Предыдущая версия `Greater` подсказывает нам решение: нужно написать класс-адаптер, в котором один из аргументов предиката задается как аргумент конструктора. Таким образом, мы превратим бинарный предикат в унарный. Адаптеров должно быть два — разные для первого и второго аргументов. Кроме того, в качестве параметра, очевидно, должен передаваться и сам бинарный предикат. Реализация представлена в листинге 12.13.

Классы достаточно простые, но мы наблюдаем еще одну не совсем обычную деталь — поле-функтор, заполняемое конструктором. Соответствующий параметр передается конструктору по константной ссылке. Конечно, то же самое можно

сделать и с указателем на функцию, но в объектно-ориентированном программировании все же привычнее работать с объектами.

Листинг 12.13. Адаптеры-фиксаторы аргументов бинарного предиката

```
template<class Predicate, class T>
class firstarg
{ public:
    firstarg(const Predicate& P, const T& Left)
        : op(P), value(Left)
    { }
    bool operator()(const T& Right) const
    { return op(value, Right); } // реальный вид предиката
private:
    Predicate op; // предикат
    T value; // фиксированный аргумент
};

template<class Predicate, class T>
class secondarg
{ public:
    secondarg(const Predicate& P, const T& Right)
        : op(P), value(Right)
    { }
    bool operator()(const T& Left) const
    { return op(Left, value); } // реальный вид предиката
private:
    Predicate op;
    T value;
};
```

Подобный класс-адаптер называется *связывателем* (binder) [28]. В [31] используется термин «привязка». Однако мне больше нравится термин *фиксатор*, поскольку он точно отражает назначение класса. Использовать эти классы можно так:

```
int a[10] = { 1,2,3,4,5,6,7,8,9,0};
int b[10] = {0};
copy_if(a, a+10, b, secondarg<Greater<int>, int>(Greater<int>(), 4));
for(int i = 0; i < 10; ++i)
    cout << b[i] << ' ';
cout << endl;
```

Этот фрагмент программы выдаст на экран следующее:

```
5 6 7 8 9 0 0 0 0 0
```

Совершенно очевидно, что писать при вызове `copy_if()` такой длинный аргумент, да еще дважды указывать предикат — не слишком удобно. Поэтому для удобства и читабельности нужно написать две функции-оболочки наших классов-фиксаторов. Тогда параметры функции-шаблона будут выводиться компилятором, и запись аргумента существенно упростится (листинг 12.14).

Листинг 12.14. Функции-шаблоны для классов-фиксаторов

```
template<class Predicate, class Type>
inline
firstarg<Predicate, Type> bindleft(const Predicate& P, const Type& Left)
{
    Type value(Left);
    return (firstarg<Predicate, Type>(P, value));
}
template<class Predicate, class Type>
inline
secondarg<Predicate, Type> bindright(const Predicate& P, const Type& Right)
{
    Type value(Right);
    return (secondarg<Predicate, Type>(P, value));
}
```

Хочется обратить ваше внимание на то, что функции возвращают объект-функтор соответствующего типа. Может показаться, что ничего интересного в этом нет, пока мы не вспомним об указателях на функцию. Попробуйте вернуть из функции указатель на функцию, особенно указатель того же типа, что и сама функция. Как показано в [10, 21], это не так просто, и без «обмана» компилятора проделать такое трудно. А объекты-функторы избавляют нас от «головной боли» и при передаче аргумента, и при возврате результата.

Функции объявлены подставляемыми, и в данном случае компилятору ничего не мешает сделать их именно таковыми, так как сложные вычисления в функциях отсутствуют. Таким образом, функторы обеспечивают нам и повышение эффективности по сравнению с параметрами-указателями на функцию.

Использовать данные функции-шаблоны можно так:

```
int a[10] = { 1,2,3,4,5,6,7,8,9,0};
int b[10] = {0};
copy_if(a, a+10, b, bindright(Greater<int>(), 4));
for(int i = 0; i < 10; ++i)
    cout << b[i] << ' ';
cout << endl;
```

Результат получается тот же самый, а запись, как видите, существенно упростилась.

Можно, естественно, продолжать обобщение этого механизма. Ведь ничто не мешает нам написать классы-фиксаторы на произвольное количество аргументов¹. Обратите внимание на красоту решения! Мы имеем в функции `copy_if()` только унарный предикат, а задавать можем практически любой функтор.

Вернемся к нашей первоначальной задаче — обобщить функцию `copy_if()` для указателей на методы. Сначала решим более простую задачу — напомним класс-адаптер для указателя на функцию. Для этого нужно указатель на функцию сделать полем класса-адаптера. Сразу напишем и шаблон функции-оболочки. Возможная реализация представлена в листинге 12.15.

¹ Так и сделано в известной библиотеке Boost.

Листинг 12.15. Класс-адаптер для указателя на унарную функцию

```
template<class Result, class TypeP>
class pointertounaryfunction
{ public:
    explicit pointertounaryfunction(Result (*PF)(TypeP)): Pfun(PF) { }
    Result operator()(TypeP arg) const
    { return (Pfun(arg)); }
private:
    Result (*Pfun)(TypeP);           // указатель на функцию
};
// функция-оболочка
template<class Result, class TypeP>
inline
pointertounaryfunction <Result, TypeP>
ptrfun(Result (*PF)(TypeP))
{ return (pointertounaryfunction<Result, TypeP>(PF)); }
```

Возвращаемый результат тоже сделан параметром, поэтому с данным классом и шаблоном функции-оболочки правильно работает любая функция, принимающая один параметр и возвращающая результат любого типа. Например, мы можем умножить элементы контейнера на 2, используя приведенную ранее функцию `twice()`:

```
int a[10] = { 1,2,3,4,5,6,7,8,9,0};
int b[10] = {0};
copy_if(a, a+10, b, ptrfun(twice));
for(int i = 0; i < 10; ++i)    cout << b[i] << ' ';
cout << endl;
```

Результат, естественно, совпадает с тем, который получался при непосредственной передаче указателя на функцию. Тогда возникает вопрос: а зачем опять «городить огород» с функторами, если можно передавать просто указатель? Во-первых, работать с объектами удобней и эффективней, чем с указателями, во-вторых, в данном случае вызов стал самодокументированным — явно показано, что мы передаем указатель на функцию.

Точно так же мы можем написать класс-адаптер для бинарных функций и соответствующую функцию-оболочку (листинг 12.16).

Листинг 12.16. Класс-адаптер для указателя на бинарную функцию

```
template<class Result, class TypeP1, class TypeP2>
class pointertobinaryfunction
{ public:
    explicit                               // конструктор
    pointertobinaryfunction(Result (*PF)(TypeP1, TypeP2)): Pfun(PF) {}
    Result operator()(TypeP1 Left, TypeP2 Right) const
    { return (Pfun(Left, Right)); }
private:
    Result (*Pfun)(TypeP1, TypeP2);       // указатель на функцию
};
// функция-оболочка
template<class Result, class Arg1, class Arg2>
```

Листинг 12.16 (продолжение)

```
inline
pointertobinaryfunction<Result, Arg1, Arg2>
ptrfun(Result (*PF)(Arg1, Arg2))
{ return (pointertobinaryfunction<Result, Arg1, Arg2>(PF)); }
```

Используя классы-фиксаторы, мы вполне можем передать в `copy_if()` любую бинарную функцию, например приведенную выше функцию `greater()`:

```
copy_if(a, a+10, b, bindright(ptrfun(greater), 4));
```

Естественно, все работает, как и прежде.

Адаптер для указателя на метод

Теперь можно написать класс-адаптер для указателя на метод. Напишем простой класс, в котором реализован метод, сравнивающий свой аргумент со значением поля класса (листинг 12.17).

Листинг 12.17. Класс с методом сравнения

```
class Constant
{   int value;
public:
    Constant():value(0) {}
    Constant(int a): value(a) {}
    bool greater(int t)           // метод сравнения
    { return value > t; }
};
```

Метод `greater()` и будем вызывать косвенно через указатель на метод, который, очевидно, имеет следующий вид:

```
bool (Constant::*pointer)(int);
```

Напомню, что параметров у метода на самом деле два: помимо явно указанного параметра типа `int` есть еще `this` — указатель на «свой» объект. Поэтому, помимо класса-адаптера, нам потребуется еще и класс-фиксатор.

ПРИМЕЧАНИЕ

Интересно, что стандарт (см. п. п. 9.3.2 в [1]) не определяет, каким образом указатель `this` попадает в тело нестатического метода. Однако и в системе Visual C++ .NET 2003, и в системе Borland C++ Builder 6 этот указатель передается как первый аргумент метода. Поэтому фиксатор нам потребуется только один — для второго аргумента.

Операций выбора члена класса две, однако мы реализуем один класс-адаптер¹ для операции `.*`. Пойдем по пути некоторого обобщения наших классов-адаптеров. Сначала напомним классы-шаблоны свойств функций-предикатов (листинг 12.18).

¹ Вообще-то классов должно быть 4, так как для каждой из двух операций выбора члена класса можно выбрать либо константный, либо неконстантный метод.

Листинг 12.18. Классы-шаблоны свойств функций-предикатов

```
template<class Arg>
struct unarypredicate           // унарный предикат
{   typedef Arg      argument_type;   };
template<class Arg1, class Arg2>
struct binarypredicate          // бинарный предикат
{   typedef Arg1      first_argument_type;
    typedef Arg2      second_argument_type;
};
```

Естественно, можно было и тип результата сделать параметром шаблона, однако в данном случае это не очень существенно. Гораздо важнее то, что мы можем теперь наследовать от этих шаблонов! Например, приведенный в листинге 12.12 обобщенный функтор `Greater` можно написать как наследник от класса бинарного предиката (листинг 12.19).

Листинг 12.19. Наследование от класса свойств

```
template<class Type>
struct Greater: public binarypredicate<Type, Type>
{   bool operator()(const Type& Left, const Type& Right) const
    {   return (Left > Right);   }
};
```

Наследование позволяет упростить шаблон класса-фиксатора, оставив только один параметр шаблона — функцию-предикат (листинг 12.20). Тип аргументов предиката, который передавался как второй аргумент шаблона, мы «позаимствуем» у базового класса свойств. Естественно, упрощается и функция-оболочка.

Листинг 12.20. Класс-фиксатор — наследник от класса свойств

```
template<class Predicate>
class secondarg: public unarypredicate<typename Predicate::first_argument_type>
{   public:
    typedef unarypredicate<typename Predicate::first_argument_type> Base;
    typedef typename Base::argument_type      argument_type;
    // конструктор
    secondarg(const Predicate& P,
              const typename Predicate::second_argument_type& Right)
        : op(P), value(Right)   {   }
    bool operator()(argument_type Left)
    {   return (op(Left, value));   } // вызываемый предикат

    Predicate
    typename Predicate::second_argument_type value;
};
// функция-оболочка
template<class Predicate, class Type>
inline
secondarg<Predicate>
bindright(const Predicate& P, const Type& Right)
{   typename Predicate::second_argument_type value(Right);
    return (secondarg<Predicate>(P, value));
}
```

Обратите внимание на важнейшую особенность: класс-фиксатор наследует от класса унарного предиката, а в качестве параметра получает бинарный предикат. Теперь напишем сам класс-адаптер для указателя на метод и функцию-оболочку. Указатель на метод, естественно, является полем этого класса, а сам класс наследует от класса бинарного предиката (листинг 12.21).

Листинг 12.21. Класс-адаптер для указателя на метод — наследник класса свойств

```
template <typename Class, typename T>
class ReferenceToMethod: public binarypredicate<Class, T>
{
public:
    explicit ReferenceToMethod(bool (Class::*pm)(T v))
        : pmethod(pm) {}
    bool operator()(Class Obj, T v)
    { return ((Obj.*pmethod)(v)); }           // вызов метода
private:
    bool (Class::*pmethod)(T v);             // указатель на метод
};
template<class Class, class T>
inline
ReferenceToMethod<Class, T> memfun(bool (Class::*Pm)(T v))
{ return (ReferenceToMethod<Class, T>(Pm)); }
```

Конструктор получает в качестве параметра указатель на метод и заполняет поле класса. Операция вызова функции `operator()` принимает объект и аргумент метода, вызывая метод через инициализированный конструктором указатель на метод. Функция-оболочка автоматизирует вывод параметров шаблона.

Использовать эти классы и функции-оболочки можно так:

```
int a[10] = { 1,2,3,4,5,6,7,8,9,0};
int b[10] = {0};
copy_if(a, a+10, b, bindright(memfun(&Constant::greater), 5));
for(int i = 0; i < 10; ++i)    cout << b[i] << ' ';
cout << endl;
```

Мощь аппарата функторов не может не вызывать изумления! Посмотрите, на месте единственного параметра-функтора фактически заданы 4 аргумента:

1. Указатель на метод `Constant::greater`.
2. Целая константа 5.
3. Функция `memfun()`.
4. Функция `bindright()`.

Попробуйте-ка реализовать аналогичную функциональность, используя указатели на функции!

Обратите внимание на то, что класс-адаптер `ReferenceToMethod` наследует от бинарного предиката, хотя его аргументы в теле класса никак не используются. В данном случае без наследования обойтись никак нельзя, поскольку «последней инстанцией» является класс-фиксатор, который, будучи наследником, задействует аргументы предиката. В классе-адаптере можно было бы написать несколько

операторов `typedef` по аналогии с классом-фиксатором, но я не стал этого делать, чтобы не засорять класс-адаптер техническими деталями.

Таким образом, итераторы и функторы позволили нам написать максимально обобщенную функцию-фильтр `copy_if()`, которая способна обрабатывать любой последовательный контейнер, обладающий интерфейсом итераторов. Механизм функторов обеспечил нам уникальные возможности в плане передачи практически любой обрабатываемой функции в качестве параметра. Функции, подобные `copy_if()`, получили название *обобщенные алгоритмы*, и большое их количество реализовано в библиотеке STL.

Можно и дальше продолжать обобщение наших классов-адаптеров. Например, в [28] приводится значительно более обобщенный вариант класса-фиксатора `Binder`. Примеры еще более обобщенных классов-шаблонов для функторов можно найти в библиотеке Boost. Вместо нескольких классов-адаптеров для различных типов указателей на методы реализован единый обобщающий механизм `mem_fn`. Понятно, что без классов-фиксаторов от этого механизма мало проку, поэтому в Boost реализован и значительно более мощный механизм `bind`. Это — целая библиотека взаимосвязанных друг с другом шаблонов¹.

Помимо этого в библиотеке Boost можно найти и обобщенный шаблон-оболочку `function`², обеспечивающий единообразную работу с функциями, методами и функторами.

Резюме

Наряду с шаблонами классов в C++ реализованы шаблоны функций, которые являются естественным развитием механизма перегрузки функций. Однако по сравнению с шаблонами классов шаблоны функций имеют довольно много ограничений. Например, нельзя задавать параметры шаблона по умолчанию. Зато работает автоматический вывод параметров (хотя тоже с некоторыми ограничениями). Шаблон функции разрешается специализировать, но частичная специализация не разрешается. Зато шаблон функции можно перегрузить как другим шаблоном, так и функцией. Ограничения, естественно, можно обойти, если «обернуть» функцию в класс-шаблон.

Шаблоны функций позволяют реализовать обобщенные алгоритмы, которые могут работать с контейнером любого вида. В качестве параметров в обобщенном алгоритме выступают итераторы. Одним из аргументов обобщенного алгоритма часто является некоторая операция, которую необходимо выполнить с каждым элементом контейнера. Обычно в качестве параметра-операции задается указатель на функцию. Однако в обобщенном алгоритме аргумент-операция должен задаваться универсальным образом, поэтому параметр не может быть задан в виде указателя, так как указатели на методы отличаются от указателей на функцию.

¹ Первоначальные варианты `mem_fn` и `bind` были написаны в 2001 году Дэвидом Абрахамсом (David Abrahams); дальнейшие обобщения и усовершенствования выполнил Петр Димов (Peter Dimov).

² Разработал Дуглас Грегор (Douglas Gregor).

Аргументы-операции в обобщенных алгоритмах задаются обычно как объекты-функторы. Функтор — это класс, в котором перегружена операция вызова функции `operator()`. Использование функторов позволяет сделать алгоритм по-настоящему универсальным. Классы-оболочки для указателей на функции и указателей на методы, которые называют адаптерами, повышают степень универсальности обобщенного алгоритма. Один из важных адаптеров — фиксатор, который позволяет превратить бинарный функтор в унарный.

Контрольные вопросы

1. Перечислите отличия шаблона функции от шаблона класса.
2. Можно ли перегружать функцию-шаблон?
3. Какие параметры функции-шаблона выводятся автоматически?
4. Разрешается ли параметрам шаблона функции присваивать параметры по умолчанию?
5. Каким образом «обойти» ограничение частичной специализации для шаблонов функций?
6. Перечислите приемы программирования, с помощью которых шаблон функции становится более универсальным.
7. Дайте определение функтора.
8. Можно ли операцию вызова функции реализовать как виртуальную функцию? А как статическую?
9. Дайте определение функтора-предиката.
10. Чем отличается функтор с состоянием от обычного функтора?
11. Можно ли указатель на функцию присвоить указателю на метод? А наоборот?
12. Отличается ли указатель на обычный метод от указателя на статический метод?
13. Может ли класс-функтор участвовать в иерархии наследования?
14. Объясните назначение адаптера-фиксатора.
15. Можно ли перегружать операцию вызова функции?

Упражнения

1. Написать шаблон функции, принимающей числовой массив в качестве аргумента и возвращающей динамический массив типа `TArray` (см. листинг 11.9) как результат. Размер обычного массива передается в качестве аргумента шаблона. Исходный массив заполнить случайными числами в диапазоне от -40 до $+60$. Отнять от каждого числа исходного массива наибольшее число. Добавить в массив-результат сумму и среднее арифметическое по абсолютной величине.
2. Написать шаблон функции, которая получает в качестве параметров два указателя на элементы обычного массива, а возвращает массив типа `TArray` (см. листинг 11.9).

3. Написать шаблон функции реверсирования последовательности элементов, задаваемых двумя итераторами. Использовать шаблон стека (см. листинг 11.1).
4. Написать шаблон функции пузырьковой сортировки, параметрами которой являются два итератора. Отсортировать с ее помощью массив целых чисел типа `int` и массив дробных чисел типа `double`.
5. Добавить к параметрам функции сортировки из предыдущего упражнения третий параметр — функцию для сравнения элементов, на которые указывают итераторы. Функция сортировки должна вызывать функцию сравнения для сравнения элементов, на которые указывают итераторы. Функция сравнения возвращает значение типа `bool`.
6. Выполнить предыдущее упражнение, создав класс-оболочку (см. листинг 12.8) для функции сравнения и реализовав ее в классе как операцию `operator<`.
7. Разработать обобщенный алгоритм `count_if()` для вычисления количества элементов контейнера, удовлетворяющих заданному условию, по образцу алгоритма `copy_if()` (см. листинг 12.7). Параметрами должны являться итераторы и функторы. Для демонстрации работы алгоритма использовать шаблоны динамических массивов и шаблоны списков. Реализовать два разных функтора.
8. Написать шаблоны функторов для всех операций сравнения по образцу шаблона функтора `Greater`.
9. Реализовать алгоритм `binoperate()`, выполняющий заданную функтором операцию с двумя интервалами и записывающий результат в выходную последовательность.
10. Использовать шаблоны функтора `Greater<>` и `less<>` из упражнения 8 для сортировки элементов последовательности в функции сортировки из упражнения 5.
11. Реализовать адаптер указателя на метод для операции `->*`.

Глава 13

Программы и модули

Большую программу невозможно написать, не разбив ее на части. До сих пор мы изучали механизмы логической декомпозиции программы: классы и функции. Однако большие программы требуется разбивать на части еще и физически. Обычно отдельная часть большой программы называется *модулем*. Разбиение программы на части выгодно по многим причинам: отдельные части крупного проекта могут одновременно разрабатываться разными программистами; с небольшим модулем быстрее и удобнее работать; разбиение способствует инкапсуляции, что облегчает обнаружение и исправление ошибок.

Разделить программу на части — это только половина работы. Отдельные модули сами по себе функционировать не будут — программу требуется снова собрать в единое целое. Обычно сборка осуществляется одним из двух способов: во-первых, можно объединить исходные тексты, во-вторых, программу собрать можно и из объектных модулей. *Объектный модуль* — это результат трансляции одного модуля. Такой способ называется *раздельной* трансляцией.

Основным достоинством первого подхода является простота, а главный недостаток — необходимость транслировать большую программу. Этот процесс может занимать очень много времени.

Во втором варианте объединяются объектные модули. Собственно, этот способ и был придуман для того, чтобы избежать трансляции всей программы целиком. Процесс сборки программы из объектных модулей называется *компоновкой*, и выполняет эту работу специальная программа, которая обязательно входит в состав любой системы программирования. Эта программа называется компоновщиком, но программисты часто называют ее линкером (linker).

При компоновке в программу собираются не только разработанные нами модули, но и *стандартные* модули. Стандартные модули, естественно, не транслируются вместе с нашей программой — они поставляются вместе с системой в виде объектных модулей и объектных библиотек. Если вы напишете в программе вызов

некоторой стандартной функции, например `sqrt(x)`, то компоновщик разыщет среди стандартных модулей тот, в котором записана эта функция, и присоединит ее к программе.

Обычно в реальном программировании используется сочетание обоих подходов: в единую программу komponуются объектные модули, но отдельный объектный модуль может собираться из нескольких модулей с исходным текстом. В C++ объединение исходных текстов делается с помощью препроцессора.

Таким образом, мы можем разбить большую программу на файлы, оттранслировать их по отдельности, а потом собрать в единое целое. Однако в C++ отсутствуют конструкции для обозначения модуля. Вместо этого в стандарте определено понятие единицы трансляции (см. п. п. 2/1 в [1]). *Единица трансляции* — это отдельный файл с исходным текстом на C++, который получается после обработки препроцессором. Поэтому в дальнейшем мы будем использовать термины «файл» и «модуль» как синонимы. В C++ отсутствуют также конструкции для сборки объектных модулей в исполняемую программу — это делается средствами интегрированной среды. Обычно в интегрированной среде создается *проект*, в составе которого указываются все исходные и объектные модули.

ПРИМЕЧАНИЕ

Средства определения и сборки модулей включены во многие языки. В клонах Modula и Oberon модуль является основной конструкцией. В языке Turbo Pascal модуль обозначается зарезервированным словом `unit`, а использование модуля в программе можно указать директивой `uses`. В Java модуль (пакет) обозначается зарезервированным словом `package`, а подключение пакета задается оператором `import`.

Раздельная трансляция требует согласования объявлений и определений в разных единицах трансляции. Если что-то не согласовано, то мы получим ошибку либо при компиляции, либо при компоновке. Во всех единицах трансляции должны быть согласованы объявления классов, функций, переменных и констант, перечислений, шаблонов и *пространств имен*.

Сборка исходных текстов

При реализации примеров наследования (см. главы 8 и 9) мы объединяли в одном модуле и базовый, и производный классы. В качестве примера можно привести реализацию базового класса `TStack` (см. листинг 6.9) и его наследников (см. листинги 8.9–8.11). Однако стек нам может понадобиться и в других программах, поэтому естественно выделить этот класс в отдельную единицу трансляции. Создадим файл `TStack.cpp` и перенесем в него определение класса `TStack`. Содержимое этого файла представлено в листинге 13.1.

Как видите, определение класса не меняется, только переносится в отдельный файл. Другой модуль с именем `main.cpp` (листинг 13.2) содержит программу-клиент, использующую наш стек.

Листинг 13.1. Содержимое файла TStack.cpp

```

class TStack
{ struct Elem
  { void *data;
    Elem *next;
    Elem (void *d, Elem *p)
      :data(d), next(p) { }
  };
  Elem * Head;
  TStack(const TStack &);           // закрыли копирование
  TStack& operator=(const TStack &); // закрыли присваивание
public:
  TStack(): Head(0) {}
  void push(void *d) { Head = new Elem(d, Head); }
  void *top()const { return Head? Head->data:0; }
  void *pop()
  { if (Head == 0) return 0;
    void *top = Head->data; Elem *oldHead = Head;
    Head = Head->next; delete oldHead;
    return top;
  }
  bool empty() { return Head==0; }
};

```

ВНИМАНИЕ

При создании проекта средствами интегрированной среды Visual Studio.NET 2003 файл main.cpp должен быть единственным в проекте.

Листинг 13.2. Содержимое файла main.cpp

```

#include "TStack.cpp"           // наш файл
#include <iostream>              // системный файл
using namespace std;           // стандартное пространство имен
int main()
{ TStack t;
  t.push(new double(1));        // помещаем в стек числа
  t.push(new double(2));
  t.push(new double(3));
  while (!t.empty())            // пока стек не пустой
  { cout << *(double *)t.top() << endl; // выводим число с вершины
    double *p = (double *)t.pop();    // удаляем элемент из стека
    delete p;                        // возвращаем память
  }
}

```

Объединение модулей в единую программу осуществляется препроцессором по директиве включения:

```
#include имя_файла
```

Эту директиву мы уже неоднократно использовали, не вдаваясь в подробности. Препроцессор, обнаружив директиву, «подставляет» на ее место указанный файл. Обратите внимание на то, что две директивы #include написаны по-разному. Первая форма директивы:

```
#include "файл"
```

Эта форма директивы задает модуль, который должен находиться в текущем каталоге — в том же, где и транслируемый файл `main.cpp`. Вообще-то говоря, можно поместить подключаемый файл в любой каталог, но тогда нужно указывать полное имя, например:

```
"c:\\Piter\\part1\\ch01\\TStack.cpp"  
"c:/Piter/part1/ch01/TStack.cpp"
```

Оба варианта эквивалентны.

Вторая форма директивы:

```
#include <файл>
```

Эта форма применяется для включения стандартных библиотечных файлов. Указанный таким образом файл должен находиться в стандартном каталоге интегрированной среды `include`. Мы могли бы и файл с классом `TStack` поместить туда же — тогда можно было бы написать директиву включения как системную:

```
#include <TStack.cpp> // наш файл
```

Однако лучше не «засорять» системные каталоги собственными файлами.

Имя файла нужно указывать абсолютно точно, иначе файл не будет найден. Пробелы внутри угловых скобок или внутри кавычек — значимы, поэтому без необходимости дополнительные пробелы писать нельзя.

После директив `#include` написана директива `using`, которую мы неоднократно прописывали в начале программы, не очень задумываясь, зачем это необходимо. С помощью этой директивы мы подключаем *стандартное пространство имен* (`namespace`), в котором прописаны все стандартные классы, переменные, константы, функции, макросы.

«Стражи» включения

Продолжая развитие нашего проекта со стеком, создадим третий модуль, в котором объект класса `TStack` будет передаваться в функцию как параметр. Назовем этот файл `function.cpp` (листинг 13.3).

Листинг 13.3. Содержимое файла `function.cpp`

```
void function (TStack &t)  
{ while (!t.empty()) // пока стек не пустой  
{ cout << *(double *)t.top() << endl; // выводим число с вершины  
  double *p = (double *)t.pop(); // удаляем элемент из стека  
  delete p;  
}  
}
```

Проблем пока нет, но они появляются при подключении файла к нашей главной программе. Выясняется, что мы должны учитывать порядок следования директив `#include`, чтобы все работало. Правильный порядок такой:

```
#include <iostream>  
using namespace std;  
#include "TStack.cpp"  
#include "function.cpp"
```

Стандартный файл `iostream` должен быть указан перед файлом `function.cpp`, так как в функции `f()` выполняется вывод в стандартный поток. Ранее должен быть включен и файл со стеком, так как функция `f()` его получает в качестве параметра. Порядок следования файлов `TStack.cpp` и `iostream` в данном случае может быть произвольным, но только потому, что в нашем стеке никак не используется стандартный ввод-вывод.

Налицо зависимость файлов друг от друга во время компиляции — ситуация крайне неприятная: мало того, что каждый раз транслируется объединенный текст, так еще мы должны «вручную» отслеживать связи между различными модулями. Это достижимо только для очень небольших программ, состоящих не более чем из пары десятков модулей, а дальше начинаются проблемы.

На ум приходит естественное решение — добавить в модуль `function.cpp` три первые строки из модуля `main.cpp`:

```
#include <iostream>
using namespace std;
#include "TStack.cpp"
```

Однако при трансляции тут же выясняется, что класс `TStack` определен дважды¹. По стандарту в программе разрешается иметь несколько определений класса, но в разных единицах трансляции (см. п. п. 3.2/5 в [1]). А у нас получается двойное определение в одной единице — в модуле `main.cpp`. Действительно, после постановки всех наших файлов (про системные файлы пока умолчим) объединенный модуль `main.cpp` содержит следующие директивы `#include`:

- `#include <iostream>` — включает системный файл `iostream`;
- `#include "TStack.cpp"` в модуле `main.cpp` — включает класс `TStack`;
- `#include "function.cpp"` в модуле `main.cpp` — включает файл `function.cpp`;
- `#include "TStack.cpp"` в модуле `function.cpp` — включает класс `TStack`.

Таким образом, класс `TStack` оказывается определенным дважды. Справиться с этой проблемой нам опять поможет препроцессор: мы должны определить во включаемом файле «стража» включения. Для этого используется директива `#define` и директивы условной трансляции препроцессора. Обычно это делается так:

```
// myfile.h
#ifndef MYFILE // страж определен?
#define MYFILE // если нет - определяем
// содержимое файла
#endif // конец #ifndef
```

Первая строка часто пишется немного по-другому:

```
#if !defined( MYFILE ) // страж определен?
```

Однако сути дела это не меняет.

Когда препроцессор обрабатывает первую директиву, имя `MYFILE` (страж) не определено:

```
#include "myfile.h"
```

¹ Обратите внимание, что с системным файлом ошибок не возникает!

Поэтому содержимое файла включается в обработку. Если эта директива встречается еще раз, то страж уже определен и содержимое файла пропускается.

Именно так организованы стандартные файлы, и поэтому не было проблем со стандартным файлом `iostream`. Например, начало стандартного файла `math.h` в интегрированной среде Borland C++ Builder 6 выглядит так:

```
/* math.h
   Definitions for the math floating point package.
*/
/*
   *      C/C++ Run Time Library - Version 11.0
   *      Copyright (c) 1987, 2002 by Borland Software Corporation
   *      All Rights Reserved.
   */
/* $Revision: 9.17.2.6 $ */
#ifndef __MATH_H           // проверка стража
#define __MATH_H          // определение стража
```

Как видите, определен страж с именем `__MATH_H`. Практически не отличается от этого файла соответствующий файл системы Visual C++.NET 2003 (пропущены некоторые строки, не относящиеся к делу):

```
/**
 *math.h - definitions and declarations for math library
 *
 *      Copyright (c) Microsoft Corporation. All rights reserved.
 *
 *Purpose:
 *      This file contains constant definitions and external subroutine
 *      declarations for the math subroutine library.
 *      [ANSI/System V]
 *
 *      [Public]
 *
 ****/
#ifndef _INC_MATH
#define _INC_MATH
```

В этой системе определен страж с именем `_INC_MATH`.

По негласному соглашению имена, определяемые в директиве `#define`, пишутся прописными буквами — это позволяет по одному виду имени определить, что оно «принадлежит» препроцессору и «работает» только на стадии компиляции. Наш файл `TStack.cpp` должен быть таким, как показано в листинге 13.4.

Листинг 13.4. Файл `TStack.cpp` со стражем

```
#ifndef _STACK           // страж определен?
#define _STACK           // определение стража
class TStack
{
    // определение класса
};
#endif /* _STACK */      // конец ifndef
```

Теперь все работает правильно.

Системные включаемые файлы

При изучении системных файлов вы, наверное, обратили внимание, что в операторе `#include` одни системные файлы пишутся с расширением `h`, а другие — без расширения. В частности, без расширения всегда пишется системный файл `iostream`. У него именно такое имя (без расширения) — так рекомендует стандарт (см. п. п. 17.4.1.2 в [1]). До принятия стандарта все системные файлы имели расширение `h`, поскольку так было принято в С. Для совместимости стандарт разрешает писать и в «старом добром» стиле С с расширением `h`, однако при таком написании нас могут поджидать некоторые неприятные сюрпризы. Например, обратите внимание на следующие операторы:

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

В первом случае обычно в интегрированной среде подключается старая версия системы ввода-вывода, а во втором — новая. Аналогично:

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

Первый оператор означает подключение набора функций для работы с символьными массивами (см. приложение), а второй подключает стандартный класс С++ для работы со строками. Поэтому, чтобы предупредить возможные конфликты и недоразумения, в стандарте С++ предписано включать файлы библиотек С начинать с символа «с». Таким образом, в С++ заголовок `string.h` переименован в `cstring`, соответственно, `ctype.h` надо писать как `cctype`, а `math.h` — как `cmath`, и т. д. Новые заголовки (без расширения `h`) определяют *стандартное пространство имен* `std`, а старые — нет.

В составе С++ имеется 32 системных файла-заголовка (см. п. п. 17.4.1.2 в [1]):

<code><algorithm></code>	<code><iomanip></code>	<code><list></code>	<code><queue></code>	<code><streambuf></code>
<code><bitset></code>	<code><ios></code>	<code><locale></code>	<code><set></code>	<code><string></code>
<code><complex></code>	<code><iosfwd></code>	<code><map></code>	<code><sstream></code>	<code><typeinfo></code>
<code><deque></code>	<code><iostream></code>	<code><memory></code>	<code><stack></code>	<code><utility></code>
<code><exception></code>	<code><istream></code>	<code><new></code>	<code><stdexcept></code>	<code><valarray></code>
<code><fstream></code>	<code><iterator></code>	<code><numeric></code>	<code><strstream></code>	<code><vector></code>
<code><functional></code>	<code><limits></code>	<code><ostream></code>		

В дополнение к этим заголовкам имеются 18 стандартных файлов-заголовков, унаследованных от С:

<code><cassert></code>	<code><ciso646></code>	<code><csetjmp></code>	<code><cstdio></code>	<code><ctime></code>
<code><cctype></code>	<code><climits></code>	<code><csignal></code>	<code><cstdlib></code>	<code><cwchar></code>
<code><cerrno></code>	<code><locale></code>	<code><cstdarg></code>	<code><cstring></code>	<code><cwctype></code>
<code><cfloat></code>	<code><cmath></code>	<code><cstddef></code>		

По негласному практическому правилу файлы-заголовки обычно содержат определения и (или) объявления констант, типов, классов, шаблонов, а также объявления функций, перечисления, именованные пространства имен, макросы, директивы условной компиляции, комментарии. Изучение стандартных заголовочных файлов может дать много информации об этом.

Каждый заголовочный файл в C++ соответствует определенной специализированной части системных средств. Например, заголовок `cmath` «отвечает» за набор математических функций и констант, подключение заголовка `cctype` предоставляет нам набор функций для распознавания и изменения регистра символов, и т. д. Поэтому будем говорить, что с помощью оператора `#include` мы подключаем соответствующую стандартную библиотеку.

Отделение интерфейса от реализации

Решение, представленное в предыдущем разделе, хоть и правильное, но имеет уже упомянутый недостаток: необходимо транслировать программу целиком. Между тем не хотелось бы компилировать каждый раз заново уже проверенные и отлаженные модули. Если модуль не изменяется, то можно оттранслировать его один раз, а потом просто компоновать с остальными — именно так поступают с библиотечными функциями.

Для этого разделим наш модуль с классом `TStack` на две части: интерфейс и реализацию. Интерфейс класса является его определением, хотя для методов указаны только прототипы. Но по интерфейсу компилятор может вычислить размер класса, поэтому интерфейса достаточно, чтобы объявлять объекты этого класса. Файл с интерфейсом назовем `TStack.h`, а файл с реализацией пусть, как и прежде, имеет имя `TStack.cpp`. Файл `TStack.h` называется заголовком (header), и именно он подключается к другим модулям с помощью препроцессора. Стража нужно прописать в нем (листинг 13.5).

Листинг 13.5. Файл `TStack.h` — интерфейс класса `TStack`

```
#ifndef _STACK
#define _STACK
class TStack
{ public:
    TStack();
    void push(void *d);
    void *top() const;
    void *pop();
    bool empty() const;
private:
    struct Elem;
    Elem * Head;
    TStack(const TStack &);           // закрыли копирование
    TStack& operator=(const TStack &); // закрыли присваивание
};
#endif /* _STACK */
```

Файл реализации просто включается в проект интегрированной среды¹. Естественно, интерфейс класса обязан присутствовать и в этом файле (листинг 13.6).

¹ В проекте два файла: файл реализации `TStack.cpp` и файл `main.cpp` с программой-клиентом.

Листинг 13.6. Файл TStack.cpp — реализация класса TStack

```

#include "TStack.h"                // подключение интерфейса
struct TStack::Elem
{ void *data;
  Elem *next;
  Elem (void *d, Elem *p): data(d), next(p)
  { }
};
TStack::TStack(): Head(0), count(0) {}
void TStack::push(void *d) { Head = new Elem(d, Head); }
void * TStack::top() const { return Head? Head->data:0; }
void * TStack::pop()
{ if (Head == 0) return 0;
  void *top = Head->data;
  Elem *oldHead = Head; Head = Head->next; delete oldHead;
  return top;
}
bool TStack::empty() const
{ return Head==0; }

```

В программе-клиенте тоже подключается только интерфейс (листинг 13.7).

Листинг 13.7. Файл main.cpp

```

#include <iostream>
using namespace std;
#include "TStack.h"                // подключаем файл-заголовок
int main()
{ TStack t;
  t.push(new double(1));           // помещаем в стек числа
  t.push(new double(2));
  t.push(new double(3));
  while (!t.empty())               // пока стек не пустой
  { cout << *(double *)t.top() << endl; // выводим число с вершины
    double *p = (double *)t.pop();    // удаляем элемент из стека
    delete p;                         // возвращаем память
  }
  return 0;
}

```

Обратите внимание, что файл реализации явным образом в тексте программы-клиента никак не упоминается. Таким образом, мы ослабили зависимость между файлами и повысили степень инкапсуляции. Файл реализации компилируется, и объектный модуль подключается компоновщиком в исполняемую программу. Еще одна составляющая интегрированной среды — *мейкер* (maker) — следит за изменениями исходных модулей. До тех пор пока мы не внесем изменения в реализацию стека, в проекте будет использоваться оттранслированный объектный модуль.

Разделение модуля TStack.cpp на интерфейс и реализацию никак не мешает нам наследовать от класса TStack. Вспомним реализацию стека без указателей StackDouble (см. листинг 8.11) и создадим файл StackDouble.cpp (листинг 13.8).

Листинг 13.8. Файл StackDouble.cpp — наследование от класса TStack

```
#include "TStack.h" // подключение определения
class StackDouble: public TStack
{
public:
    void push(const double& d)
    { double *p = new double(d); TStack::push(p); }
    double top() const
    { double *p = (double *)TStack::top();
      double t = *p;
      return t;
    }
    double pop()
    { double *p = (double *)TStack::pop();
      double t = *p; delete p;
      return t;
    }
};
```

Для наследования достаточно подключить файл интерфейса, чтобы было видимо имя и определение класса TStack. Обратите внимание, что в данном случае стража мы не определяем, поскольку класс StackDouble будет подключаться только к одной единице трансляции — модулю main.cpp. Поэтому повторов определения не будет. Однако в общем случае стража нужно задавать всегда — чтобы в дальнейшем при повторном использовании не возникало неожиданностей. Программа-клиент, применяющая стек StackDouble, представлена в листинге 13.9.

Листинг 13.9. Файл main.cpp — программа-клиент

```
#include <iostream>
using namespace std;
#include "StackDouble.cpp" // подключаем новый стек
int main()
{ StackDouble t;
  t.push(1); // помещаем в стек числа
  t.push(2);
  t.push(3);
  while (!t.empty()) // пока стек не пустой
  { cout << t.top() << ' '; // выводим число с вершины
    cout << t.pop() << endl; // удаляем элемент из стека
  }
  return 0;
}
```

По-прежнему в проекте два файла: файл реализации TStack.cpp и файл main.cpp с программой-клиентом. Новый модуль StackDouble.cpp подключается к программе-клиенту с помощью препроцессора. Мы могли бы поступить с классом StackDouble точно так же, как и с классом TStack: разделить его на интерфейс и реализацию. Тогда файл реализации надо включить в проект, а файл с интерфейсом подключить к программе-клиенту. Однако можно включить в проект все четыре файла: TStack.h, TStack.cpp, StackDouble.cpp и main.cpp. Тогда за изменениями всех файлов будет следить мейкер интегрированной среды, и компилироваться будут только те файлы, которые изменились с момента последней сборки.

Шаблоны и модули

Обычные классы мы разделяли на интерфейс и реализацию, помещая интерфейс в модуль с расширением `h`. Этот модуль потом и подключается к другим файлам проекта оператором `#include`, а реализация класса просто включается в проект. Попробуем аналогичным образом поступить и с классами-шаблонами. Разделим наш шаблон стека на две части: определение и реализацию (листинг 13.10).

Листинг 13.10. «Определение» шаблона стека

```
//-----файл TSTACK.h
#ifndef TSTACK
#define TSTACK
#include <exception>
template <class T>
class TStack
{ struct Elem
  { T data;
    Elem *next;
    Elem (const T& d, Elem *p)
      : data(d), next(p) { }
  };
  Elem * Head;
  int count;
  TStack(const TStack &); // закрыли копирование
  TStack& operator=(const TStack &); // закрыли присваивание
public:
  class Error: public std::exception { };
  TStack();
  void push(const T& d);
  T top() const;
  T pop();
  bool empty() const;
  int count() const;
};
#endif
//-----файл TSTACKDEF.h
#include "TStack.h" // "подключение" определения
template <class T>
TStack<T>::TStack(): Head(0), count(0) {}
template <class T>
void TStack<T>::push(const T& d)
{ Head = new Elem(d, Head); ++count; }
template <class T>
T TStack<T>::top() const
{ if(!empty()) return Head->data;
  else throw Error();
}
template <class T>
T TStack<T>::pop()
{ if (empty()) throw Error();
  T top = Head->data;
  Elem *oldHead = Head; Head = Head->next;
```

```

        delete oldHead;
        --count;
        return top;
    }
    template <class T>
    bool TStack<T>::empty() const { return Head==0; }
    template <class T>
    int TStack<T>::count() const { return count; }

```

Как видите, чисто формально этот код ничем не отличается от кода обычного класса TStack (см. листинги 13.5 и 13.6). Третьим модулем в проект включается функция main(), в которой мы наш шаблон стека попытаемся использовать (листинг 13.11).

Листинг 13.11. «Использование» шаблона стека

```

#include "TStack.h"                // подключение "определения"
#include <iostream>
#include <string>
using namespace std;
int main()
{ TStack<double> t;
  t.push(1);                       // помещаем в стек числа
  t.push(2);
  t.push(3);
  cout << t.count() << endl;
  while (!t.empty())               // пока стек не пустой
  { cout << t.top() << endl;        // выводим число с вершины
    double p = t.pop();             // удаляем элемент из стека
  }
  cout << t.count() << endl;
  TStack<string> S;
  S.push("one");                   // помещаем в стек числа
  S.push("two");
  S.push("three");
  cout << S.count() << endl;
  while (!S.empty())               // пока стек не пустой
  { string p = S.pop();             // удаляем элемент из стека
    cout << p << endl;             // выводим строку
  }
  cout << S.count() << endl;
  try { string p = S.pop(); }        // стек пустой - генерируется исключение
  catch(const exception &e)
  { cout << e.what() << endl; }
  return 0;
}

```

Транслируется все нормально — компилятор ошибок не обнаруживает (система Visual C++.NET 2003). Однако при компоновке выдается 13 (!) сообщений! Причина заключается, конечно, в отсутствии инстанцирования шаблона¹.

Позвольте, но вот же в функции main()!... Однако у нас три отдельных модуля в проекте. Поэтому когда компилятор обрабатывает определение и реализацию

¹ Точнее, в отсутствии инстанцирования методов класса-шаблона.

шаблона-стека, у него нет указаний, что шаблон должен быть инстанцирован для каких-то конкретных аргументов. Когда же компилятор «видит» определение объекта-стека, наподобие одного из следующих, в его поле зрения нет реализации:

```
TStack<double> t;  
TStack<string> S;
```

Поэтому компилятор не может выполнить инстанцирование реализации, но предполагает, что где-то в других частях это сделано, и оставляет «пустую» ссылку компоновщику. Компоновщик, естественно, ничего не находит, так как инстанцирования-то компилятор не выполнил!

Аналогичная картина — с шаблонами функций. Попробуем шаблон функции `Summa()` (см. листинг 12.1) поместить в отдельный файл `Summa.h`, а в функции `main()` просто зададим прототип-объявление (листинг 13.12).

Листинг 13.12. «Использование» шаблона функции

```
#include <iostream>  
#include <string>  
using namespace std;  
template <typename T>                // прототип шаблона функции  
T Summa(T const *begin, T const *end);  
int main()  
{  
    int a[10]= {1,2,3,4,5,6,7,8,9,10};  
    cout << Summa(a, a+10) << endl;  
    double b[10]= {1.1,2,3,4,5,6,7,8,9,10.1};  
    cout << Summa(b, b+10) << endl;  
    string d[4] = {"1+", "2+", "3+", "4+"};  
    cout << Summa(d, d+4) << endl;  
    short s[2]= { 10000,10000 };  
    cout << Summa(s, s+2) << endl;  
    int x; cin >> x;  
    return 0;  
}
```

Транслируется все опять без ошибок, однако компоновщик снова не находит определения функции и не может скомпоновать программу.

Модель включения

Таким образом, шаблоны нельзя «просто делить» на части и транслировать раздельно. Шаблон представляет собой только заготовку для построения кода: пока шаблон не инстанцирован, объектного кода из него не получится, и компоновщику нечего будет собирать. Модули, использующие шаблон, с помощью директивы `#include` должны подключить файл с полным определением шаблона. Это касается и шаблонов классов, и шаблонов функций. Такой способ организации кода с шаблонами называется «моделью включения» [28]. Его главный недостаток нам уже хорошо известен — необходимость трансляции сразу всей программы.

Явное инстанцирование

Стандарт C++ предлагает еще один вариант организации кодов с шаблонами — модель *явного инстанцирования* (см. п. п. 14.7.2 в [1]). При такой организации шаблон инстанцируется *вручную* с помощью *директивы* явного инстанцирования. Рассмотрим наш пример с шаблоном функции `Summa()`. Создадим в проекте три модуля:

1. Модуль с шаблоном функции (см. листинг 12.1).
2. Модуль с функцией `main()`, в котором задан прототип шаблона (см. листинг 13.12).
3. Модуль с набором директив явного инстанцирования `SummaInst.h` (листинг 13.13).

Листинг 13.13. Файл с директивами явного инстанцирования

```
//-----файл-SummaInst.h
#include <string>
using namespace std;
#include "Summa.h"           // подключение шаблона
// набор директив явного инстанцирования
template double Summa<double>(double const *begin, double const *end);
template int Summa<int>(int const *begin, int const *end);
template short Summa<short>(short const *begin, short const *end);
template string Summa<string>(string const *begin, string const *end);
```

Директива явного инстанцирования должна начинаться с ключевого слова `template`; за ним следует *объявление* объекта, экземпляр которого со всеми выполненными подстановками надо генерировать. В нашем примере (см. листинг 13.12) в функции `main()` четыре вызова функции `Summa()`. Поэтому в файле с директивами инстанцирования у нас четыре директивы с аргументами как раз тех типов, которые используются во время вызовов функции `Summa()`.

Обратите внимание, что в модуле с функцией `main()` (см. листинг 13.12) мы не подключаем ни шаблон, ни файл с директивами явного инстанцирования. Нам в этом модуле необходим только прототип шаблона. Тем не менее все транслируется, собирается и выполняется правильно.

Можно сочетать модель явного инстанцирования и модель включения. Например, можно подключить файл явного инстанцирования `SummaInst.h` в модуле `main.cpp`:

```
#include "SummaInst.h"
```

Естественно, прототип функции-шаблона в этом случае можно убрать.

Теперь рассмотрим модель явного инстанцирования для классов. Включим в проект следующие модули:

- файл `TStack.h` с определением интерфейса класса-шаблона `TStack` (см. листинг 13.10);
- файл `TStackdef.h` с реализацией методов класса-шаблона `TStack` (см. листинг 13.10);

- файл `main.cpp` с главной функцией (см. листинг 13.11);
- файл `TStackInst.cpp` с директивами явного инстанцирования (листинг 13.14).

Листинг 13.14. Модуль явного инстанцирования для шаблона стека

```
#include "TStackdef.h"
#include <string>
using std::string;
template class TStack<double>;
template class TStack<string>;
```

ПРИМЕЧАНИЕ

В системе Visual C++.NET 2003 директивы явного инстанцирования можно писать без ключевого слова `class`, например `template TStack<double>;`; это очевидная ошибка разработчиков компилятора.

Обратите внимание на два подключения:

- к главной функции (см. листинг 13.11) подключается только определение интерфейса класса-шаблона `TStack`;
- к модулю с директивами инстанцирования (см. листинг 13.14) подключается модуль с реализациями методов класса-шаблона (см. листинг 13.10).

Таким образом, явное инстанцирование обеспечивает нам как раз такую организацию файлов с шаблонами, которую мы хотели иметь с самого начала. Однако этот способ тоже не свободен от недостатков. Как указано в [28], программист должен тщательно следить за тем, какой тип шаблона инстанцируется. Для больших проектов это быстро становится слишком обременительным, поэтому применять этот метод в промышленных проектах не рекомендуется¹.

ПРИМЕЧАНИЕ

В [28] описан определенный еще в стандарте [1] механизм экспорта шаблонов, который обеспечивает организацию файлов с шаблонами, известную как модель разделения. Однако на момент написания книги очень мало компиляторов поддерживали эту модель. В частности, ни Visual C++.NET 2003, ни C++ Builder 6 эту модель не поддерживают, поэтому останавливаться мы на ней не будем.

Межмодульное взаимодействие

Как указано в стандарте (см. п. п. 3.2/1 в [1]), единица трансляции не должна содержать более одного определения любой переменной, функции, класса, перечисления и шаблона (в каждой области видимости, естественно). Это — так называемое правило одного определения (One-Definition Rule, ODR). В первую очередь это правило касается определений классов, переменных и функций. Для переменной компилятор по ее определению вычисляет размер и резервирует

¹ Я думаю, разработчики интегрированных сред в конце концов автоматизируют процесс создания модулей явного инстанцирования — это не очень сложно сделать, выбирая из текста строки с инстанцированными шаблонами.

место в памяти, поэтому при наличии более одного определения в одной области видимости у него «возникают проблемы». Для класса компилятор тоже вычисляет размер, хотя обычно и не резервирует место в памяти — это делается при объявлении объектов класса.

ПРИМЕЧАНИЕ

Для статических полей класса память резервируется до первого объявления объекта класса — при определении статического поля.

Для функции компилятор генерирует код, который в конечном счете тоже займет свое место в памяти.

А вот объявлений может быть несколько. Объявление лишь добавляет некоторое имя в данную область видимости и обычно используется для согласования типов. Однако при разделении программы на отдельные модули возникают вопросы о взаимодействии определений и объявлений, прописанных в разных модулях. Естественно, речь не идет о локальных объектах — с ними все ясно.

Межмодульные переменные и функции

Начнем с простых переменных. Допустим, у нас есть два модуля: А.сpp и В.сpp. В модуле А *определена* целая переменная *i* вне всех классов и функций:

```
int i = 2;
```

Такая переменная называется глобальной. В файле А она видна от точки определения и до конца файла. Однако в модуле В эта переменная не видна. И если вдруг нам потребуется в модуле В присвоить ей другое значение, у нас возникнут некоторые проблемы. Нельзя просто написать:

```
i = 1;
```

В этом случае компилятор при обработке модуля В «не видит» модуль А и ничего не знает об определенной там переменной, поэтому мы получим сообщение о неопределенной переменной. Также нельзя написать:

```
int i = 1;
```

Такая запись является повторным определением. Компилятор-то «возражать» не станет — он транслирует модули по отдельности, а вот компоновщик будет «воротить нос» и сообщит, что одна и та же переменная определена дважды. Для таких случаев в С++ включено специальное ключевое слово `extern`. В модуле В надо объявить переменную следующим образом:

```
extern int i;
```

После этого можно использовать переменную *i* в файле В любым разрешенным способом. Например, присвоить новое значение:

```
i = 1;
```

Однако попытка совместить объявление с присвоением значения является ошибкой:

```
extern int i = 1;
```

Такая запись служит определением, поэтому мы опять получим от компоновщика сообщение о повторном определении.

ПРИМЕЧАНИЕ

Хотя ключевое слово `extern` в стандарте определено как один из четырех классов хранения, проще понимать его как обозначение «внешнего» имени для данного модуля. Имя называется внешним по отношению к модулю, если объект с этим именем не определен в данном модуле.

Аналогичная картина наблюдается и с функциями. Определение функции включает тело, а объявлением является прототип. Пусть в модуле А определена функция:

```
void f(void)
{ cout << "f()" << endl; }
```

Для того чтобы эту функцию можно было использовать в модуле В, нужно объявить там ее прототип:

```
void f(void);
```

Слово `extern` писать не требуется, хотя и не запрещается. Следующие прототипы эквивалентны:

```
void f(void);
extern void f(void);
```

Локализация имен в модуле

Итак, определив глобальную переменную или функцию в некоторой единице трансляции, мы должны придерживаться определенных правил, чтобы не возникло конфликтов имен. Говорят, что для имен глобальных переменных и функций применяется *внешняя компоновка* (*external linkage*), то есть эти имена становятся видны компоновщику во время компоновки программы.

Однако иногда бывает нужно, чтобы глобальная переменная или функция были видны только в том файле, где определены. Это позволяет сделать атрибут `static`, например:

```
static int a = 1;
static void f(void) { ... }
```

Для определенных таким образом имен применяется *внутренняя компоновка* (*internal linkage*) — они являются локальными в модуле, где определены. Таким образом, можно говорить, что глобальные имена обладают свойством внешней или внутренней компоновки.

Разберемся с некоторыми подробностями на примере функций. Пусть у нас есть, как обычно, два модуля, А.cpp и В.cpp:

```
//--модуль А.cpp
void f1(void) {...};           // определение глобальной функции
static void f2(void){...};     // определение локальной функции
//--модуль В.cpp
f1();                          // вызов глобальной функции
f2();                          // ОШИБКА!! -- вызов невидимой функции
```

Функция `f2()` не видна в модуле В, поэтому ее вызов ведет к ошибке трансляции. Имя функции с атрибутом `static` должно быть уникальным в данном модуле, но может повторяться в других модулях. Более того, локальная в модуле функция (с атрибутом `static`) перекрывает глобальную функцию в пределах модуля (аналогично тому, как локальная переменная в теле функции перекрывает глобальную). Например, в следующем примере функция `f1()` в модуле `B.cpp` перекрывает глобальную функцию, определенную в модуле `A.cpp`.

```
//--модуль A.cpp
void f1(void) {...};           // определение глобальной функции
static void f2(void){...};     // определение локальной функции
f1();                          // вызов глобальной функции
//--модуль B.cpp
static void f1(void) {...};     // определение локальной функции
f1();                          // вызов локальной функции
```

Все то же самое относится и к глобальным переменным с атрибутом `static`.

Атрибут `static` в данном случае похож на модификатор доступа `private`, работающий на уровне файла. Налицо два совершенно разных смысла одного ключевого слова: с одной стороны, для локальных переменных `static` означает класс хранения (в статической памяти); с другой стороны, на уровне модуля атрибут `static` имеет смысл ограничителя видимости имен. Последнее объявлено устаревшим, поэтому применение атрибута `static` в таком значении является нежелательным (см. п. D2 в [1]). Вместо этого для решения проблем локализации в C++ включили *пространства имен*, которые мы рассмотрим далее.

Константы *по умолчанию* компоуются внутренним образом. Это означает, что при объявлении константы нет необходимости указывать атрибут `static`, чтобы сделать ее локальной в модуле. Поэтому в разных модулях можно объявлять глобальные константы с одинаковыми именами — конфликта при компоновке не происходит. Более того, чтобы сделать константу, объявленную в одном модуле, видимой в другом, нужно использовать слово `extern`:

```
// модуль A.cpp
extern const int a = 2;
// модуль B.cpp
extern const int a;
```

Тогда компоновщик будет считать, что в модулях `A.cpp` и `B.cpp` используется одна и та же целая константа с именем `a`. Если мы пропустим слово `extern` в объявлении константы в модуле `A.cpp`, то получим локализованную константу и при обработке объявления в модуле `B.cpp` компоновщик выдаст сообщение о неопределенном имени.

Внутренняя компоновка констант оказывает «медвежью услугу» в шаблонах. Поскольку строковые литералы — это объекты со свойством внутренней компоновки, использовать их в качестве аргумента шаблона не разрешается:

```
template<char const *str>
class Template{ ... };
Template<"literal"> T;           // Ошибка!
```

Нельзя использовать и глобальный указатель:

```
template<char const *str>
class Template { ... };
char const *s = "Literal";
Template<s> T; // Ошибка!
```

Однако глобальный символьный массив использовать можно:

```
template<char const *str>
class Template { ... };
char const s[] = "Literal";
Template<s> T; // Ошибки нет!
```

Для функций, объявленных как подставляемые (с ключевым словом `inline`), по умолчанию тоже применяется внутренняя компоновка. Функция, определенная в одном модуле, не видна в другом модуле. Даже если определения в модулях абсолютно синтаксически совпадают — это все-таки могут быть разные функции, например:

```
// модуль A.cpp
const int a = 7;
inline void f(void)
{ cout << a << "f()\n"; }
// модуль B.cpp
const int a = 10;
inline void f(void)
{ cout << a << "f()\n"; }
```

При вызове первой функции в модуле А получим на экране `7f()`, а при вызове второй в модуле В на экране появится `10f()`.

Так же как и определение класса, определение подставляемой функции может быть включено в программу несколько раз — по одному определению в каждом модуле. Чтобы гарантированно иметь в разных модулях одно и то же определение, мы вынесем его в отдельный модуль и подключим этот модуль в нужных файлах с помощью директивы `#include` — компоновщик против не будет. Естественно, определение не должно зависеть от локализованных имен, как в приведенном ранее примере.

Можно сделать подставляемую функцию глобальной — точно так же, как и константу, указав в определении ключевое слово `extern`:

```
// модуль A.cpp
// Определение глобальной inline-функции
extern inline void f(void)
{ cout << a << "f()\n"; }
```

В другом модуле достаточно указать прототип:

```
// модуль B.cpp
// объявление внешней inline-функции
void f(void);
```

К прототипу можно добавить спецификаторы `extern` и `inline`:

```
extern inline void f(void);
```

Как видите, в этом случае можно обойтись без подключения модуля с определением.

Альтернативные спецификации компоновки

Одним из важнейших принципов C++ является обратная совместимость с C, поэтому в программе на C++ позволено вызывать внешние функции, написанные на C. Например, можно воспользоваться некоторой нестандартной библиотекой, написанной на C. Обычно библиотеки предоставляются в объектном коде, и программисту сообщаются прототипы входящих в библиотеку функций. Файл библиотеки просто подключается к проекту, а в исходный код на C++ прописываются прототипы используемых функций. Предположим, в библиотеке есть функция с прототипом:

```
double f(int, double);
```

Если мы в программе на C++ напишем прототип в таком виде, то получим ошибку компоновки. Дело в том, что компилятор C++ «украшает» имя функции лишними символами — после трансляции имя станет примерно таким: `_f_int_double`. Это делается для того, чтобы обеспечить перегрузку функций. Однако в C перегрузки функций нет, поэтому имена компилятором C не корректируются¹. Имена функций из библиотеки, естественно, не «украшались», поэтому они отличаются от тех, которые сгенерирует компилятор C++ по их прототипам. Для решения проблемы в стандарте определена *спецификация компоновки* (см. п. 7.5 в [1]). Чтобы приведенная ранее внешняя C-функция `f()` правильно компоновалась с программой на C++, необходимо прописать прототип следующим образом:

```
extern "C" double f(int, double);
```

Этим мы сообщаем компилятору, что функция `f()` будет компоноваться по правилам C, поэтому компилятор не должен корректировать ее имя. Стандарт определяет два вида спецификации компоновки: "C" и "C++", однако разработчики компиляторов могут реализовать аналогичную поддержку для других языков.

Группа объявлений с компоновкой C заключается в скобки:

```
extern "C"  
{ double f(int, double);  
  void print(int);  
}
```

Можно включить в спецификацию компоновки заголовочный файл C-библиотеки:

```
extern "C" { #include "cLibrary.h" }
```

Тогда все имена всех функций библиотеки будут скорректированы по правилам C, что нам и требуется.

Инициализация глобальных объектов

Глобальные (а также статические, как локальные, так и нет) переменные компилятор размещает в статической памяти, и время жизни таких переменных совпадает с временем выполнения программы. В стандарте указано (см. п. п. 3.6.2

¹ Вернее, корректируются, но не так радикально.

в [1]), что такие переменные, в отличие от локальных (не статических) и динамических, неявно инициализируются по умолчанию, причем до начала выполнения функции `main()`. Такая инициализация называется *статической*, в отличие от *динамической*, задаваемой программистом явно. Встроенные типы инициализируются нулями. Для объектов реализованных классов инициализация нулями обычно невозможна, поэтому для глобальных объектов невстроенных типов вызывается конструктор по умолчанию (без аргументов). Если в классе его нет, то возникает ошибка трансляции. Обратите внимание: вызывается именно конструктор без аргументов, а не конструктор инициализации. Последний применяется для явной инициализации.

В рамках одного модуля порядок инициализации переменных встроенных типов определяется порядком объявления. Конструкторы для создания и инициализации глобальных объектов тоже вызываются в порядке объявления этих объектов. Деструкторы вызываются перед завершением программы в обратном порядке.

Однако порядок статической инициализации глобальных объектов, размещенных в разных единицах трансляции, стандартом *не определен* — тут все зависит от компилятора и компоновщика. Проблемы нет, пока определения переменных в разных модулях не зависят друг от друга. Но если взаимозависимости избежать не удастся, то надо быть аккуратным, иначе можно получить разные результаты в зависимости от порядка компоновки модулей. Например:

```
// модуль A.cpp
extern int b;
int a = b+1;
// модуль B.cpp
extern int a;
int b = a+1;
```

Если модули будут инициализироваться в указанном порядке (сначала A.cpp, потом B.cpp), то в первую очередь переменная `b` статически инициализируется нулем, затем выполняется динамическая инициализация (в порядке объявления), и переменная `a` получает значение 1, а после этого переменная `b` становится равной двум. Если же инициализация модулей пойдет в обратном порядке (сначала B.cpp, потом A.cpp), то нулем статически инициализируется переменная `a`, потом переменная `b` получает значение 1, и после этого переменная `a` становится равной двум.

Таким образом, стабильную работу программы, в которой есть подобные взаимозависимые переменные, гарантировать нельзя. Брюс Эккель в [12] предлагает три основных решения этой проблемы.

1. Не создавать себе проблем — обойтись без зависимостей глобальных объектов.
2. Если уж без зависимостей все же не обойтись, по крайней мере, разместите все зависимые глобальные переменные в одном модуле. Расставляя определения объектов в нужном порядке, можно управлять их инициализацией.
3. Если же без распределения глобальных объектов совершенно нельзя обойтись, то нужно использовать программное решение. Их существует даже два, но мы рассмотрим наиболее простое и понятное.

Проблема решается с помощью функций-оболочек — по одной на каждый глобальный объект. Нужная нам переменная объявляется статической внутри функции, а функция возвращает ссылку на нее. Таким образом, время жизни переменной по-прежнему будет совпадать с временем выполнения программы, а инициализацией мы сможем управлять, вызывая функции в нужном месте и в нужном порядке. Простой пример демонстрирует эту методику (листинг 13.15).

Листинг 13.15. Управляемая инициализация статических объектов

```
// -----модуль FirstClass.h
// от этого класса (FirstClass) зависит второй (SecondClass)
#ifdef _FIRST
#define _FIRST
class FirstClass
{
    unsigned int Year;
public:
    FirstClass();
    void print(void)const;
};
#endif
//----- модуль SecondClass.h
// этот класс (SecondClass) зависит от первого (FirstClass)
#ifdef _SECOND
#define _SECOND
#include "FirstClass.h"
class SecondClass
{
    FirstClass t; // вот зависимость от порядка инициализации
public:
    SecondClass(const FirstClass& x);
    void print(void)const;
};
#endif
//----- модуль implementation.cpp
#include <iostream>
// реализация класса FirstClass
#include "FirstClass.h"
FirstClass::FirstClass():Year(2000) // инициализация по умолчанию
{ std::cout << "FirstClass constructor " << Year << std::endl; }
void FirstClass::print(void)const
{ std::cout << "FirstClass " << Year << std::endl; }
// реализация класса SecondClass
#include "SecondClass.h"
SecondClass::SecondClass(const FirstClass& x):t(x)
{ std::cout << "SecondClass "; print(); }
void SecondClass::print(void)const { t.print(); }
//----- модуль FirstClassFunction.cpp
#include "FirstClass.h"
FirstClass& dl(void)
{ static FirstClass d; // статический объект
  return d; // возврат ссылки на него
}
//----- модуль SecondClassFunction.cpp
#include "FirstClass.h"
#include "SecondClass.h"
```


Листинг 13.15 (продолжение)

```

FirstClass& d1(void);
SecondClass& d2()
{ static SecondClass d(d1());          // статический объект
  return d;                            // возврат ссылки на него
}
//----- модуль main.cpp
#include <iostream>
#include "FirstClass.h"
#include "SecondClass.h"
// моделирует неверный порядок инициализации
extern FirstClass dd1;
SecondClass dd2(dd1);
FirstClass dd1;
// правильная инициализация
SecondClass& d2();
int main()
{   d2();                             // гарантирует правильный порядок инициализации
    return 0;
}

```

Проект включает 5 модулей: файлы с определениями классов `FirstClass.h` и `SecondClass.h`, файл `implementation.cpp` с реализацией обоих классов, файлы `FirstClassFunction.cpp` и `SecondClassFunction.cpp` с определениями функций и файл `main.cpp`, в котором указан порядок инициализации. В файлах `implementation.cpp`, `FirstClassFunction.cpp` и `SecondClassFunction.cpp` для наглядности подключаются оба файла-заголовка. Если бы в файле `SecondClass.h` не был прописан страж, это привело бы к повторному определению класса `FirstClass`. В принципе достаточно подключать только `SecondClass.h` — тогда стража можно убрать.

Зависимость от порядка инициализации проявляется в том, что объект типа `SecondClass` должен инициализироваться объектом `FirstClass`. Поэтому объект типа `FirstClass` должен быть уже проинициализирован к моменту определения объекта типа `SecondClass`. Для наглядности в главной программе моделируется неправильный порядок инициализации — глобальные объекты определяются в обратном порядке. В результате на экран выводятся следующие строки:

```

SecondClass FirstClass 0
FirstClass constructor 2000

```

Это означает, что в момент определения объекта `dd2` типа `SecondClass` объект `dd1` типа `FirstClass` не был проинициализирован — конструктор для создания `dd1` вызван после конструктора, создающего `dd2`. Подчеркнем, что такая ситуация *может* сложиться, если глобальные объекты зависимых классов определяются в *разных* модулях.

Вместо явного определения объектов мы реализовали функции. Функция `d1()` реализует определение объекта типа `FirstClass`, а функция `d2()` — определение объекта типа `SecondClass`. Самы объекты объявлены в функциях как статические, следовательно, время жизни их совпадает с временем жизни глобальных объектов. В функции `d2()`, реализующей определение объекта типа `SecondClass`, мы

и управляем зависимостью, передавая конструктору в качестве аргумента вызов функции `d1()`, которая возвращает ссылку на статический объект. В главной функции мы в нужном месте просто прописываем вызов функции `d2()`, создающей объект типа `SecondClass`. В результате на экран выводятся строки:

```
FirstClass constructor 2000
SecondClass FirstClass 2000
```

Это и означает, что конструкторы вызываются в правильном порядке.

Еще раз подчеркнем, что все описанное касается ситуации, когда глобальные объекты зависимых классов определяются в разных модулях. Стандарт не гарантирует правильного порядка инициализации таких объектов, поэтому мы должны этим явно управлять.

Статические элементы в шаблонах

В классе-шаблоне, конечно, разрешено объявлять и статические методы, и статические поля — мы уже делали это, определяя шаблон для подсчета объектов (см. листинг 11.13). Однако нужно учесть, что каждая конкретизация обладает собственной копией статических членов. Это естественно, так как при разных инстанцированиях получаются фактически разные конкретные классы. Рассмотрим простой пример — напомним простой класс-шаблон со статическими полями и статическим методом для вывода на экран (листинг 13.16). Подключать его к другим модулям программы сначала будем в соответствии с моделью включения.

Листинг 13.16. Класс-шаблон со статическими элементами

```
//-----файл StaticField.h
#ifndef StaticField
#define StaticField
#include <iostream>
template <class T>
class Static
{
    static T t; // зависимое имя
    static int count; // независимое имя
public:
    Static () { ++count; } // конструктор считает объекты
    ~Static () { --count; } // деструктор
    static void print(T x = t);
};
template <class T> int Static<T>::count = 0; // счетчик объектов
template <class T> T Static<T>::t = T(); // определение t
template <> int Static<int>::t = 55; // специализация t
template <> double Static<double>::t = 155; // специализация t
template <class T> // реализация метода
void Static<T>::print(T x)
{ std::cout << x << " " << count;
  t = x;
}
#endif
```

В классе определен счетчик объектов — статическое поле `count`. Это поле не зависит от параметра шаблона, тем не менее при его определении мы обязаны соблюдать синтаксис определения элементов класса-шаблона.

Определение статического поля `t`, тип которого зависит от шаблона, написано с инициализацией конструктора без аргументов `T()`, хотя можно этого не делать, так как он и так будет вызван по умолчанию. Стандарт разрешает специализировать статические поля, что мы и проделали для аргументов `int` и `double`. Специализаций статического поля вполне достаточно, чтобы компилятор смог инстанцировать классы `Static<int>` и `Static<double>`. Соответственно, для каждого из классов создаются статические поля-счетчики `count`; в первом классе инстанцируется статическое поле `static int t`, которому присваивается значение 55, и статический метод `void print(int)`. Во втором классе инстанцируются статическое поле `static double t`, которое инициализируется значением 155.0, и статический метод `void print(double)`.

Далее прописано обычное определение метода класса-шаблона, метод получает в качестве параметра по умолчанию статическое поле класса и как побочный эффект присваивает значение своего параметра статическому полю.

Поэкспериментируем¹ с этим классом-шаблоном и его статическим методом `print()` (листинг 13.17).

Листинг 13.17. Статические элементы шаблона — модель включения

```
#include "StaticField.h"                                // модель включения
#include <iostream>
using namespace std;
Static<long> r;                                         // глобальный объект
int main()
{
    // объектов типа Static<double> не существует
    Static<double>::print(); cout << endl;             // значение по умолчанию
    Static<double>::print(234); cout << endl;          // изменили значение поля t
    Static<double> r;                                  // локальный объект создан
    Static<double>::print(); cout << endl;             // значение поля t = 234.0
    // один объект типа Static<long> уже существует!
    Static<long>::print(); cout << endl;               // значение 0
    Static<long>::print(4321); cout << endl;           // изменили значение поля t
    r.print(); cout << endl;                           // локальный объект
    ::r.print(); cout << endl;                         // глобальный объект
    Static<int> t;
    Static<int>::print(); cout << endl;
    Static<int>::print(100); cout << endl;
    return 0;
}
```

Сначала объявляется глобальный объект `r`. Соответственно создаются статические элементы класса `Static<long>`:

- ☐ статическое поле `count`, не зависящее от аргумента шаблона;
- ☐ статическое поле `static long t`, проинициализированное нулем;
- ☐ статический метод `void print(long)`.

¹ В Visual C++.NET 2003.

Создать глобальный объект того же типа с тем же именем в другом модуле нам не удастся — компоновщик сообщит о повторном определении.

Далее входим в функцию `main()` и вызываем статический метод:

```
Static<double>::print();
```

Программа выдаст на экран

```
155 0
```

Выводится значение поля `static double t`, присвоенное параметру по умолчанию, и количество объектов типа `Static<double>`, равное нулю — это естественно, так как объектов мы не создавали. Далее вызывается тот же метод, но с явно заданным параметром 234. На экран выводится

```
234 0
```

Количество объектов по-прежнему равно нулю.

Создаем локальный объект типа `Static<double>` с именем `r`, он перекрывает глобальный объект `r` типа `Static<long>`. Тем не менее статические поля `count` и `t` заново не создаются. При вызове метода `Static<double>::print();` на экран выводится

```
234 1
```

Это означает, что используется значение того же поля `t`, которое было изменено в предыдущем вызове, и существует один объект.

Далее выполняются два вызова:

```
Static<long>::print(); cout << endl;  
Static<long>::print(4321); cout << endl;
```

Первый выводит на экран числа 0 и 1, которые соответствуют значению поля `static long t` и количеству объектов типа `Static<long>`. Второй вызов выводит 4321 и 1 соответственно.

Далее программа демонстрирует вызов статического метода для конкретного объекта: локального типа `Static<double>` и глобального типа `Static<long>`. Тем не менее выводятся те значения поля `t`, которые были присвоены ранее.

И наконец, демонстрируется вывод и изменение значения статического поля `t` для специализации `int`: сначала значение, присвоенное при определении, а затем новое.

Теперь добавим в проект модуль с функцией (листинг 13.18), объявив в модуле `main.cpp` ее прототип, и вызовем ее в главной функции.

Листинг 13.18. Модуль с функцией

```
#include "StaticField.h"                // включение  
#include <iostream>  
using namespace std;  
Static<int> t;                          // глобальный объект  
void f (void)
```

продолжение ➤

Листинг 13.18 (продолжение)

```
{ cout << "Inside f()-t ";
  t.print(); cout << endl;
  int y = 123;
  cout << "Inside f()-t ";
  Static<int>::print(y); cout << endl;
}
```

Объявляется глобальный объект типа `Static<int>` и демонстрируются вывод и изменение его статического поля `static int t`. В главной функции `main()` после вызова функции `f()` вызов метода `Static<int>::print()` показывает, что статическое поле `static int t` — единственное, так как выводится значение, присвоенное полю в функции `f()`. Оно остается единственным даже для динамического объекта того же типа, что можно наблюдать, добавив в программу следующие строки:

```
// указатели!!
Static<int> *t = new Static<int>;
t->print(); cout << endl;
t->print(100); cout << endl;
delete t;
```

Теперь сделаем то же самое, используя модель явного инстанцирования: разделим модуль с шаблоном на два (интерфейс и реализацию) и добавим модуль с директивами инстанцирования (листинг 13.19).

Листинг 13.19. Статические элементы шаблона — модель с явным инстанцированием

```
//-----файл StaticField.h
#ifndef StaticField
#define StaticField
#include <iostream>
template <class T>
class Static
{
    static T t; // зависимое имя
    static int count; // независимое имя
public:
    Static () {++count; } // конструктор
    ~Static () {--count; } // деструктор
    static void print(T x = t);
};
#endif

//-----файл StaticField.cpp
#ifndef StaticFieldDef
#define StaticFieldDef
#include "StaticField.h" // подключение определения шаблона
// определение статических элементов
template <class T> int Static<T>::count = 0;
template <class T> T Static<T>::t = T();
template <> int Static<int>::t = 55;
template <> double Static<double>::t = 155;
template <class T>
void Static<T>::print(T x)
{ std::cout << x << " " << count;
  t = x;
}
#endif
```

```
// -----модуль явного инстанцирования
#include "StaticField.cpp" // подключение определений
template class Static<int>;
template class Static<long>;
template class Static<double>;
// -----модуль с функцией f()
#include "StaticField.h" // подключение интерфейса шаблона
#include <iostream>
using namespace std;
Static<int> t;
void f (void)
{ cout << "Inside f()-t "; t.print(); cout << endl;
  int y = 123;
  cout << "Inside f()-t ";
  Static<int>::print(y); cout << endl;
}
// -----модуль main.cpp
#include "StaticField.h" // подключение шаблона
#include <iostream>
using namespace std;
void f(void); // объявление внешней функции
Static<long> r; // глобальный объект
int main()
{ Static<double>::print(); cout << endl;
  Static<double>::print(234); cout << endl;
  Static<double> r; // локальный объект
  Static<double>::print(); cout << endl;
  // вывод для Static long t
  Static<long>::print(); cout << endl;
  Static<long>::print(4321); cout << endl;
  r.print(); cout << endl; // вывод для Static<double> r;
  r.print(); cout << endl; // вывод для Static<long> r;
  f();
  // указатели!!
  Static<int> *t = new Static<int>;
  t->print(); cout << endl;
  t->print(100); cout << endl;
  delete t;
  return 0;
}
```

Эта программа компилируется и компоуется без ошибок. Результаты, выводимые на экран, совпадают с результатами выполнения программы, реализованной в соответствии с моделью включения.

Пространства имен

Каждое имя, обозначающее объект, имеет некоторую область видимости (действия), в которой это имя объявлено и может использоваться. В стандарте (см. п. 3.3 в [1]) определены следующие области видимости (по возрастанию «объема»):

- **Оператор** (см. п. п. 3.3.2 в [1]). Идентификатор, объявленный в условии операторов `if`, `switch`, `while` или в операторе `for`, действителен только до конца тела оператора.

- *Прототип* (см. п. п. 3.3.3 в [1]). Идентификаторы, указанные в списке параметров прототипа функции, имеют область действия только прототип функции.
- *Блок* (см. п. п. 3.3.2 в [1]). Имена, объявленные в блоке, являются локальными в этом блоке и не видимы вне его. Параметры, объявленные в заголовке функции, видимы до конца блока определения и локальны в этом блоке¹. Если тело функции является функциональным блоком `try`, то область видимости параметров продолжается до последней секции-ловушки. Параметр, объявленный в блоке `catch`, локален в блоке данного обработчика.
- *Функция* (см. п. п. 3.3.4 в [1]). Только метки имеют такую область действия. В одной функции все метки должны быть разными.
- *Класс* (см. п. п. 3.3.6 в [1]). Все имена, объявленные в классе, видимы внутри него без ограничений независимо от модификатора доступа. Только дружественные классы и функции имеют неограниченный доступ к элементам данного класса. Вне класса доступные имена (это зависит от модификатора доступа) могут быть использованы только с квалификаторами (`.`, `::` и `u ->`).
- *Файл*. Имена объектов, определенных вне указанных выше областей видимости, видимы в пределах единицы трансляции. За счет объявлений такие имена можно использовать и в других единицах трансляции. С помощью ключевого слова `static` видимость можно ограничить данной единицей трансляции.
- *Пространство имен* (см. п. п. 3.3.5 в [1]). C++ позволяет явно задать область видимости имен, присвоив области видимости некоторое имя.

Имена в одной области видимости не должны быть одинаковыми, но в разных областях они могут совпадать.

Именованные пространства имен

Разделение программы на части и последующее их объединение в одну программу может привести к конфликтам имен, когда в разных модулях, написанных разными программистами, определены глобальные объекты с одинаковыми именами. Для небольшой программы из пары десятков модулей отследить вхождения имени и заменить его другим труда не составляет. Однако в больших проектах, состоящих из сотен и тысяч модулей, это может превратиться в проблему. В стандарте C++ определен механизм, с помощью которого можно глобальную совокупность имен разделить на части — *пространство имен* (см. п. 7.3 в [1]).

Как написано на с. 211 в [2], пространства имен — это механизм отражения логической структуры программы. Если некоторые объявления и определения можно объединить по тому или иному критерию, их следует поместить в одно пространство имен. Например, можно поместить все имена, относящиеся к вводу-выводу,

¹ Это правило работает и для методов класса.

в пространство имен `IO`. В стандарте определено *стандартное пространство имен* `std`, которым мы неоднократно пользовались.

Пространство имен может иметь имя. Объявление пространства имен — это назначение имени для области видимости, в которую будут входить компоненты пространства имен. Общий синтаксис объявления выглядит так:

```
namespace имя  
{ // объявления и определения }
```

Идентификатор `namespace` является зарезервированным словом. Пространство имен может включать в себя объявления и определения переменных, функций, классов, типов, шаблонов и т. д. Эти имена считаются членами данного пространства имен. Определения, естественно, должны быть в единственном числе (по правилу одного определения). Рассмотрим простой пример:

```
namespace MySpace {  
    void f1(void) {...};  
    int x;  
    void f2(void) {...};  
    int y = 6;  
    class A {};  
}
```

В пространстве имен `MySpace` представлены пять определений: функции `f1()` и `f2()`, целые переменные `x` и `y`, класс `A`.

В пространство имен можно включать и заголовки, например:

```
namespace RSDN {  
#include "TStack.h"  
}
```

Таким способом мы включаем определение класса `TStack` в пространство имен `RSDN`.

Доступ к элементам пространства имен (в той же единице трансляции, но вне его, или в другой единице трансляции) выполняется при помощи операции разрешения контекста (`::`):

<пространство имен>::*имя компонента*

Идентификатор пространства имен служит квалификатором для имени компонента, например:

```
MySpace::f();
```

Еще пример:

```
MySpace::x = 6;
```

Для стандартного пространства имен это выглядит так:

```
std::cout
```

Другой пример:

```
std::endl
```


Однако члены пространства имен могут использовать имена «товарищей по партии» без указания квалификатора. Например, мы можем вынести определения функций из пространства имен:

```
namespace MySpace {
    void f1(void);           // объявление
    int x;                  // определение
    void f2(void);          // объявление
    int y = 6;              // определение
    class A{};              // определение
}
// внешние определения функций
void MySpace::f1(void)      // квалификатор
{ std::cout << x << std::endl }; // x – без префикса-квалификатора
void MySpace::f2(void)
{ std::cout << y << std::endl }; // y – без квалификатора
```

Как видите, определения функций очень похожи на внешние определения методов класса — указывается квалификатор пространства имен, в котором они «прописаны». Однако внутри функций имена переменных *x* и *y* из того же пространства имен используются без квалификатора. Сходство с определениями методов класса не случайно — класс тоже задает пространство имен.

Понятно, что программистам каждый раз писать префикс-квалификатор быстро надоедает¹. Чтобы не писать имена с квалификаторами, можно использовать объявления `using`:

```
using <пространство имен>::<имя компонента>;
```

Слово `using` так же, как и `namespace`, является зарезервированным.

Мы уже использовали эту конструкцию при закрытом наследовании, чтобы открыть имена в классе-наследнике (см. листинг 8.13). Для стандартных имен из пространства `std` это выглядит так:

```
using std::cout;
using std::endl;
```

После этого объявленные имена можно использовать в программе без квалификатора. Однако большое количество имен писать тоже утомительно, поэтому в C++ добавили еще и директиву `using` (см. п. п. 7.3.4 в [1]):

```
using namespace <пространство имен>;
```

После этого все имена из указанного пространства имен можно писать без префикса. Однако действует такая директива, естественно, только до конца единицы трансляции. На другие файлы ее действие не распространяется, поэтому можно осуществлять управление пространствами имен на уровне отдельных файлов. Мы неоднократно использовали эту директиву для задания стандартного пространства `std`:

```
using namespace std;
```

¹ Все-таки интересная психология у программистов — они готовы сутками работать над решением сложной задачи, но писать длинные имена им лень!

Эту директиву можно писать всякий раз, когда у вас в программе выполняется включение стандартного заголовка из списка, указанного в стандарте (см. п. п. 17.4.1.2 в [1]).

Любую форму `using` можно использовать внутри пространства имен, чтобы включить в него имена из другого пространства имен.

Именованные пространства имен позволяют легко избавиться от возможных конфликтов имен. Если два программиста работают над разными частями одной большой программы, то каждый из них может объявить собственное пространство имен (область видимости) и больше не беспокоиться о возможных повторных объявлениях. Главное, чтобы имена пространств имен не совпадали.

Однако старайтесь не использовать глобальную директиву `using` в заголовочных файлах — это с большой вероятностью может привести к конфликтам имен. Любой другой файл, включивший такой заголовок, «открывает» все имена. Если в этом файле определены такие же имена, то получаем конфликт имен, хотя директива `using` в файле явным образом не указана — она «спряталась» в заголовке!

СОВЕТ

Не используйте глобальную директиву `using` в заголовочных файлах.

Стандарт C++ позволяет объявлять *синонимы* (alias) имен, например:

```
namespace RSDN = Russian_Software_Development_Network;
```

Обычно синонимы используются для переопределения длинных имен, придуманных разработчиками библиотек.

Пространство имен разрешается разбивать на части, например:

```
//--модуль m1.cpp
namespace RSDN {
    void f1(void) {...};
    int x;
}
//--модуль m2.cpp
namespace RSDN {
    void f2(void) {...};
    int y;
    class A{};
}
```

Если в разных единицах трансляции объявлено одно и то же пространство имен, то оно «склеивается» в единое пространство:

```
namespace RSDN {
    void f1(void) {...};
    int x;
    void f2(void) {...};
    int y;
    class A{};
}
```

Именно таким образом «склеивается» стандартное пространство `std`, которое можно наблюдать в системных заголовочных файлах, размещенных в каталоге `include` интегрированной среды.

Это свойство «склеивания» пространства имен можно использовать для организации библиотеки. Аналогично классам, мы разобьем исходный код на две части: интерфейс и реализацию. Интерфейсную часть поместим в отдельный `h`-файл, указав в нем только объявления, например:

```
// ---- matrix.h ----
namespace matrix {
    class matrix { /* ... */ };
    const double pi = 3.141615926;
    matrix operator+(const matrix &m1, const matrix &m2);
    void transposition(matrix &);
}
```

В файл `matrix.cpp` поместим реализацию:

```
// -----matrix.cpp-----
#include "matrix.h"
namespace matrix {
    matrix operator+(const matrix &m1, const matrix &m2)
    { /* ... */ }
    void transposition(matrix &m)
    { /* ... */ }
} // реализация методов класса matrix...
```

Тогда в клиентском коде можно будет подключать только заголовок:

```
// -----user.cpp-----
#include "matrix.h"
//...
void f(matrix::matrix &m)
{ //...
    matrix::transposition(m);
    //...
}
```

Как видно, такая организация пространства имен совершенно аналогична разделению класса на интерфейс и реализацию.

Пространства имен могут быть вложенными. В стандарте (см. п. п. 7.3/5 в [1]) приведен такой пример:

```
namespace Outer {                // внешнее пространство имен
    int i;
    namespace Inner {            // вложенное пространство имен
        void f() { i++; }        // работает Outer::i
        int i;
        void g() { i++; }        // работает Inner::i;
    }
}
```

Естественно, для обращения к именам из вложенного пространства нужно писать двойной префикс-квалификатор, например:

```
Outer::Inner::g();
```

Синонимы можно назначать и для вложенных пространств. Например, в справочной системе интегрированной среды Borland C++ Builder 6 приведен такой пример:

```
namespace BORLAND_SOFTWARE_CORPORATION {           // внешнее
    /* тело пространства имен */
    namespace NESTED_BORLAND_SOFTWARE_CORPORATION { // вложенное
        /* тело пространства имен */
    }
}
// синонимы пространства имен
// внешнее пространство
namespace BI = BORLAND_SOFTWARE_CORPORATION;
// внутреннее пространство задается с квалификатором
namespace NBI =
BORLAND_SOFTWARE_CORPORATION::NESTED_BORLAND_SOFTWARE_CORPORATION;
```

Как видим, при определении синонима внутреннего пространства имен указывается квалификатор — имя внешнего пространства.

Неименованные пространства имен

Имена, определенные на самом верхнем уровне, вне всех пространств имен, входят в *глобальное* пространство имен (см. п. п. 3.3.5/3 в [1]). Оно, естественно, не имеет имени и является единственным. Некоторые имена из именованных пространств могут совпадать с глобальными. Чтобы обратиться именно к глобальному имени, нужно использовать операцию разрешения контекста (::), так как по умолчанию (без указания квалификаторов) всегда выбирается имя с наименьшей областью видимости. Например, имена всех API-функций Windows входят в это глобальное пространство имен, поэтому лучше их указывать только таким способом:

```
int z = ::max(5,6);           // вызывается нестандартная функция C++
```

Для локализации имени в файле вместо атрибута `static` в C++ разрешается задавать *анонимные* (неименованные) пространства имен. Анонимные пространства имен являются локальными пространствами для единицы трансляции. Для каждого анонимного пространства компилятор генерирует уникальное «внутреннее» имя, поэтому анонимные пространства имен не «склеиваются». В справочной системе интегрированной среды Borland C++ Builder 6 приводится простой пример, который здесь немного модифицирован (листинг 13.20).

Листинг 13.20. Локальное пространство имен

```
// модуль m1.cpp
#include <iostream>
void func(void);           // функция в этом модуле не определена
namespace                  // анонимное пространство имен
{
    double pi = 3.1415926; // это имя pi доступно только в этом файле
}
```

продолжение ➤

Листинг 13.20 (продолжение)

```

int main()
{
    double pi = 0.1;           // 1: локальное определение того же имени
    std::cout << "pi = " << pi << std::endl;
    func();
    return 0;
}
//-----
// модуль m2.cpp
#include <iostream>
namespace                    // анонимное пространство
{
    float pi = 10.0001F;      // имя pi доступно только в этом файле
    void f(void)              // определение локальной функции
    {
        std::cout << "First func() called; pi = " << pi << std::endl;
    }
}
void func(void)              // определение глобальной функции
{
    f();                      // вызов локальной функции
    std::cout << "Second func() called; pi = " << pi;
}

```

Эта программа выводит на экран следующее:

```

pi = 3.14159
First func() called; pi = 10.0001
Second func() called; pi = 10.0001

```

В модуле `m1.cpp` определено локальное пространство имен, в котором задана константа `pi`. В функции `main()` строка 1 закомментирована, поэтому выводится значение `pi` из локального пространства имен. Если же комментарий удалить, то в операторе вывода будет использоваться имя с более ограниченной областью видимости (имя `pi` в теле главной функции). В этом случае мы никаким способом не сможем добраться до имени, определенного в локальном пространстве имен, — префикса-то нет!

Во втором файле `m2.cpp` определено свое локальное пространство имен, в котором прописано то же имя `pi`, но задана другая константа. В том же локальном пространстве имен определена локальная функция `f()`. Вне локального пространства определена глобальная функция `func()`, которая вызывает локальную версию `f()`. В модуле `m1.cpp` прописан прототип глобальной функции, чтобы ее можно было вызвать в этом модуле. В файле `m1.cpp` невозможно использовать ни функцию `f()`, ни имя `pi` из файла `m2.cpp`.

Резюме

Помимо логической декомпозиции, C++ позволяет физически разделить большую программу на части. Отдельный файл с исходным текстом является единицей трансляции. Объединение программ в единое целое выполняется двумя способами: исходные тексты объединяются препроцессором, а объектные модули собираются компоновщиком, который входит в состав любой интегрированной среды.

Класс обычно разделяется на две части: интерфейс, который по традиции помещается в `h`-файл, и реализацию, хранящуюся в `ccp`-файле. Интерфейс класса

является его определением, и стандарт разрешает иметь определение в каждом модуле, где это необходимо. Поэтому именно интерфейс подключается к модулю с помощью препроцессора. Чтобы не присоединять к модулю более одного определения класса, применяют «стражей» включения.

Использование шаблонов в большой программе, состоящей из нескольких модулей, представляет некоторые сложности. Подключение шаблона можно выполнять в соответствии либо с моделью включения, либо с моделью явного инстанцирования. Определенную в стандарте модель с разделением в настоящее время поддерживают очень мало компиляторов.

Разделение программы на части требует согласования объявлений и определений переменных, функций и классов, объявленных в разных модулях, — должно выполняться «правило одного определения». Глобальные и статические переменные по умолчанию инициализируются нулями, но порядок инициализации не определен. Если переменные из разных модулей зависят друг от друга, возможны проблемы, которые решаются с помощью некоторых приемов программирования.

В программе на C++ разрешается использовать C-функции. Для корректной компоновки необходимо объявить такие функции с альтернативной спецификацией компоновки `extern "C"`.

Одним из механизмов определения области видимости являются пространства имен. Пространства имен сами имеют имена; одним из именованных пространств имен является стандартное пространство `std`. Неименованное пространство имен обеспечивает локализацию имен в модуле. Пространства имен могут быть вложенными.

Контрольные вопросы

1. Назовите причины, требующие разделения программ на части.
2. Дайте определение термина «единица трансляции»?
3. Чем отличается файл с исходным текстом от единицы трансляции?
4. Существуют ли в C++ конструкции, позволяющие идентифицировать отдельный модуль?
5. Какие способы сборки программы вы можете назвать?
6. Что такое «объектный модуль»?
7. Как называется программа, которая «собирает» объектные модули в программу?
8. В чем разница между аргументами "файл" и <файл> в директиве `#include`?
9. Что такое ODR?
10. Объясните, что такое «страж» включения и зачем он нужен.
11. Является ли интерфейс класса его определением?
12. Сколько определений класса может быть в единице трансляции?
13. Сколько определений класса может быть в программе из нескольких файлов?
14. Чем различаются стандартные заголовки `<string>`, `<string.h>` и `<cstring>`?

15. Каковы функции программы «мейкер»?
16. Каким образом глобальную переменную, определенную в одной единице трансляции, сделать доступной в другой единице трансляции? А константу?
17. Можно ли использовать слово `extern` при объявлении функций?
18. Как локализовать объявление функции в файле?
19. Чем отличается «внешнее» связывание от «внутреннего»?
20. Что такое «спецификации компоновки»?
21. Для каких объектов по умолчанию характерно внутреннее связывание?
22. Какие области видимости имен вы знаете?
23. Для чего используются пространства имен?
24. Чем различаются именованные и неименованные пространства имен?
25. Сколько неименованных пространств имен может быть в программе?
26. Что такое «глобальное пространство имен»?
27. Могут ли пространства имен быть вложенными?
28. Для чего применяются синонимы в пространствах имен?
29. Как сделать члены одного пространства имен доступными в нескольких (в пределе — во всех) файлах программного проекта?
30. Объясните разницу между статической и динамической инициализацией.
31. В чем состоит проблема инициализации глобальных статических переменных?
32. Для чего применяются директивы явного инстанцирования?
33. Объясните, в чем состоят проблемы, возникающие при разделении шаблонного класса на интерфейс и реализацию?
34. Что такое «модель явного инстанцирования» и как она работает?
35. Какие проблемы вызывает внутреннее связывание в шаблонах?

Упражнения

1. Реализовать упражнения из глав 3 и 4 с разделением на интерфейс и реализацию.
2. Реализовать упражнения из глав 8 и 9, разделив каждый класс на интерфейс и реализацию.
3. Реализовать упражнения из главы 11 с использованием модели явного инстанцирования.
4. Реализовать упражнения из главы 8, поместив базовые классы в пространство имен `Base`, а наследников — в пространство имен `Derived`.
5. Реализовать упражнения из глав 5 и 6, разделив каждый класс на интерфейс и реализацию. Классы-контейнеры поместить в пространство имен `Library`.

Глава 14

Библиотека ввода-вывода

До изобретения языка С средства ввода-вывода всегда являлись частью языка программирования. Создатели С приняли новаторские решения, которые во многом способствовали успеху и языка, и операционной системы UNIX. Во-первых, средства ввода-вывода были отделены от языка и вынесены в отдельную библиотеку, получившую название `<stdio.h>`. Это — стандартная библиотека ввода-вывода, которая входит в стандарт С; в стандарте С++ она фигурирует под именем `<cstdio>` (см. п. п. 17.4.1.2/3 и 27.8.2 в [1]). Во-вторых, удалось разработать и реализовать концепцию ввода-вывода, независимого от устройств. Программа на С не имеет дела ни с устройствами, ни с файлами — она работает с *потоками*. Поток — это последовательность символов. Ввод информации осуществляется из *входного* потока, вывод программа производит в *выходной* поток. А уж поток можно связать и с устройством, и с файлом. Таким образом, чтобы выполнять ввод-вывод на С и С++, программисту необходим минимальный объем знаний о внешней среде — файловой системе. Практически все, что нужно знать (по крайней мере, на первых порах), — как записываются имена файлов в программе. Однако потоки — это очень простые абстракции, поэтому при сложной организации данных приходится писать много кода. Именно поэтому различные СУБД предоставляют собственные интерфейсы API для доступа к данным и их обработки. Библиотека `<cstdio>` является процедурно-ориентированной. Многолетняя практика использования библиотеки `<cstdio>` выявила как ее достоинства, так и недостатки, с которыми трудно мириться в С++. Да, библиотека переносима и эффективна. Однако функции библиотеки не обеспечивают контроль типов. Главный ее недостаток — библиотеку сложно расширять. Поэтому в С++ реализована новая библиотека ввода-вывода — объектно-ориентированная, получившая название `<iostream>`.

ПРИМЕЧАНИЕ

Именно так нужно писать заголовок в операторе `#include`. Заголовок `<iostream.h>` соответствует «достандартной» версии библиотеки ввода-вывода.

В этой библиотеке концепции «поточного» ввода-вывода, независимого от устройств, получили дальнейшее развитие. Мы уже знакомы с элементарными операциями, реализованными в этой библиотеке: операцией ввода из потока (`operator>>`) и операцией вывода в поток (`operator<<`). Далее мы рассмотрим концепции и конструкции стандартной объектно-ориентированной библиотеки.

Иерархия классов

Стандартная библиотека ввода-вывода описана на с. 605 стандарта (см. п. 27 в [1]). Прежде чем рассматривать возможности ввода-вывода, разберемся с иерархией основных классов библиотеки (листинг 14.1). Иерархия довольно сложная и построена с использованием шаблонов, а также виртуального и множественного наследования.

Листинг 14.1. Иерархия классов объектно-ориентированной библиотеки ввода-вывода

```
class ios_base { ... };
template<...>class basic_ios: public ios_base;
template<...>class basic_istream: virtual public basic_ios<...>;
template<...>class basic_ostream: virtual public basic_ios<...>;
template<...>class basic_iostream:      // множественное наследование
    public basic_istream<...>,
    public basic_ostream<...>;
template<...>class basic_fstream: public basic_iostream<...>;
template<...>class basic_stringstream: public basic_iostream<...>;
template<...>class basic_ifstream: public basic_istream<...>;
template<...>class basic_ofstream: public basic_ostream<...>;
template<...>class basic_istringstream: public basic_istream<...>;
template<...>class basic_ostringstream: public basic_ostream<...>;
template<...>class basic_streambuf { ... };
template<...>class basic_filebuf: public basic_streambuf<...>;
template<...>class basic_stringbuf: public basic_streambuf<...>;
```

Все классы включены в стандартное пространство имен `std`. Базовым классом является класс `ios_base` (см. п. п. 24.4.2 в [1]). Это — не шаблонный класс, и он не зависит от типа символов. В этом классе определены доступные пользователю константы, поля и методы, управляющие состоянием потока и форматированием.

Все остальные классы — шаблонные. Все классы-шаблоны имеют два обязательных параметра: тип символов и класс свойств (трактовок) символов. Например, определение шаблона `basic_ios` выглядит так:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
    class basic_ios;
}
```

Параметр `charT` определяет тип символов, а параметр `traits` — класс свойств. Кроме того, в пространстве `std` определены специализации шаблонов для двух типов символов (`char` — «узкие», `wchar_t` — «широкие»), например:

```
namespace std {  
    typedef basic_istream<char>      istream;           // узкий поток  
    typedef basic_istream<wchar_t> wistream;          // широкий поток  
}
```

Для совместимости с предыдущей (достандартной) реализацией библиотеки определены две специализации шаблона `basic_ios`:

```
namespace std {  
    typedef basic_ios<char>      ios;  
    typedef basic_ios<wchar_t> wios;  
}
```

Благодаря подобным определениям мы избавлены от необходимости писать длинные обозначения классов ввода-вывода.

Как мы знаем, определения нового класса обычно помещаются в отдельную единицу трансляции (см. главу 13). В этом случае может иногда возникнуть необходимость в опережающих объявлениях потоковых классов. Так как потоковые классы представляют собой шаблоны, то простое объявление, наподобие следующего, работать не будет:

```
class ostream;
```

В объектно-ориентированной библиотеке опережающие объявления всех потоковых классов собраны в специальный системный модуль (см. п. 27.2 в [1]), который подключается оператором

```
#include <iosfwd>
```

Использование модуля `<iosfwd>` вместо полных определений из модуля `<iostream>` существенно сокращает время трансляции для больших многомодульных программ.

Принципы организации потоков

С одной стороны, потоки можно разделить на входные и выходные: из входного потока программа данные читает (извлекает, вводит), в выходной поток — пишет (помещает, выводит). С другой стороны, потоки бывают форматируемые и неформатируемые, буферизуемые и небуферизуемые, широкие (*wide*) и узкие (*narrow*). Широкие потоки оперируют широкими символами `wchar_t` (см. п. п. C.2.2.1 в [1]), а узкие — обычными символами `char`.

Форматируемость означает, что при операциях ввода-вывода выполняется преобразование информации. Так как поток — это последовательность символов, при вводе, как правило, выполняется преобразование данных из символьного вида в двоичное представление, а при выводе — наоборот. Форматирование, например, всегда выполняется для стандартных потоков.

Буферизация позволяет еще больше «развязать» программы и устройства. Если поток буферизован, то вывод выполняется не на устройство, а в специально выделяемую область памяти — буфер. Заполненный буфер выводится на устройство, как правило, без непосредственного указания в программе. Однако программа может потребовать вывести неполный буфер. При вводе программа тоже имеет дело с буфером, а не с устройством, то есть данные в программу попадают из буфера. Буфер обычно заполняется при выполнении первой операции ввода, но специально программировать это не требуется. Обычно буферизация реализуется по умолчанию, но в библиотеках есть средства, позволяющие управлять назначением буферов. Схема ввода-вывода с буферизацией изображена на рис. 14.1.

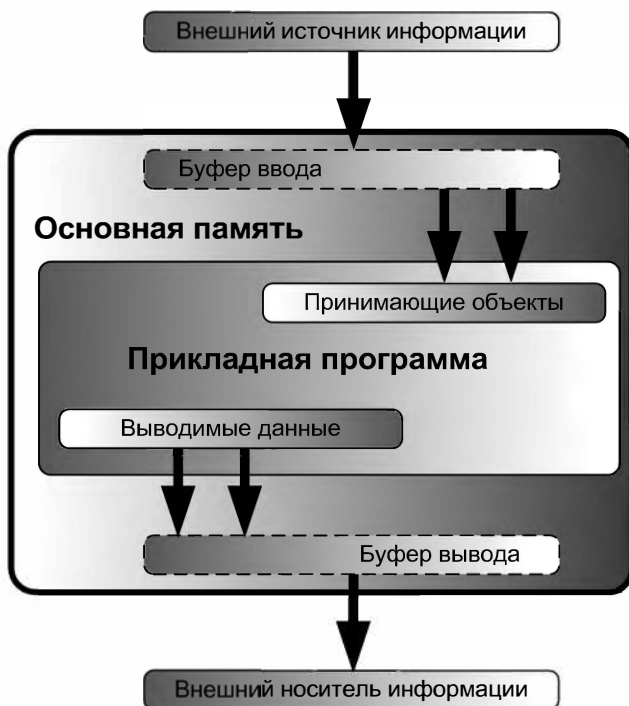


Рис. 14.1. Буферизация ввода-вывода

Библиотека предоставляет программисту ряд классов (см. листинг 14.1), которые реализуют три вида потоков: стандартные, строковые и файловые потоки. Стандартные потоки бывают только однонаправленными: либо информация только читается из потока, либо — только пишется в поток. А вот файловые потоки могут быть и двунаправленными: из потока можно вводить информацию и в тот же поток — выводить. И это естественно, так как файловые потоки обычно связываются с файлами на диске. Объектно-ориентированные строковые потоки позволяют работать и с символьными массивами, и со строками типа `string`. В отличие от потоков библиотеки `<stdio>`, эти потоки также могут быть двунаправленными.

Стандартные потоки обозначаются именами, определенными в стандарте, которые нельзя использовать в другом смысле. Эти имена «привязаны» к стандартным устройствам: клавиатуре и экрану, хотя есть возможности перенаправить стандартные потоки на другие устройства. Кроме того, стандартные потоки — форматируемые, среди них есть как буферизуемые, так и не буферизуемые.

Стандартные потоки

Стандартные потоки¹ нам уже знакомы. Для использования стандартных потоков достаточно включить в программу оператор

```
#include <iostream>
```

Еще раз напомним, что расширение указывать не нужно.

ВНИМАНИЕ

Не забудьте, что объектно-ориентированная библиотека ввода-вывода включена в стандартное пространство имен. Поэтому либо все имена библиотеки нужно писать с префиксом `std::`, либо использовать директиву `using`.

Заголовок `<iostream>` содержит описания классов ввода-вывода и 4 системных объекта (см. п. п. 27.3.1 в [1]), которые связаны со стандартными потоками `<cstdio>` (см. п. п. 27.3/1 в [1]):

- `cin` — объект класса `istream`, соответствующий стандартному вводу (`stdin`), по умолчанию связан с клавиатурой;
- `cout` — объект класса `ostream`, соответствующий стандартному выводу (`stdout`), по умолчанию связан с экраном;
- `clog` — объект класса `ostream`, соответствующий стандартному выводу для ошибок (`stderr`), по умолчанию связан с экраном;
- `cerr` — объект класса `ostream`, соответствующий стандартному выводу для ошибок (`stderr`), по умолчанию связан с экраном.

Имена `stdin`, `stdout`, `stderr` определены в библиотеке `<cstdio>`.

Вывод выполняется обычно с помощью перегруженной операции сдвига влево (оператор `<<`), а ввод — перегруженной операцией сдвига вправо (оператор `>>`), которые мы уже использовали для ввода-вывода денежных сумм (см. листинг 3.12).

Объект `cout` предназначен для «нормального» вывода, а объекты `cerr` и `clog` — для вывода сообщений об ошибках. Различаются они только буферизацией: вывод в `clog` буферизуется, а в `cerr` — нет [29]. Используются они точно так же, как и объект `cout`:

```
cerr << "Это сообщение об ошибке!" << endl;  
clog << "И это тоже сообщение об ошибке!" << endl;
```

«Ошибочные» объекты позволяют различать в тексте программы «нормальный» вывод и сообщения об ошибках.

¹ Речь идет об узких потоках. Широкие потоки будут рассмотрены позже.

Вывод элементарных типов и строк

Вывод значений встроенных типов (см. п. п. 27.6.2.5 в [1]) очень прост и осуществляется их записью в объект `cout`. При этом, естественно, выполняется преобразование значений (для несимвольных типов) из двоичного формата в последовательность символов. Разрешается выводить константы, переменные и выражения, например:

```
cout << 5; cout << '\n';
cout << 5.5; cout << '\n';
int i = 10;
cout << i; cout << '\n';
double d = 4.567e-12;
cout << d; cout << '\n';
float f = 7.123;
cout << f; cout << '\n';
char ch = 'w';
cout << ch; cout << '\n';
```

Никаких специальных символов форматирования не требуется — выводится именно так, как написано. Более того, разрешается выводить несколько значений в один объект `cout`, например:

```
cout << d << ' '; cout << i << '\n';
```

На экран выводится все подряд без пропусков:

```
4.567e-12;10
```

Однако при выводе по умолчанию существуют и ограничения:

- ❑ целые числа выводятся в десятичной системе счисления, например, шестнадцатеричная константа `0xFFFF` на экране будет показана как десятичное число `65535`;
- ❑ для дробного числа (независимо от типа — `float` или `double`) выводится только 6 значащих цифр, например, число `7.123456789` выведется как `7.12346`, а число `1234567.15` — как `1.23456e+006`;
- ❑ значения типа `bool` по умолчанию выводятся как целые, например, `true` выводится как `1`, а `false` — как `0`;
- ❑ значения перечислимого типа по умолчанию выводятся как целые.

По умолчанию символьные переменные выводятся как символы. Если требуется вывести символьную переменную как число, то нужно задать явное преобразование типа, например:

```
char n = -100;
cout << int(n) << '\n';
```

Одиночный символ можно вывести и методом `put()`, например:

```
char ch = 'a'; cout.put(ch);
```

Метод `put()` возвращает ссылку на поток, поэтому в этом случае вывести символ в виде целого невозможно. Еще один метод, с помощью которого можно вывести один символ, — это метод `write()` (см. п. п. 27.6.2.6 в [1]):

```
char ch = 'a'; cout.write(&ch,1);
```

Как видите, метод `write()` не слишком удобен для выполнения такой простой операции; обычно он используется для вывода в двоичные файлы.

Указатели по умолчанию выводятся в шестнадцатеричной системе счисления, независимо от типа. Исключение составляет указатель на символьную константу — по умолчанию выводится не указатель, а строка символов, на которую он указывает. Поэтому и в этом случае требуется преобразование типа, причем к типу `void *`:

```
int *pi = &i;
double *pd = &d;
char *s = "Константа символов\n";
cout << pi << ' ' << pd << '\n';
cout << s << ' ' << static_cast<void *>(s) << '\n';
```

При выводе выражений нужно быть внимательным. Дело в том, что в С и С++ операция `operator<<` изначально является операцией сдвига влево и имеет свой приоритет (см. табл. 3.1). Поэтому при выводе выражений, если не ставить скобок, порядок выполнения операций может оказаться неожиданным, например:

```
cout << d=f << '\n';
```

Этот оператор вызывает ошибку трансляции из-за того, что интерпретируется на основе приоритета операций так:

```
(cout << d)=(f << '\n');
```

Следующий оператор транслируется без сообщений об ошибках:

```
cout << (d<f)?d:f;
```

Однако работает он совершенно не так, как ожидается: на экран выводится не значение `d` или `f`, а 0 или 1 в зависимости от выполнения неравенства `(d<f)`. Поэтому возьмите себе за правило использовать скобки при выводе выражений, например:

```
cout << (d=f) << '\n';
cout << ((d<f)?d:f);
```

Такие операторы не вызывают протестов компилятора и работают совершенно правильно.

Вывод строк так же прост, как вывод числовых типов. Операция `operator<<` работает и с символьными массивам, и со строками типа `string`, например:

```
char *s1 = " Константа символов\n";
char s2[] = " Константа символов\n ";
string s3 = " Константа символов\n ";
cout << s1 << s2 << s3;
```

ПРИМЕЧАНИЕ

В Visual Studio.NET 2003 и C++ Builder 6 в стандартные узкие потоки правильно выводятся только английские символьные константы.

Массивы, в общем случае, нужно выводить поэлементно в цикле. Единственное исключение из этого правила — символьный массив. Для вывода всего символьного массива достаточно (как в приведенном примере) задать его имя.

Вывод строки символьного массива можно осуществить и методом `write()`, например:

```
cout.write(s1, strlen(s1));  
cout.write(s2, strlen(s2));
```

Первым параметром метода `write()` указывается адрес символьного массива, вторым — количество выводимых символов.

Так как обычно методы вывода возвращают ссылку на поток, можно объединять их в один оператор, например:

```
cout.write(s1, strlen(s1)).put('\n').put('\n');
```

При использовании метода `write()` надо быть очень аккуратным — метод не отслеживает нулевой байт в конце строки, поэтому можно очень легко совершить ошибку, задав неверное количество выводимых символов. Кроме того, метод не работает непосредственно с переменными типа `string`.

Однако мы все же можем «заставить» его правильно обрабатывать строки типа `string`, используя метод `c_str()` класса `string`, например:

```
cout.write(s3.c_str(), strlen(s3.c_str()));  
cout.write(s3.c_str(), s3.length());  
cout.write(s3.c_str(), s3.size());
```

Метод `c_str()` возвращает как раз адрес символьного массива, в котором записаны символы строки. Здесь показаны три разных способа задания длины строки — они эквивалентны с точки зрения результата.

Ввод элементарных типов

Ввод (см. п. п. 27.6.1.2 в [1]) осуществляется из входного объекта `cin` в переменные программы, например:

```
int i; cin >> i;
```

Ввод завершается нажатием клавиши `Enter`. Оператор чтения осуществляет разбор и преобразование последовательности вводимых символов в двоичное представление в соответствии с типом приемника. На клавиатуре мы можем набирать любые символы, однако ввод в переменную прекращается при наборе недопустимого для данного типа символа. Например:

```
\123.567
```

Если бы мы набрали это число на клавиатуре, то в переменную `i` попало бы значение `123`, так как точка не является допустимым символом для целого числа.

Однако в связи с тем, что ввод буферизуется, символы `.567` не пропадают и могут использоваться при следующем вводе в другую переменную. Буферизация при вводе, пока к ней не привыкнешь, несколько сбивает с толку. Например, пусть мы собираемся ввести значения для двух переменных:

```
int ii; double dd;  
cin >> ii; cin >> dd;
```

Для этого набираем на клавиатуре число 123.567. В результате в переменную `i` попадет значение 123, а в переменную `dd` — значение 0.567.

Ввод нескольких переменных тоже можно объединять в одном операторе, например:

```
int ii; double dd;  
cin >> ii >> dd;
```

В том и другом случаях набирать на клавиатуре значения можно любым из следующих двух способов:

- ввод числа 432, нажатие клавиши Enter, ввод числа 456.78, нажатие клавиши Enter;
- ввод числа 432, нажатие клавиши пробела, ввод числа 456.78, нажатие клавиши Enter.

Пробелов может быть несколько. Пробел (и табуляция) — это символ-разделитель. При вводе чисел ведущие символы-разделители пропускаются, а первый же пробел после допустимых символов для данного типа переменной является ограничителем значения.

Булевы значения должны вводиться как 1 (`true`) и 0 (`false`), любые другие символы воспринимаются как ошибочные.

Несмотря на то что вывод перечислимых типов реализован по умолчанию в виде целых (выполняется преобразование типа), ввод перечислимого типа как целого по умолчанию не работает. Это связано с тем, что в C++ перечислимый тип представляет собой новый тип данных, а для каждого нового типа требуется писать собственные операции ввода-вывода.

Значения символьных типов вводятся так же, как и значения других элементарных типов, например:

```
char c1, c2, c3;  
cin >> c1 >> c2 >> c3;
```

Ввод символа (даже единственного) должен заканчиваться нажатием клавиши Enter. При вводе символов операцией `operator>>` пробелы также пропускаются, поэтому последовательность символов, например «йцу», мы можем задать разными способами, например: поочередным нажатием клавиш `й`, `ц`, `у`, Enter, или перемежая нажатия этих клавиш пробелом: `й`, пробел, `ц`, пробел, `у`, Enter.

Пробелов, как обычно, может быть сколько угодно. Чтобы пробелы не пропускались, можно воспользоваться манипулятором `noskipws`, например:

```
cin >> noskipws >> c1 >> c2 >> c3;
```

В этом случае если мы нажмем клавиши `й`, пробел, `ц`, пробел, `у`, Enter, то в переменную `c1` попадет буква «й», в переменную `c2` — код пробела, а в переменную `c3` — буква «ц».

Манипуляторы (см. п. п. 27.4.5 в [1]) используются для форматирования ввода-вывода, и мы рассмотрим их позже.

Ввести в символьную переменную любой символ можно с помощью метода `get()` (см. п. п. 2.6.1.3 в [1]). Ввод символа реализуется любым из следующих операторов:

```
ch = cin.get();  
cin.get(ch);
```

И в этом случае ввод нужно завершить нажатием клавиши `Enter`.

Еще один метод, который можно использовать для ввода символа, — метод `read()` (см. п. п. 27.6.1.3 в [1]). Ввод выполняется так:

```
cin.read(&ch, 1);
```

Как видим, этот метод является «парой» методу `write()`, поэтому он не слишком удобен для ввода символа. Обычно этот метод применяется для ввода данных из двоичных файлов.

Хотя символьные типы могут использоваться в арифметических выражениях как целые и мы можем без проблем вывести переменную типа `char` в виде целого числа, ввести целое значение в символьную переменную нельзя. Даже если мы набираем на клавиатуре цифры, в символьную переменную всегда попадает код цифровой клавиши.

Ввод строк

Операция `operator>>` позволяет вводить и символьные массивы, и строки типа `string`, однако ввод выполняется только до первого символа-разделителя (обычно это пробел). Например, объявим в программе символьный массив:

```
char s[50];
```

Попробуем ввести в переменную `s` строку «привет от строки», выполнив операцию `cin >> s;`

В массив попадет только первое слово «привет». Точно так же операция работает и при вводе строки в переменную типа `string`.

ПРИМЕЧАНИЕ

Операции ввода-вывода для типа `string` реализованы в библиотеке `<string>`, а не в `<iostream>`. Все операции реализованы как внешние функции.

Для ввода строк с пробелами в символьный массив мы должны воспользоваться либо методом `get()`, либо методом `getline()` (см. п. п. 27.1.6.3 в [1]).

Метод `get()` позволяет ввести в символьный массив из n элементов не более $n - 1$ символов, следит за символом завершения ввода и проставляет после введенных символов нулевой байт. Метод перегружен и имеет несколько прототипов:

```
int_type get();  
istream& get(char& Ch);  
istream& get(char *str, streamsize count);  
istream& get(char *str, streamsize count, char delim);
```

Первые два вида метода `get()`, как мы уже знаем, применяются для ввода символов. Последние два — для ввода строк. Для ввода строки с пробелами в массив `s` нам достаточно написать в программе вызов:

```
cin.get(s, 50);
```

Завершается ввод, как всегда, нажатием клавиши `Enter`, после чего во входной поток попадает символ `'\n'`, а в массив символов на последнее место записывается нулевой байт. Чтобы можно было корректно продолжить ввод (следующей строки), символ `'\n'` надо удалить из входного потока. Это делается методом `ignore()` (см. п. п. 27.1.6.3 в [1]), который извлекает символы из потока, ничего не занося в переменные. Таким образом, ввод одной строки в массив `s` выполняется так:

```
cin.get(s,50); cin.ignore();
```

Символ `'\n'` является символом завершения ввода строки по умолчанию, однако мы можем задать любой символ завершения, например точку `('.'`):

```
cin.get(s,50,'.');
```

Правда, для окончания ввода все равно придется нажать клавишу `Enter`.

Метод `ignore()` на самом деле имеет два параметра:

- количество удаляемых из потока символов (по умолчанию — 1);
- символ-ограничитель, который тоже удаляется из потока (значение по умолчанию — конец файла);

Прототип метода:

```
istream& ignore(streamsize count = 1, int delim = EOF);
```

Поэтому мы можем пропустить вообще все символы потока. Наиболее часто метод используется для пропуска всех символов до конца строки:

```
cin.ignore(numeric_limits<std::streamsize>::max(), '\n');
```

ПРИМЕЧАНИЕ

Класс-шаблон `numeric_limits` (см. п. п. 18.2.1 в [1]) включает многообразные числовые пределы, зависящие от аппаратной платформы.

Как видим, метод `get()` не слишком удобен. Функция-метод `getline()` работает при вводе строк аналогично методу `get()`, но еще и удаляет из входного потока тот самый символ-завершитель строки (тоже символ `'\n'` по умолчанию), так что `ignore()` вызывать нет необходимости. Прототипы метода следующие:

```
istream& getline(char* str, streamsize count);  
istream& getline(char* str, streamsize count, char delim);
```

Наши примеры при использовании метода `getline()` выглядят так:

```
cin.getline(s,50);  
cin.getline(s,50,'.');
```

Последовательность символов, естественно, можно вводить и методом `read()`, но мы этого делать не будем, так как это не слишком удобно — метод никак не обрабатывает конец строки, поэтому ошибку совершить очень легко.

Кстати, посчитать количество реально введенных символов позволяет такой хороший метод, как `gcount()` (см. п. п. 27.1.6.3 в [1]):

```
const int lineSize = 250;
char Buf[lineSize];
// читается до конца строки, но не более 250 символов
cin.getline(Buf, lineSize);
int readSymbol = cin.gcount(); // сколько символов фактически прочитано
```

Методы `get()` и `getline()` входного потока не предназначены для ввода строк типа `string`, так как в качестве первого параметра принимают указатель на символ. В библиотеке `<string>` имеется собственная функция `getline()` (см. п. п. 21.3.7.9 в [1]), которая позволяет вводить строки с символами-разделителями. Вызывать функцию надо так:

```
string s;
getline(cin, s); // -- ввод из стандартного потока
```

В качестве первого параметра эта функция принимает поток. Очевидно, это сделано для большей общности, так как на месте первого параметра может стоять не только стандартный поток `cin`, а, например, файловый поток.

Эта функция (вернее, шаблон функции) перегружена и может иметь третий параметр, представляющий собой ограничитель строки — аналогично методу `getline()` из библиотеки `<iostream>`. Поэтому вводить строку можно и так:

```
string s;
getline(cin, s, '.'); // -- ввод из стандартного потока
```

Точка является завершающим символом строки (однако нажимать клавишу `Enter` все равно необходимо), она не включается в строку, и в переменную `s` попадут символы до точки. Например, если мы наберем на клавиатуре строку «привет от строки. продолжение строки.», в переменную `s` попадет строка «привет от строки».

Как видите, поведение этой функции совершенно аналогично поведению одноименного метода из библиотеки `<iostream>`.

ВНИМАНИЕ

Символы кириллицы, вводимые с клавиатуры в строковые переменные, затем правильно выводятся на экран в консольное окно — никаких специальных действий для перекодировки выполнять не требуется.

Состояния потока

Каждый поток (в том числе стандартный) в каждый момент времени находится в некотором состоянии. Эти состояния имеют названия `good`, `bad`, `fail` и `eof` (end-of-file — конец файла). Сами состояния определены в классе `ios_base`

как целые статические константы, называемые флагами состояния потока (см. п. п. 27.4.2.1.3 в [1]):

```
typedef int iostate;  
goodbit    = 0x00  
badbit     = 0x01  
eofbit     = 0x02  
failbit    = 0x04
```

Значения флагов зависят от реализации (показанные значения флагам присвоены в системе C++ Builder 6), однако имена определены в стандарте, поэтому именно так флаги состояния называются во всех без исключения системах. В программе имена флагов состояния нужно записывать с префиксом, например:

```
std::ios_base::eofbit
```

С таким же успехом можно использовать класс, производный от `ios_base`. В частности, в текстах программ стандартной библиотеки часто встречается более короткий префикс:

```
std::ios::eofbit
```

В базовом классе `ios_base` определено поле

```
iostate _M_iostate;           // библиотека STLport
```

ПРИМЕЧАНИЕ

Такое имя поле имеет в системе Borland C++ Builder 6. В другой реализации имя поля может быть другим.

В этом поле сохраняются флаги состояния во время работы программы. Это поле мы можем прочитать методом¹

```
iostate rdstate();
```

Установить любой флаг можно методом

```
void setstate(iostate flag);
```

Для установки нескольких флагов, естественно, нужно воспользоваться битовыми операциями, например:

```
setstate(std::ios::eofbit | std::ios::failbit);
```

Система ввода-вывода C++ предоставляет несколько методов, с помощью которых мы всегда можем узнать состояние потока:

```
bool good() const;    // следующая операция может выполняться  
bool eof() const;     // окончание ввода  
bool fail() const;    // следующая операция не выполняется  
bool bad() const;     // поток испорчен
```

Если после выполнения некоторой операции поток находится в состоянии `good`, это означает, что во время предыдущей операции не произошло никаких

¹ Все методы, работающие с битами состояния, описаны в стандарте (см. п. п. 27.4.4 в [1]).

непредвиденных событий и может быть выполнена следующая операция ввода-вывода. В остальных случаях следующая операция выполнена не будет.

Состояния `fail` и `bad` являются состояниями ошибки потока. Если поток находится в одном из этих состояний, то операции обмена не выполняются. Состояние `bad` включает в себя состояние `fail`: когда поток находится в состоянии `bad`, он находится и в состоянии `fail`; обратное неверно.

Если поток находится в состоянии `fail`, то операция ввода-вывода завершилась неудачно, однако поток не испорчен и никакие символы не потеряны. Обычно это состояние устанавливается при ошибках форматирования в процессе чтения, например, когда программа пытается прочитать целое число, а первый же символ (буква или знак операции) оказывается недопустимым. Поток из состояния `fail` мы можем вернуть в нормальное состояние с помощью метода `clear()`, например:

```
if (stream.fail()) stream.clear();
```

После этого можно выполнять операции обмена — опять до очередной ошибки. Метод `clear()` перегружен и позволяет не только сбрасывать, но и устанавливать нужные флаги состояния. Прототип этого метода:

```
void clear(iostate flag);
```

Состояние `bad` — это более тяжелое состояние. Флаг `badbit` указывает на неработоспособность потока данных или потерю данных. Когда поток оказывается в состоянии `bad`, ни в чем нельзя быть уверенным, поэтому в таких случаях лучше завершить выполнение программы.

Состояние `eof` может возникнуть только при операции чтения. Для стандартного потока ввода это состояние возникает при нажатии комбинации клавиш `Ctrl+Z`. Для файлов это состояние устанавливается при первой попытке чтения после последнего байта файла. При этом поток тоже переводится в состояние `fail`.

Состояние потока и логические условия

Состояние потока можно проверять непосредственно в условиях `if` и `while`. Например, ввод массива целых значений можно выполнять в цикле таким образом:

```
int aa[10] = {0};  
i = 0;  
while(cin >> aa[i++]);
```

Цикл завершается по одной из двух причин:

1. Создалась ситуация конца файла, что для входного потока эквивалентно нажатию комбинации клавиш `Ctrl+Z`.
2. В потоке встретился символ, не соответствующий типу вводимой переменной. В этом случае поток переводится в состояние `fail`.

Например допустим, мы поочередно нажали следующие клавиши: 1, пробел, 2, пробел, 3, пробел, 4, пробел, 5, e, 6, Enter. Тогда элементы массива `aa` будут иметь следующие значения:

```
aa[0] = 1;  
aa[1] = 2;
```

```
aa[2] = 3;  
aa[3] = 4;  
aa[4] = 5;  
aa[5] = 0;  
aa[6] = 0;  
aa[7] = 0;  
aa[8] = 0;  
aa[9] = 0;
```

Символ «е», введенный после цифры 5, не является допустимым для целого числа и поэтому ограничивает ввод.

В условиях можно задавать и явный вызов методов. Например, пусть мы хотим посчитать количество символов '\n', которым оканчиваются вводимые строки. Это можно сделать, используя метод `get()`:

```
int nl = 0;  
while(cin.get(ch))           // читать все символы, в том числе пробельные  
{ if(ch=='\n') nl++;  
  cout.put(ch);  
}
```

Так как «неправильных» символов не существует, цикл можно завершить только вводом комбинации клавиш `Ctrl+Z`. Можно использовать и другую форму метода `get()`:

```
int nl = 0;  
while(ch = cin.get() && ch != EOF)           // читать все символы  
{ if(ch=='\n') nl++;  
  cout.put(ch);  
}
```

Константа `EOF` помещается в переменную `ch` при нажатии комбинации `Ctrl+Z`. В библиотеке реализована логическая операция `operator!`, с помощью которой можно проверить аварийное состояние потока, например:

```
if (!(cin >> x))  
{    // ввод завершился неудачей  
    ...  
}
```

Обратите внимание: выражение ввода заключено в скобки. Это необходимо, так как в соответствии с приоритетом операций выражение без скобок интерпретируется так:

```
(!cin) >> x
```

Естественно, ввод и проверку состояния потока можно разделить, например:

```
cin >> x;  
if(!cin)  
{    // ввод завершился неудачей  
    ...  
}
```

Во всех приведенных примерах на месте `cin` может стоять любой поток.

При разделении ввода и проверки можно явно вызывать соответствующий метод. Например, таким способом мы можем проверять наступление ситуации конца файла для потока, связанного с файлом:

```
while (!from.eof())           // явная проверка признака конца файла
{ getline(from, s);           // чтение строки из потока from
  cout << s << endl;
}
```

Однако тут возможны сюрпризы. По умолчанию ограничителем строки является символ '\n', который записан в конце каждой строки. В данном случае при считывании последней строки состояния eof файла не возникает — она возникает только при попытке ввода за концом файла. Поэтому цикл выполнится лишний раз и на экране дважды появится последняя строка.

Правильная последовательность операторов может быть такой:

```
getline(from, s);              // чтение строки из потока from
while (!from.eof())           // явная проверка признака конца файла
{ cout << s << endl;
  getline(from, s);           // чтение строки из потока from
}
```

Другой вариант — бесконечный цикл с проверкой ситуации конца файла внутри цикла:

```
while (true)
{  getline(from, s);           // чтение строки из потока from
   if(from.eof()) break;      // явная проверка признака конца файла
   cout << s << endl;
}
```

Состояние потока и исключения

По умолчанию при ошибках ввода-вывода исключения не генерируются. Однако можно указать, при установке каких флагов состояния должно генерироваться исключение. В состав класса `basic_ios` (см. п. п. 27.4.4 в [1]) входит метод `exceptions()`, который и позволяет это сделать. Прототип метода:

```
void exceptions(iostate flags);
```

Указать на необходимость генерации исключения при установке флага `ios::badbit` можно так:

```
stream.exceptions(ios::badbit);
```

Флаги, как обычно, можно комбинировать, например:

```
stream.exceptions(ios::badbit|ios::failbit);
```

Исключения будут генерироваться не только во время операций ввода-вывода, но и при установке флагов методами `clear()` и `setstate()`.

Если аргумент равен 0 или `ios::goodbit`, исключения генерироваться не будут.

Вызов без аргумента возвращает текущие флаги, при установке которых генерируется исключение, например:

```
ios_base::iostate flags = stream.exceptions();
```

Если возвращается флаг `goodbit`, исключения не генерируются.

Генерируемые исключения являются объектами класса `ios_base::failure` (см. п. п. 27.4.2.1.1 в [1]), который является наследником класса `exception` (см. главу 7). Класс `ios_base::failure` выглядит следующим образом:

```
class ios_base::failure : public exception
{ public:
    explicit failure(const string& msg);
    virtual ?failure();
    virtual const char* what() const throw();
};
```

Как видите, сложного ничего нет.

Работа с файлами

Стандартные потоки «привязаны» к клавиатуре (`cin`) и экрану (`cout`, `cerr`, `clog`) и функционируют из командной строки. В настоящее время, когда оконный интерфейс стал повсеместным, стандартные потоки для ввода-вывода информации практически не применяются. Значительно чаще информация организуется в виде файлов, которые тем или иным способом затем обрабатываются программой.

Концепция файлов родилась задолго до концепции потоков. Файл — это поименованная порция информации, размещенная на внешнем носителе. В настоящее время файлы размещаются практически исключительно на магнитных дисках, хотя так было не всегда (до распространения магнитных дисков основным носителем информации являлись магнитные ленты). Для управления множеством файлов в состав операционной системы входит *файловая система*. Именно файловая система [55, 56, 57, 58] определяет, каким образом именуются файлы и где они размещаются. В разных операционных системах — разные правила именования файлов. Даже в разных версиях одной операционной системы эти правила бывают различными.

Поскольку мы работаем в Windows, надо разобраться, как именуются файлы в этой операционной системе и как представляются имена файлов в программе на C++.

В программе на C++ имя файла обычно представляется либо константой-строкой, либо переменной — символьным массивом, в который помещается строка-имя файла. Тип `string` использовать нельзя! Имя файла должно удовлетворять следующим ограничениям:

- ❑ нельзя использовать символы

> < " | .

- ❑ имена могут содержать пробелы и символы кириллицы;

- имена нечувствительны к регистру, однако регистр в именах сохраняется (то есть если файл был создан с именем `file`, то в таком виде имя и будет выводиться, но в то же время к файлу можно обращаться и по имени `FILE`, и по имени `File`).

Имя файла состоит из собственно имени и расширения, которое может и отсутствовать. Количество символов в расширении не регламентируется, но по традиции, оставшейся от MS-DOS, обычно задают от одного до трех символов. Если расширение есть, оно отделяется от имени точкой. Расширение часто (но не всегда) определяет тип файла.

Полное имя файла включает еще путь к файлу. Путь — это идентификатор диска и имя каталога, где размещен файл. Если ни диск, ни каталог не заданы, то по умолчанию обычно принимается тот каталог, откуда была запущена программа. Для получения/изменения текущего диска/каталога операционная система (не C++) предоставляет ряд API-функций. Длина полного имени ограничена Windows-константой `MAX_PATH`.

ПРИМЕЧАНИЕ

В Borland C++ Builder 6 и Visual C++.NET 2003 определена константа `FILENAME_MAX`, равная 260. Эта константа находится в файле `stdio.h`.

Диски обозначаются латинской буквой A–Z (или a–z) с двоеточием. Идентификатор A (и B тоже!) обычно обозначает дискету. Вложенные каталоги в имени отделяются друг от друга символом-разделителем. По традиции в Windows разделителем является символ обратной косой черты (`\`). В программе на C++ этот символ необходимо писать дважды (`\\`), так он является специальным символом. Однако допускается использовать и обычную косую черту (`/`). Разделитель должен отделять диск от имени каталога. Примеры имен:

```
"question.txt"
"c:\\test\\question.doc"
"a:/number.bin"
```

Правила работы с файлами тоже были разработаны в «доисторические» времена, однако сохранились до настоящего времени практически в неизменном виде:

1. Файл открывается.
2. Выполняются операции обмена между программой и файлом.
3. Файл закрывается.

Файл может быть открыт для чтения и (или) записи. В зависимости от режима открытия, информация либо читается из файла, либо пишется в файл. В первом случае файл называется *входным*, во втором — *выходным*. При извлечении информации из входного файла программа должна уметь определять, когда данные закончились.

Программа на C++ имеет дело не с файлом, а с абстрактным потоком, который представлен в программе обычной переменной. Эта переменная имеет область

видимости и время жизни в соответствии с объявлением и не имеет никакого отношения к файлам на диске. Таким образом, чтобы иметь возможность работать с файлом, необходимо как-то связать переменную-поток с этим файлом. Обычно эта связь задается либо при открытии файла, либо при создании потока. При закрытии файла связь разрывается. Подчеркнем, что разрывается только связь переменной-потока с внешним файлом на диске, сама переменная продолжает «жить» в соответствии с объявлением, и ее снова можно связать с тем же или с другим файлом.

ПРИМЕЧАНИЕ

В «доисторические» времена операции открытия и закрытия выполнялись для файлов — потоков тогда еще не придумали. Поскольку в программе мы работаем с потоковой переменной, то сейчас мы можем говорить об открытии (и закрытии) как файлов, так и потоков. Мы будем употреблять и то, и другое выражение.

Так как потоки обычно буферизуются, то при закрытии файла буфер освобождается — выводится в файл на внешнее устройство. Если программа заканчивается аварийно, файлы остаются незакрытыми, и последняя порция информации на диск не попадает. Такие файлы при просмотре каталога обычно показываются с нулевой длиной.

Обычно различают *текстовые* и *двоичные* файлы. Текстовые файлы состоят из строк, которые завершаются символом конца строки. В программе на C++ этот символ обозначается как `'\n'`.

ПРИМЕЧАНИЕ

В системе Windows строки в текстовом файле завершаются комбинацией двух байтов 0x0D0A, поэтому при операциях ввода-вывода система выполняет преобразование.

Обычно операции обмена с текстовым файлом сопровождаются преобразованием информации аналогично тому, как это происходит для стандартных потоков. По нашей классификации, приведенной в начале главы, текстовые файлы являются форматируемыми. Форматирование не выполняется только в том случае, если содержимое текстового файла обрабатывается именно как символы и строки.

Двоичные файлы не разбиваются на строки, и никаких преобразований при обмене не выполняется — двоичные файлы не являются форматируемыми. Это, во-первых, означает, что операции обмена для двоичных файлов выполняются быстрее. Во-вторых, при операции записи в двоичный файл попадает ровно столько байтов, сколько записываемый объект занимает в памяти. Например, целое число, записанное в двоичный файл, займет на диске `sizeof(int)` байтов. Это существенно отличается от записи в текстовый файл, где количество записываемых по умолчанию символов зависит от величины числа. Например, число 12 в текстовом файле займет 2 или 3 байта (в зависимости от того, выводится ли число со знаком или без него), а 123 456 — 6 или 7 байт. Примером двоичного файла является исполняемый файл (с расширением `exe`).

Файлы, естественно, связываются с потоками (см. п. п. 27.8.1 в [1]). Для использования файловых потоков необходимо задать в программе оператор

```
#include <fstream>
```

Объекты типа `fstream` связываются с файлами, которые можно читать и в которые можно записывать информацию. Так как класс `basic_fstream` (см. п. п. 27.8.1.11 в [1]) является наследником от `basic_istream` (см. листинг 14.1), все операции ввода-вывода, описанные ранее для стандартных потоков, работают и с файловыми потоками.

Если нам нужны только входные (см. п. п. 27.8.1.5 в [1]) потоки-файлы, достаточно использовать оператор

```
#include <ifstream>
```

Для работы с выходными (см. п. п. 27.8.1.8 в [1]) файловыми потоками можно задать оператор

```
#include <ofstream>
```

Впрочем, заголовок `<fstream>` объединяет то и другое, поэтому программисты редко используют последние два варианта.

ПРИМЕЧАНИЕ

В некоторых реализациях заголовок `<fstream>` включает объявление `<iostream>`, поэтому при использовании первого можно не прописывать в программе второго. Однако в системе Visual C++.NET 2003 требуется подключать обе библиотеки, если используются оба вида потоков.

Текстовые файлы

Для того чтобы открыть требуемый файл, нужно просто объявить объект соответствующего типа, указав параметры конструктора, который берет на себя всю работу. По умолчанию файл считается текстовым. Закрывать файл тоже не требуется, так как при уничтожении объекта всю работу выполнит деструктор. Напишем простой пример создания и чтения текстового файла `oonumber.txt` (листинг 14.2). Нам нужно предварительно создать на диске C папку `TextFiles` — все текстовые файлы будем записывать в нее.

Листинг 14.2. Создание текстового файла

```
#include <fstream>
#include <iostream>           // требуется в Visual C++.NET 2003
#include <ctime>
using namespace std;
int main()
{
    srand((unsigned)time(NULL)); // инициализация датчика случайных чисел
    /* открываем выходной текстовый файл для записи */
    {
        ofstream strm("c:/textfiles/oonumber.txt");
        for(int i = 0; i < 10; i++) // выводим 10 чисел
            strm << rand()%10 << '\n';
    }
}
```

```

} // деструктор закрывает файл
/* открываем тот же текстовый файл для чтения */
{ ifstream strm("c:/textfiles/oonumber.txt");
  char ss[20] = {0};
  while(true)
  { strm.getline(ss, 20); // читаем числа как строки
    if (strm.eof()) break; // проверяем признак конца файла
    cout << ss << '\n'; // выводим на экран
  }
} // опять деструктор закрывает файл
return EXIT_SUCCESS;
}

```

Обратите внимание на внутренние блоки в теле функции `main()`. Внутри блоков объявлены объекты-потоки: в первом — выходной, во втором — входной. Больше ничего не требуется, так как объявление объекта одновременно и открывает файл, и связывает поток с файлом, а при выходе из блока деструктор закрывает файл и разрывает связь. Мы не только не задаем явных операций открытия и закрытия, но еще и не указываем режима открытия файла — все это реализовано в конструкторе соответствующего класса по умолчанию.

Напишем ту же программу в более традиционном виде — с открытием и закрытием, с проверкой ошибок открытия. Вместо простого вывода на экран содержимого файла просуммируем числа, записанные в файл (листинг 14.3).

Листинг 14.3. Создание файла и суммирование чисел

```

#include <fstream>
#include <iostream> // требуется в Visual C++.NET 2003
#include <ctime>
using namespace std;
int main()
{
    srand((unsigned)time(NULL)); // датчик случайных чисел
    ofstream strm; // выходной поток-объект
    strm.open("c:/textfiles/oonumber.txt"); // открываем
    if (strm.is_open()) // проверка открытия
    { for(int i = 0; i < 10; i++) // выводим 10 чисел
        strm << rand()%10 << '\n';
        strm.close(); // закрываем выходной поток-файл
        // суммирование чисел, записанных в файле
        // открываем тот же текстовый файл для чтения
        ifstream strm("c:/textfiles/oonumber.txt");
        if (strm) // проверка открытия
        { int number, summa = 0, count = 0;
            while(strm >> number) // ввод числа
            { ++count; // подсчет количества
              summa+=number; // суммирование
            }
            cout << summa << "; " << count; // вывод результатов
            strm.close(); // закрываем поток-файл
        }
    }
    return EXIT_SUCCESS;
}

```

Так же легко написать программу копирования файлов¹ (листинг 14.4). Если не проверять ошибки, то программа получается очень простой!

Листинг 14.4. Копирование файлов

```
#include <fstream>
#include <ctime>
using namespace std;
// функция копирования потока in в поток out; потоки должны быть открыты
void filecopy(istream &in, ostream &out)
{ char ch;
  while(in.get(ch))          // читать все символы, в том числе пробельные
    out.put(ch);
}

int main()
{   ifstream instrm("c:/textfiles/oonumber.txt");
    ofstream outstrm("c:/textfiles/oonumber.new");
    if (instrm) filecopy(instrm, outstrm); // копирование файлов
    return EXIT_SUCCESS;
}
```

Заглянув в каталог TextFiles, мы обнаружим там два файла: исходный oonumber.txt и новый oonumber.new. Просмотрев оба файла в Блокноте (Notepad), можно убедиться в их идентичности.

Необходимо обратить внимание на функцию копирования: параметры должны передаваться по ссылке, так как в процессе операций чтения и записи состояние потоков изменяется. Никакой другой способ передачи параметров не работает.

Функция копирования может быть и такой:

```
void filecopy(istream &in, ostream &out)
{ char ch;
  while(in.read(&ch, 1))    // читать все символы, в том числе пробельные
    out.write(&ch, 1);
}
```

И еще один вариант функции копирования:

```
void filecopy(istream &in, ostream &out)
{ char ch = in.get();
  while(!in.eof())
  { out.put(ch);
    ch = in.get();          // читать все символы, в том числе пробельные
  }
}
```

Пример обработки текстовых файлов

В качестве примера обработки текстовых файлов напомним программу, которая реализует наиболее часто используемую функцию препроцессора — подключение файлов с помощью оператора `#include`. Ограничим синтаксис оператора:

□ оператор начинается с первой позиции строки;

¹ Другой, значительно более эффективный способ копирования файлов мы рассмотрим позже.

- подключаемый файл всегда задается в формате «имя_файла»;
- имя файла отделяется от оператора `#include` только одним пробелом.

Таким образом, непосредственно имя файла начинается в операторе всегда с 11-й позиции и продолжается до символа-кавычки. Подключаемый файл тоже может содержать операторы `#include`. Ограничение вложенности, очевидно, задается глобальной константой `FOPEN_MAX`, определяющей максимальное количество одновременно открытых файлов и определенной в заголовке `<stdio.h>`.

Итак, в результате получаем:

```
#include "имя_файла"
```

При подключении этот оператор записывается в результирующий файл как комментарий:

```
//-----#include имя_файла-----
```

После этой строки в результирующий файл записывается содержимое указанного файла. Если подключаемый файл не был открыт, то в результирующий файл записывается строка-комментарий:

```
//-----#include имя_файла -- Error! -- File not open!
```

Подключаемые файлы могут, вообще говоря, располагаться в различных каталогах, но мы для проверки работы программы соберем их все в нашем каталоге `TextFiles`. Файл, который надо обрабатывать первым, задается в командной строке. Если имя задано неверно, то программа завершает работу. Результирующий файл с именем `result.files` размещается в том же каталоге `TextFiles`. Текст программы представлен в листинге 14.5.

Листинг 14.5. Обработка оператора `#include`

```
string NameFiles(const char *namefile)
{ string path = "c:/textfiles/";
  return path+namefile;
}

bool isInclude(const string line)
{ if (line.substr(0,8) == "#include") return true;
  else return false;
}

void includeFile(const char *namefile, ofstream &result)
{ static int countfiles = 0;
  ++countfiles;
  if (countfiles > FOPEN_MAX) // файлов больше положенного?
  { cerr << "Too mani files is opened!" << endl;
    abort();
  }
  string line; // читаемая строка
  const string tenminus = "-----";
  const string include = "#include ";
  string comment;
  string name = NameFiles(namefile); // полное имя файла
  ifstream in; // очередной входной файл
```

продолжение ➤

Листинг 14.5 (продолжение)

```

    in.open(name.c_str());           // открыли
    if(in.is_open())                // если открылся
    {
        getline(in, line);
        while(!in.eof())
        { if (isInclude(line))      // если #include
            {
                char file[FILENAME_MAX];
                int i = 0; // извлекаем имя файла из #include
                while(line[i+10]!='') { file[i]=line[i+10]; i++; }
                file[i] = 0;        // завершающий ноль
                comment = "/" + tenminus + include + file + tenminus;
                result << comment << endl;
                includeFile(file, result); // пошли внутрь
                result << "///-end-----\n";
            }
            else result << line << endl;
            getline(in, line);
        }
        in.close();
    }
    else // формируем ошибочный комментарий
    { const string Error = "Error! -- File not open!";
      comment = "/" + tenminus + include + namefile + " -- " + Error + '\n';
      result << comment;
    }
}

int main(int argc, char *argv[])
{
    ofstream result("c:/textfiles/result.files");
    if(!result.is_open())           // ошибка при открытии
    {
        cerr << "Error result file!" << endl;
        return 1;
    }
    includeFile(argv[1], result);
    result.close();
    return 0;
}

```

Главная программа просто открывает результирующий файл и вызывает рекурсивную функцию обработки, передавая ей в качестве параметра имя файла из командной строки и поток-результат. Ошибки практически не обрабатываются, чтобы не отвлекаться от основной задачи — обработки файлов.

Основную работу выполняет рекурсивная функция `includeFile()`, аргументом которой является имя обрабатываемого файла и поток-результат. Функция в начале считает количество открытых файлов — можно запрограммировать здесь более серьезную обработку ошибки (например, генерировать исключение). Если все в порядке, то формируется полное имя файла — работает вспомогательная функция `NameFiles()`. Далее файл открывается, и в цикле читаются строки файла. Если прочитана обычная строка, то она просто выводится в результирующий файл. Если же строка содержит оператор `#include` (что определяет простая функция-предикат `isInclude()`), то формируется и выводится строка-комментарий о включаемом файле и выполняется рекурсивный вызов. По окончании

файла выводится дополнительная строка минусов со словом «end». Таким образом, включенный файл оказывается обрамленным двумя строками-комментариями. Если же при открытии файла возникли проблемы, то формируется строка-комментарий с ошибкой и функция завершает обработку текущего файла. Создадим в каталоге TextFiles три небольших текстовых файла, a.txt, b.txt и d.txt:

□ файл a.txt:

```
aaaaaaaaaaaaaaaa
#include "b.txt"
aaaaaaaaaaaaaaaa
```

□ файл b.txt:

```
bbbbbbbbbbbbbbbb
#include "d.txt"
bbbbbbbbbbbbbbbb
#include "dd.txt"
bbbbbbbbbbbbbbbb
```

□ файл d.txt:

```
dddddddddddddddd
dddddddddddddddd
```

Файл a.txt подключает файл b.txt, который, в свою очередь, подключает файл d.txt и ошибочный файл dd.txt. Результат работы нашей программы получается следующий:

```
aaaaaaaaaaaaaaaa
//-----#include b.txt-----
bbbbbbbbbbbbbbbb
//-----#include d.txt-----
dddddddddddddddd
dddddddddddddddd
//-----end-----
bbbbbbbbbbbbbbbb
//-----#include dd.txt-----
//-----#include dd.txt -- Error! -- File not open!
//-----end-----
bbbbbbbbbbbbbbbb
//-----end-----
aaaaaaaaaaaaaaaa
```

Как видим, вложенные файлы обрабатываются совершенно правильно.

Собственно, если бы не нужно было формировать комментарий вокруг вложенного файла, функция `includeFile()` была бы значительно короче (листинг 14.6).

Листинг 14.6. Функция `includeFile()`

```
void includeFile(const char *namefile, ofstream &result)
{
    static int countfiles = 0;
    ++countfiles;           // проверяем countfiles > FOPEN_MAX
    if (countfiles > FOPEN_MAX)
    { cerr << "Too mani files is opened!" << endl; abort(); }
}
```

продолжение ➤

Листинг 14.6 (продолжение)

```

string line;
string name = NameFiles(namefile);
ifstream in;
in.open(name.c_str());
if(in.is_open())
{
    getline(in, line);
    while(!in.eof())
    { if (isInclude(line)) includeFile(file, result);
      else result << line << endl;
      getline(in, line);
    }
    in.close();
}
else result << "Error! -- File not open!" << endl;
}

```

Этот вариант функции вполне можно использовать в «настоящем» препроцессоре.

Режимы открытия потоков (файлов)

Объектно-ориентированные файловые потоки можно открывать в различных режимах (см. п. п. 27.4.2.1.4 в [1]). Режим открытия задается в виде целой статической константы размером в один бит (как и состояния потока, режимы открытия часто называют флагами). Режимы открытия описаны в табл. 14.1.

Таблица 14.1. Режимы открытия потоков

Режим	Описание
in	Открытие потока для чтения (по умолчанию для ifstream)
out	Открытие потока для записи (по умолчанию для ofstream)
trunc	Удаление прежнего содержимого файла (по умолчанию для ofstream)
app	Открытие потока для записи в конец файла
ate	Открытие потока для чтения и (или) записи и позиционирование в конец файла
binary	Открытие потока в двоичном режиме (по умолчанию режим текстовый)

ПРИМЕЧАНИЕ

В некоторых системах (в частности, в Visual C++.NET 2003) имеются дополнительные режимы вроде `poscreate` или `noreplace`, однако они не являются стандартными и поэтому не рассматриваются.

Поскольку режимы представляют собой биты, то их можно объединять битовой операцией `operator|`, например:

```
std::ios::in|std::ios::binary
```

Еще пример:

```
std::ios::in|std::ios::out|std::ios::ate
```

В конструкторах и методах `open()` присутствует второй параметр, определяющий режим открытия потока. Открытие файлового потока для чтения/записи не зависит от типа объекта, объявляемого в программе — этот тип определяет только режим открытия по умолчанию при отсутствии второго аргумента. Это означает, что поток типа `istream` вполне можно открывать и для записи тоже, а поток `ostream` — для чтения. Однако операции, разрешенные для объявленного объекта, определяются классом потока данных. Допустимые комбинации флагов открытия приведены в табл. 14.2.

Таблица 14.2. Допустимые комбинации флагов открытия

Комбинация флагов	Описание
<code>in</code>	Чтение (файл должен существовать)
<code>out</code>	Стирание и запись (файл создается, если его нет)
<code>out trunc</code>	Стирание и запись (файл создается, если его нет)
<code>app</code>	Дозапись (файл создается, если его нет)
<code>out app</code>	Дозапись (файл создается, если его нет)
<code>in out</code>	Чтение и запись (файл должен существовать)
<code>in out trunc</code>	Стирание, чтение и запись (файл создается, если его нет)

Комбинации флагов, не указанные в таблице, запрещены. Однако в любую из указанных комбинаций можно добавить флаги `binary` и (или) `ate`, например:

```
std::ios::in|std::ios::out|std::ios::binary|std::ios::ate
```

Такое сочетание флагов означает открытие файлового потока в двоичном режиме для чтения/записи с первоначальным позиционированием в конец файла.

Для того чтобы добавить к существующему файлу `oonumber.new` существующий же файл `oonumber.txt`, достаточно в программе копирования файлов (см. листинг 14.4) изменить всего одну строчку — объявление выходного файлового потока:

```
ofstream outstrm("c:/textfiles/oonumber.new", std::ios::app);
```

Все остальное остается без изменений. После работы можно в Блокноте (Notepad) открыть файл `oonumber.new` и убедиться, что он содержит две одинаковые последовательности чисел, записанные в файле `oonumber.txt`.

Двоичные файлы

Вывод в двоичный файл обычно выполняется методом `write()`, который мы уже рассматривали, изучая вывод символов и строк. Все же обычно с его помощью выводят как раз не символы, а данные других типов. Метод имеет прототип

```
ostream& write(const char *str, streamsize count);
```

Метод записывает `count` символов символьного массива `str` в поток данных. Тип `streamsize` (см. п. п. 27.4.1 в [1]) обычно представляет собой знаковую версию

`size_t`. Никакие символы-ограничители не влияют на вывод. Метод возвращает ссылку на поток, поэтому после операции можно проверить состояние потока. Путем преобразования указателей можно вывести в выходной двоичный поток значение переменной любого типа, например:

```
int i = 5;
to.write((char *)&i, sizeof(i));
TMoney d(200.56);
to.write(reinterpret_cast<char *>(&d), sizeof(TMoney));
```

Можно вывести и массив, например:

```
long t[10];
to.write((char *)&t[0], sizeof(t));
```

Ввод из двоичных файловых потоков делается методом `read()`, который имеет такой же прототип:

```
istream& read(char *str, streamsize count);
```

Метод читает `count` символов в символьный массив `str`. Размер символьного массива должен быть достаточным, чтобы вместить `count` символов. Метод возвращает ссылку на поток, поэтому после операции можно проверить состояние потока. Никакие символы-разделители и символы-завершители не влияют на ввод. Если обнаружен конец файла, то устанавливаются флаги `eofbit` и `failbit`.

Путем преобразования указателей можно ввести из входного двоичного потока значение переменной любого типа, например:

```
int ii; TMoney dd;
from.read((char *)&ii, sizeof(ii));
from.read(reinterpret_cast<char *>(&dd), sizeof(TMoney));
long t[10];
from.read((char *)&t[0], sizeof(t));
```

Существует еще один метод ввода, имеющий следующий прототип (см. п. п. 27.6.1.3 в [1]):

```
streamsize readsome(char *str, streamsize count);
```

Метод работает аналогично методу `read()`, но возвращает не ссылку на поток, а количество введенных символов.

Теперь легко написать программы обработки двоичных файлов. В первом примере мы создаем два двоичных файла из одного массива (листинг 14.7). Для сохранения двоичных файлов нужно создать на диске C каталог `BinFiles`.

Листинг 14.7. Создание и обработка двоичных файлов

```
#include <fstream>
#include <iostream>
#include <ctime>
using namespace std;
int main()
{   int m[10]={0};
    srand((unsigned)time(NULL)); // инициализация датчика случайных чисел
```

```
/* заполняем массив m числами */
for(int i = 0; i < 10; i++)
{ m[i] = rand()%10;
  cout << m[i] << ' '; // контрольный вывод на экран
}
cout << '\n';
/* открываем файл для записи */
ofstream outstrm ("c:/binfiles/oonumber1.bin", std::ios::binary);
if(outstrm.is_open())
{ for(int i = 0; i < 10; i++) // выводим массив в файл поэлементно
  outstrm.write((char *)&m[i], sizeof(int));
  outstrm.close();
}
/* открываем другой файл для записи */
outstrm.open("c:/binfiles/oonumber2.bin", std::ios::binary);
if(outstrm.is_open())
{ outstrm.write((char*)m, sizeof(m)); // выводим массив в файл
  outstrm.close();
}
// вывод двоичного файла на экран
// открываем второй файл для чтения
{ ifstream instrm ("c:/binfiles/oonumber2.bin", std::ios::binary);
  int a = 0;
  // читаем числа по одному из файла и выводим
  while(instrm.read((char *)&a, sizeof(int)))
    cout << a << ' ';
  cout << '\n';
}
// открываем первый файл для чтения
ifstream instrm ("c:/binfiles/oonumber1.bin", std::ios::binary);
int t[10] = {0};
instrm.read((char *)t, sizeof(t)); // чтение файла в массив
instrm.close(); // закрываем
for(int i = 0; i < 10; i++)
  cout << t[i] << ' ';
cout << '\n';
char ch = getchar();
return 0;
}
```

В этом примере два двоичных файла из одного массива создаются разными способами: в файл `oonumber1.bin` массив выводится поэлементно, а в файл `oonumber2.bin` — сразу целиком одним оператором. Если мы заглянем в каталог `BinFiles`, то увидим, что эти два файла имеют одинаковый размер в 40 байт.

Затем те же файлы открываются как входные, читаются и выводятся на экран. Сначала открывается файл `oonumber2.bin` (в который мы писали массив целиком), и чтение из него выполняется по одному числу. Нетрудно вместо вывода на экран выполнять в цикле, например, суммирование чисел, записанных в этот файл.

Первый файл `oonumber1.bin`, который записывался в цикле по одному числу, читается сразу целиком в массив `t` одним оператором, и поток тут же закрывается. И снова мы наблюдаем на экране, что чтение выполнено совершенно правильно. Такое «смешение» для двоичных файлов безопасно, так как и в памяти, и на диске

размеры данных равны `sizeof(тип) * n`, где n — количество элементов, участвующих в обмене.

Копирование и дозапись двоичных файлов можно выполнить той же функцией `filecopy()` (см. листинг 14.4), открыв потоки как двоичные, например:

```
// копирование файлов
{ ifstream instrm ("c:/binfiles/oonumber1.bin", std::ios::binary);
  ofstream outstrm("c:/binfiles/oonumber.new", std::ios::binary);
  if (instrm) filecopy(instrm, outstrm);
}
// дозапись нового файла в конец старого
{ ifstream instrm ("c:/binfiles/oonumber2.bin", std::ios::binary);
  ofstream outstrm("c:/binfiles/oonumber.new",
                  std::ios::app|std::ios::binary);
  if (instrm) filecopy(instrm, outstrm);
}
```

Как и при обработке текстовых файлов, разница заключается только в режиме открытия.

Напишем программу сравнения двоичных файлов, чтобы убедиться, что файл `oonumber2.bin` совпадает с исходным файлом `oonumber1.bin`. Реализуем эту операцию в виде функции сравнения `filecompare()`:

```
bool filecompare(ifstream &first, ifstream &second);
```

Параметрами этой функции являются открытые потоки (как в функции `filecopy()`), а результат — булево значение, равное:

- ☐ `true`, если файлы идентичны;
- ☐ `false`, если файлы различаются.

Нам достаточно сравнить файлы побайтно, выдав значение `false` при первом различии:

```
// функция сравнения файлов побайтно
bool filecompare(ifstream &first, ifstream &second)
{ char ch1 = 0, ch2;
  while (ch1 != EOF)
  { ch1 = first.get(); ch2 = second.get();
    if (ch1!=ch2) return false;
  }
  return true;
}
// -----в главной программе...
ifstream first ("c:/binfiles/oonumber1.bin", std::ios::binary);
ifstream second("c:/binfiles/oonumber2.bin", std::ios::binary);
if(filecompare(first, second)) cout << "Files are equals!"<< endl;
else cout << "Files are not equals!" << endl;
```

Если файлы совпадают, программа выводит на экран строку

```
Files are equals!
```

Между прочим, эта же функция сравнения прекрасно работает с любыми текстовыми файлами, поскольку ввод информации осуществляется посимвольно.

Ввод-вывод объектов в двоичный файл

До сих пор мы выводили в двоичные файлы объекты только встроенных типов. Теперь настала пора разобраться с проблемами, возникающими в двоичных файлах при вводе-выводе объектов произвольного типа. Как мы уже знаем (см. главу 1), размеры классов и структур в памяти зависят от выравнивания. Рассмотрим тот же вопрос с точки зрения записи в двоичный файл. Напишем простую программу (листинг 14.8), в которой просто выведем в двоичный файл несколько структур в двух режимах: с выравниванием по байту и без.

Листинг 14.8. Размеры объектов на диске

```
#include <fstream>
#include <iostream>
using namespace std;
// #pragma pack(1) // выравнивание по байту
int main()
{
    ofstream f;
    class Empty {}; // пустой класс
    Empty t;
    f.open("c:/binfiles/binempty.bin", ios::binary);
    if(!f.is_open()) abort();
    f.write((char *)&t, sizeof(t)); // пишем пустой класс
    f.close();
    struct BinNotPack1 // неупакованная структура
    { double a; char ch; int b; };
    BinNotPack1 A = { 1.0, 'a', 1 };
    cout << sizeof(A) << endl; // размер = 16
    f.open("c:/binfiles/binA.bin", ios::binary);
    if(!f.is_open()) abort();
    f.write((char *)&A, sizeof(A)); // пишем структуру A
    f.close();
    struct BinNotPack2 // неупакованная структура
    { int b; double a; char ch; };
    BinNotPack2 B = { 1, 1.0, 'a' };
    cout << sizeof(B) << endl; // размер = 24
    f.open("c:/binfiles/binB.bin", ios::binary);
    if(!f.is_open()) abort();
    f.write((char *)&B, sizeof(B)); // пишем структуру B
    f.close();
    return 0;
}
```

Эта простая программа записывает в наш каталог BinFiles три файла: binempty.bin, binA.bin и binB.bin. В первый файл записывается «пустой» класс, а во второй и третий — специально невыровненные структуры. Как выясняется, на диск записывается точно столько же байтов, сколько структура занимает в памяти. В системе Visual C++.NET 2003 при отсутствии директивы выравнивания размеры переменных Empty t, BinNotPack1 A и BinNotPack2 B равны 1, 16 и 24 байт соответственно. Ровно столько же места они занимают и на диске — каждая в своем файле. Если же задать директиву выравнивания, то размеры структур станут равными 13:

```
#pragma pack(1)
```

Столько же записывается и в файлы.

В системе Borland C++ Builder 6 наблюдается аналогичная картина. При отсутствии выравнивания на диск записывается 8, 16 и 24 байта¹. При наличии директивы выравнивания на диск записывается столько же байтов, сколько и в системе Visual C++ — 1, 13 и 13.

Сериализация

В листинге 14.8 в файлы записывались объекты, не содержащие методов. Как мы знаем, обычные методы в классе места не занимают. Однако «за кадром» пока остался важный вопрос: влияет ли наличие виртуальных функций на размер выводимого объекта? Ведь в памяти объект класса с виртуальными функциями занимал на 4 байта больше, чем без виртуальных функций. Еще один важный вопрос — каким образом организовать вывод информации из контейнера во внешний файл. Разберемся сначала с первым вопросом, написав простейший пример (листинг 14.9).

Листинг 14.9. Вывод объекта с виртуальными функциями

```
#include <iostream>
#include <fstream>
using namespace std;
class Object // класс с виртуальными функциями
{ public:
    virtual void print() { cout << "virtual method!" << endl; }
};
int main(int argc, char* argv[])
{ Object t; // объект создан
  cout << sizeof(Object) << endl; // в памяти занимает 4 байта
  ofstream ostrm ("d:/oonumber1.bin", std::ios::binary);
  // выводим объект в файл
  if(ostrm.is_open()) ostrm.write((char *)&t, sizeof(Object));
  ostrm.close();
  return 0;
}
```

Хотя в классе нет полей, размер объекта в памяти составляет 4 байта. Как известно (см. главу 9), эта величина представляет собой размер указателя на таблицу виртуальных функций. На диск записываются те же 4 байта. Однако совершенно очевидно, что этому указателю на диске не место — он реально никуда не указывает. Следовательно, записывать объект в файл, не учитывая его внутреннюю структуру, просто нельзя. Особенно если объект представляет собой не единичную скалярную величину, а динамический контейнер (например, стек `TStack`, показанный в листинге 6.9).

Фактически при выводе объекта в двоичный файл на внешнем носителе осуществляется его «развертывание» в последовательность байтов, а при вводе происходит

¹ Вспомним (см. главу 1), что без директивы выравнивания в системе C++ Builder 6 пустой класс занимает 8 байт.

обратный процесс — из последовательности байтов конструируется объект. В объектно-ориентированном программировании придумали специальный термин для обозначения этого процесса: *сериализация*. Сериализация — это *обратимый* процесс преобразования произвольного набора структур данных C++ в последовательность байтов. Обратимость означает, что сериализованный объект можно снова «собрать» из последовательности байтов.

Нужно упомянуть об одной важной детали, связанной с двоичным вводом-выводом. Программа, которая считывает информацию из двоичного файла, должна «знать» размер и структуру считываемой информации. При записи в файл туда попадает только та информация, которая явно задана в операторах вывода. Никаких данных о типе выводимого значения на диске нет, если только мы сами их туда не запишем. Поэтому очень просто совершить ошибку: записать в файл объект одного типа, а прочитать записанные байты в переменную другого типа. Например, `int` и `float` в системе Visual C++.NET 2003 занимают в памяти одинаковое количество байтов — четыре. Поэтому в приведенных ранее примерах (см. листинг 14.7) можно было бы записать на диск массив дробных чисел типа `float`, а считывать его как массив целых чисел. Никаких сообщений об ошибках, естественно, не выдается; программа может даже работать, но результат, как вы понимаете, будет абсолютно неверным. Поэтому функции ввода-вывода обычно разрабатываются «парами»: одна — для записи в файл, другая — для чтения из файла.

Никто не знает структуру объекта лучше, чем сам объект. Поэтому, если требуется сохранить объект на диске, можно в классе реализовать два метода: `save()` и `load()`. Методы `save()` и `load()`, очевидно, должны быть симметричными — процесс сериализации должен быть обратимым. Эти методы должны работать с полями объекта. Например, для класса `TDate` (см. далее раздел «Перегрузка операций ввода-вывода») требуется выводить поля

```
unsigned long date;  
static fmtflags fmt;
```

Один из вариантов реализации методов `save()` и `load()` позволяет создать «моментальный снимок» объекта во внешней памяти, который в дальнейшем можно восстановить. Прототипы методов выглядят просто:

```
void save();  
void load();
```

В методе `save()` нужно определить локальный выходной поток и открыть его как двоичный. Поток, естественно, должен быть связан с файлом, который метод `load()` через свой локальный поток должен открывать как входной. Поэтому оба метода должны как-то получить доступ к одному и тому же имени файла. Очевидно, что этот файл может быть временным. Так как имя файла не должно совпадать с именем никакого другого файла, файл можно создать с помощью стандартной функции `tmpnam()` из библиотеки `<stdio>`. Прототип функции:

```
char *tmpnam(char *s);
```

В качестве аргумента можно задать массив символов, тогда имя будет сгенерировано в нем. Назначение метода `save()` понятно, и написать его несложно.

Единственная проблема — передать информацию об имени файла в метод `load()`. Это проще всего сделать, определив в классе поле, представляющее собой символьный массив, например:

```
char name[13];
```

Тогда метод `save()` будет заносить в это поле имя, сгенерированное функцией `tmpnam()`, а метод `load()` — открывать файл с этим именем.

Ввод-вывод скалярных объектов относительно прост, так как поля в классе известны и занимают фиксированное количество байтов. Значительно сложнее запрограммировать сериализацию динамических контейнеров, ведь требуется выводить не указатели, а значения, записанные в динамической памяти. Например, для динамического массива `TArray` (см. листинг 6.10) требуется выводить поле-размер `size_array` и само содержимое динамического массива, расположенного по адресу, записанному в поле-указателе `data`.

Реализуем методы для класса-шаблона `TArray` (см. листинг 11.9). Если метод `save()` написать относительно легко, то метод `load()` требует более пристального внимания. Метод похож на операцию присваивания: нужно создать новый динамический массив, в который скопировать сохраненные данные, а прежний массив уничтожить. Таким образом, метод `save()` должен сохранить и поле `size_array`, и сами элементы динамического массива (листинг 14.10).

Листинг 14.10. Методы `save()` и `load()` для массива `TArray`

```
template <class T, std::size_t n>
void TArray<T,n>::save()
{ tmpnam(name); // генерация случайного имени
  std::ofstream file(name, std::ios::binary);

  if(file.is_open())
  { file.write((char*)&size_array, sizeof(size_type));
    file.write((char*) data, sizeof(value_type)*n);
  }
  file.close();
  return;
}

template <class T, std::size_t n>
void TArray<T,n>::load()
{ std::ifstream file(name, std::ios::binary);
  if(file.is_open())
  { file.read((char*)&size_array, sizeof(size_type));
    value_type *p = data; // сохранили для возврата
    data = new T[size_array]; // новый массив
    file.read((char*) data, sizeof(value_type)*n);
    delete[]p;
  }
  file.close();
  return;
}
```

Поле `name` описано в приватной части класса `TArray` так:

```
char name[13];
```

Метод `save()` в начале работы генерирует случайное имя, открывает поток на запись и выводит данные в файл, который тут же и закрывается. Метод `load()`, соответственно, открывает поток как входной и читает данные из потока. При этом прежняя память возвращается, а данные из файла записываются в новый динамический массив.

Если нам нужна возможность ввода-вывода объектов в произвольный файл, то нужно передавать методам `save()` и `load()` в качестве параметра поток, связанный с этим файлом, например:

```
void save(ofstream &os);  
void load(ifstream &is);
```

В этом случае методы, естественно, упрощаются, так как никаких временных файлов не требуется, не нужно также открывать и закрывать поток. Класс ничего не знает о том, куда реально будет выведен объект, он только обеспечивает правильную сериализацию объекта. Аналогично метод чтения не знает, правильно ли установлена позиция чтения/записи в потоке — он просто выполняет чтение и конструирование объекта. Ответственность за правильное обращение с файлом возлагается на программу-клиента, использующую объекты данного класса.

Можно пойти дальше, и реализовать независимую библиотеку сериализации. Одна из библиотек Boost так и называется библиотекой сериализации — *Serialization*¹. Это сложная профессиональная работа, в которой используются нетривиальные возможности шаблонов, RTTI и множественное наследование (см. главу 10).

Форматирование¹ ввода-вывода

До сих пор мы обходились без форматирования. Это было возможно, поскольку в системе ввода-вывода установлены режимы форматирования по умолчанию для всех стандартных типов данных. Однако часто требуется управлять форматированием явным образом, не полагаясь на умолчания. Форматировать можно стандартные и строковые потоки, а также потоки, связанные с текстовыми файлами. Во всех случаях средства форматирования применяются совершенно одинаково. Собственно, нас более других интересуют ответы на следующие вопросы:

- ☐ Каким образом задать ширину поля вывода?
- ☐ Как указать выравнивание данных внутри поля?
- ☐ Как вывести дробные числа с заданной точностью?

Все это можно сделать с помощью стандартных средств, к которым относятся:

- ☐ флаги форматирования;
- ☐ методы форматирования;
- ☐ манипуляторы.

¹ Первоначальную версию разработал Роберт Рэми (Robert Ramey) в 2002 году.

Флаги форматирования

Флаги — это, как обычно, битовые константы. Флаги форматирования (см. п. п. 27.4.2.1.2 в [1]) определены в базовом классе библиотеки ввода-вывода `ios_base` (см. п. п. 27.4.2 в [1]):

```
left      = 0x0001,    // выравнивание влево
right     = 0x0002,    // выравнивание вправо
internal  = 0x0004,    // знак влево, число вправо
dec       = 0x0008,    // вывод десятичного целого
hex       = 0x0010,    // вывод шестнадцатеричного целого
oct       = 0x0020,    // вывод восьмеричного целого
fixed     = 0x0040,    // вывод дробного в виде dddd.dd (%f в printf())
scientific = 0x0080,   // вывод дробного в научном виде (%e в printf())
boolalpha = 0x0100,    // вывод true и false вместо 1 и 0
showbase  = 0x0200,    // вывод префиксов для целых oct и hex
showpoint = 0x0400,    // вывод незначащих нулей спереди
showpos   = 0x0800,    // вывод знака + для положительных целых
skipws    = 0x1000,    // пропускать символы-разделители (по умолчанию)
unitbuf   = 0x2000,    // очищать буфер после каждой операции
uppercase = 0x4000     // вывод символов E и X вместо e и x
```

Названия и назначение флагов являются стандартными, а вот конкретные значения битов в стандарте, естественно, не определены. В данном случае приведены значения, определенные в библиотеке `STLport`, которая поставляется вместе с системой `Borland C++ Builder 6`.

Для манипуляции флагами определено несколько методов (см. п. п. 27.4.2.2 в [1]):

```
fmtflags flags() const;           // получить флаги
fmtflags flags(fmtflags flags);   // получить и установить флаги
fmtflags setf(fmtflags flag);     // получить и установить флаги
void unsetf(fmtflags flags);      // сбросить флаги
```

Флаги сохраняются в поле типа `ios::fmtflags`. Установить (присвоить единичный бит) несколько флагов для стандартного потока `cout` можно любым из следующих способов:

```
cout.flags(ios::showpos|ios::showbase|ios::uppercase);
cout.setf(ios::showpos|ios::showbase|ios::uppercase);
```

Однако нужно помнить, что метод `flags()` предварительно сбрасывает все флаги (присваивает нулевой бит всем флагам), а метод `setf()` — не сбрасывает. Отдельные флаги сбрасываются методом `unsetf()`:

```
cout.unsetf(ios::showpos|ios::showbase|ios::uppercase);
```

Все флаги, кроме флага `skipws`, сброшены. Флаг `skipws` по умолчанию установлен, поэтому, если мы хотим, чтобы символы-разделители не пропускались, мы его должны сбросить:

```
cout.unsetf(std::ios::skipws);
```

Важным флагом является `unitbuf`. По умолчанию он сброшен, что означает накопление символов потока в буфере вывода. Если же флаг установить, то вывод выполняется практически без буферизации — выходной буфер очищается после каждой операции вывода. Этот флаг устанавливается по умолчанию для потока `cerr`.

Ряд флагов объединяется в группы (см. п. п. 27.4.2.1.2 в [1]). Для каждой группы определена маска, чтобы с флагами в группе было проще работать:

```
adjustfield = internal | left | right
basefield   = dec | hex | oct
floatfield  = fixed | scientific
```

В каждый конкретный момент времени только один из флагов группы может быть установлен. Для работы с такими флагами перегружен метод `setf()`:

```
fmtflags setf(fmtflags flag, fmtflags mask);
```

Этот метод сначала сбрасывает все флаги группы, а потом устанавливает требуемый. Например, восьмеричная система счисления для стандартного потока `cout` задается так:

```
cout.setf(std::ios::oct, std::ios::basefield);
```

Метод `copyfmt()` (см. п. п. 27.4.4.2 в [1]) позволяет скопировать состояние формата потока в другой поток, например:

```
strm.copyfmt(cout);
```

Поток `strm` получает состояние формата стандартного потока `cout`.

Методы форматирования

Помимо установки и сброса флагов, можно установить ширину поля, точность представления и задать символ-заполнитель (по умолчанию — пробел). Эти методы имеют следующие прототипы (см. п. п. 27.4.4.2 в [1]):

```
streamsize precision() const;
streamsize precision(streamsize newprecision);
streamsize width() const;
streamsize width(streamsize newwidth);
char fill() const;
char fill(char fill);
```

Уже по названиям легко догадаться о назначении методов:

- метод `precision()` позволяет получить и задать точность — количество цифр после десятичной точки;
- метод `width()` позволяет получить и задать ширину поля (вывода);
- метод `fill()` позволяет получить и задать символ-заполнитель.

Методы `precision()` и `fill()` влияют на все операции вывода. Это означает, что один раз установленная точность не изменяется до следующего вызова метода `precision()`. А вот метод `width()` работает только для ближайшей операции вывода `<<`. Например:

```
cout.fill('~'); cout.width(5);
cout << 123 <<';' << 15;
```

В результате выполнения этих операторов на экран будет выведено

```
~~123;15
```

Если бы ширина поля задавалась сразу для всех операций <<, на экране появилась бы следующая строка:

```
~~123~~~;~~15
```

Если выводимое значение не помещается в заданной ширине поля, то все равно выводится значение целиком, например:

```
cout.width(5);
cout << "1234567";
```

В результате на экране появляется строка

```
1234567
```

Позже мы познакомимся с применением этих методов и флагов на примерах.

Манипуляторы

Явная установка флагов форматирования и использование методов не слишком удобны для форматирования потока. Поэтому в библиотеке реализован набор операций для манипулирования состоянием потока данных непосредственно во время операции ввода-вывода, которые так и называются *манипуляторами*. В библиотеке <iostream> определены манипуляторы (см. п. п. 27.4.5 в [1]), соответствующие флагам форматирования (табл. 14.3). Эти манипуляторы не имеют аргументов, и их названия совпадают с именами флагов — фактически они тоже устанавливают и сбрасывают одноименные флаги.

Таблица 14.3. Манипуляторы, соответствующие флагам форматирования

Манипулятор	Эквивалентный вызов метода
left	stream.setf(std::ios::left, std::ios::adjustfield);
right	stream.setf(std::ios::right, std::ios::adjustfield);
internal	stream.setf(std::ios::internal, std::ios::adjustfield);
dec	stream.setf(std::ios::dec, std::ios::basefield);
hex	stream.setf(std::ios::hex, std::ios::basefield);
oct	stream.setf(std::ios::oct, std::ios::basefield);
fixed	stream.setf(std::ios::fixed, std::ios::floatfield);
scientific	stream.setf(std::ios::scientific, std::ios::floatfield);
boolalpha	stream.setf(std::ios::boolalpha);
noboolalpha	stream.unsetf(std::ios::boolalpha);
showbase	stream.setf(std::ios::showbase);
noshowbase	stream.unsetf(std::ios::showbase);
showpoint	stream.setf(std::ios::showpoint);
noshowpoint	stream.unsetf(std::ios::showpoint);
showpos	stream.setf(std::ios::showpos);

Манипулятор	Эквивалентный вызов метода
noshowpos	stream.unsetf(std::ios::showpos);
skipws	stream.setf(std::ios::skipws);
noskipws	stream.unsetf(std::ios::skipws);
unitbuf	stream.setf(std::ios::unitbuf);
nounitbuf	stream.unsetf(std::ios::unitbuf);
uppercase	stream.setf(std::ios::uppercase);
nouppercase	stream.unsetf(std::ios::uppercase);

Одиночным флагам соответствует по два манипулятора: один — установленному флагу, другой (с префиксом «но») — сброшенному. Флагам, объединенным в группы, соответствует по одному манипулятору.

В заголовке `<iostream>` определен манипулятор ввода `ws` (см. п. п. 27.6.1.4 в [1]) — ввод с игнорированием пропусков, в заголовке `<ostream>` — манипуляторы вывода `endl`, `ends`, `flush` (см. п. п. 27.6.2.7 в [1]). Манипулятор `flush` принудительно выводит выходной буфер на устройство. Манипулятор `endl` мы уже неоднократно применяли. Его действие состоит в записи символа `'\n'` в выходной поток, после чего выходной буфер принудительно выводится на устройство. Манипулятор `ends` записывает символ `'\0'` (символ завершения строки) в поток.

В заголовке `<iomanip>` определены шесть манипуляторов с аргументами (см. п. п. 27.6.3 в [1]); пять из них представлены в табл. 14.4. Чтобы их использовать, нужно задать в программе оператор

```
#include <iomanip>
```

Таблица 14.4. Манипуляторы с аргументами

Манипулятор	Эквивалентный метод
setprecision(streamsize n)	streamsize precision(streamsize n);
setw(streamsize n)	streamsize width(streamsize n);
setfill(char ch)	char fill(char fill);
setiosflags(fmtflags n)	fmtflags setf(fmtflags flag);
resetiosflags(fmtflags n)	fmtflags unsetf(fmtflags flag);

Шестой манипулятор позволяет установить систему счисления:

```
setbase(int base)
```

Этот манипулятор может использоваться вместо манипуляторов `dec`, `oct`, `hex` (или вместо установки соответствующих флагов):

- вызов `setbase(8)` эквивалентен использованию манипулятора `oct`;
- вызов `setbase(10)` эквивалентен использованию манипулятора `dec`;
- вызов `setbase(16)` эквивалентен использованию манипулятора `hex`.

Вызов `setbase(0)` задает вывод в десятичной системе счисления, например:

```
cout << setbase(16) << 10 << ';' << setbase(0) << 0x10 << endl;
```

В результате на экран выводится следующее:

```
a;16
```

Буква «a» — это 10 в шестнадцатеричной системе счисления, а шестнадцатеричная константа `0x10` выводится как десятичное число 16.

Применение манипуляторов вместо установки флагов и вызова методов облегчает программисту жизнь, так как манипуляторы задаются непосредственно в операции ввода-вывода в том месте, где необходимы. Простой пример показывает, насколько проще иметь дело с манипуляторами, чем с флагами и методами:

```
// вывод с использованием методов и флагов
cout.width(10); cout.fill('~');
cout.setf(ios::internal, ios::adjustfield);
cout.setf(ios::hex, ios::basefield);
cout << val;
cout.put('\n'); cout.flush();
// тот же вывод с использованием манипуляторов
cout << setfill('~') << setw(10) << hex << internal << val << endl;
```

Комментарии, как говорится, излишни!

Форматированный ввод

Большинство средств форматирования предназначены для форматирования вывода (см. п. п. 27.6.2.5 в [1]). Однако некоторые из них оказывают влияние на ввод (см. п. п. 27.6.1.2 в [1]). Как уже отмечалось, сброс флага `skipws` (или использование манипулятора `noskipws`) отменяет пропуск символов-разделителей при вводе.

Если выставлен флаг `boolalpha` (или использован манипулятор), то во входном потоке булевы значения должны быть представлены как строки `true` и `false` (без кавычек). Если этот флаг не установлен (а по умолчанию он не установлен), то во входном потоке должны быть заданы числа 0 (`false`) и 1 (`true`).

На ввод влияют флаги группы `basefield` (манипулятор `setbase()` или манипуляторы `dec`, `hex` и `oct`). Если ни один флаг не установлен (или использован манипулятор `setbase(0)`), то основание вводимого числа определяется при вводе. Если в начале числа стоит префикс `0x` или `0X`, то число считается шестнадцатеричным; если число начинается с префикса `0`, то оно считается восьмеричным. Во всех остальных случаях число считается десятичным.

Если установлен один из флагов (или использован один из манипуляторов: `dec`, `hex`, `oct`, `setbase(10)`, `setbase(8)`, `setbase(16)`), то он задает фиксированное основание; в соответствии с основанием выполняется контроль вводимых символов числа.

Задание ширины поля (метод `width()` или манипулятор `setw()`) влияет на ввод в объект типа `string` или в символьный массив. Если размер поля равен `n`,

то ввод выполняется либо до первого символа-разделителя, либо до n символов, например:

```
char s[81];  
// чтение не более 80 символов  
cin >> setw(sizeof(s)) >> s;
```

Вводится не более 80 символов, хотя вызов `sizeof(s)` возвращает 81. После этого ширина поля сбрасывается в нуль.

При вводе других типов данных значение ширины поля не играет никакой роли. Тем не менее при перегрузке операции ввода можно реализовать ввод из поля фиксированной длины. Для этого нужно сначала выполнить ввод в строку, а потом использовать строковый поток для чтения из этой строки.

Написание собственных манипуляторов

Иногда требуется нестандартный манипулятор. Программирование манипуляторов без аргументов не представляет особой сложности. Для этого надо просто написать функцию, которая получает и возвращает ссылку на поток. Например, пусть нам нужен манипулятор, вставляющий в поток символ табуляции (листинг 14.11).

Листинг 14.11. Манипулятор без аргументов

```
std::ostream& tab(std::ostream& os)  
{ return (os << '\t'); }
```

Использовать такой манипулятор так же просто, как и стандартный:

```
cout << 12 << tab << 25 << endl;
```

Обычно манипуляторы без аргументов пишутся для объединения свойств нескольких стандартных манипуляторов. Например, ранее мы рассматривали следующий оператор вывода:

```
cout << setfill('~') << setw(10) << hex << internal << val << endl;
```

Этот оператор можно записать значительно короче, если объединить все использованные в нем манипуляторы в одной функции (листинг 14.12).

Листинг 14.12. «Объединяющий» манипулятор

```
ostream& setfix(ostream& os)  
{ os.width(10); os.fill('~');  
  os.setf(ios::internal, ios::adjustfield);  
  os.setf(ios::hex, ios::basefield);  
  return os;  
}
```

В результате вывод в `cout` той же переменной `val` в том же формате записывается значительно короче:

```
cout << setfix << val << endl;
```


Написать манипулятор с аргументами несколько сложнее. Собственно, для создания манипулятора, совместимого по интерфейсу со стандартными манипуляторами, нужно разобраться в стандартном механизме реализации. Однако вместо этого можно просто написать некоторый класс, для которого, как обычно, определить функцию `operator<<`. Джерри Шварц¹ назвал такие классы *эффекторами* (см. также [12]). Например, пусть нам требуется вывести целые значения в двоичном виде. Напишем класс `binary` (листинг 14.13).

Листинг 14.13. «Манипулятор» для двоичного вывода

```
class binary
{ unsigned long k;
public:
    binary(unsigned long k): k(k) {}
    friend ostream& operator<<(ostream& os, const binary& t);
};
inline ostream& operator<<(ostream& os, const binary& t)
{ const unsigned long MAX = numeric_limits<unsigned long>::max();
  unsigned long bit = ~(MAX >> 1); // старший бит
  while(bit) { os << (t.k & bit?'1':'0'); bit >>= 1; }
  return os;
}
```

В функции `operator<<` самыми важными являются две строки:

```
const unsigned long MAX = numeric_limits<unsigned long>::max();
unsigned long bit = ~(MAX >> 1); // старший бит
```

В первой строке в переменную `MAX` заносится максимально возможное беззнаковое целое число, определенное в стандартном классе числовых пределов `numeric_limits`.

ПРИМЕЧАНИЕ

Класс-шаблон `numeric_limits<>` определен в заголовке `<limits>`.

В двоичной записи это максимальное число представляет собой набор единичных битов:

1111...111

Во второй строке все биты числа сдвигаются вправо на один разряд; самый правый бит (младший разряд числа) теряется; в самый левый бит (старший разряд числа) заносится нуль, так как тип — беззнаковый. Таким образом, в результате сдвига образуется такая конфигурация битов:

0111...111

Потом эта конфигурация инвертируется операцией `~` и превращается в

1000...000

Это значение и заносится в переменную `bit`. Далее старшая единица в цикле сдвигается на 1 разряд, пока значение `bit` не станет равно нулю.

¹ Разработчик объектно-ориентированной библиотеки ввода-вывода.

Такой класс позволяет нам выводить любые целые в двоичном виде, например:

```
short a = -2;
cout << binary(a) << endl;
```

Однако этот класс выводит в выходной поток всегда такое количество битов, которое содержится в типе `unsigned long`. Мы легко можем преобразовать этот класс в шаблон, который позволит выводить для беззнаковых целых нужное количество битов (листинг 14.14).

Листинг 14.14. Эффектор-шаблон

```
// опережающие объявления
template <class T> class binary;
template<class T> ostream& operator<<(ostream& os, const binary<T>& t);
// класс-шаблон
template <class T>
class binary
{ T k;
public:
    binary(T k): k(k) {}
    friend ostream& operator<< <<(ostream& os, const binary<T>& t);
};
template<class T>
inline
ostream& operator<<(ostream& os, const binary<T>& t)
{ T MAX = numeric_limits<T>::max();
  T bit = ~(MAX >> 1);
  while(bit)
  { os << (t.k & bit?'1':'0'); bit >>= 1; }
  return os;
}
```

Обратите внимание на опережающие объявления и прототип функции-шаблона `operator<<` в классе `binary`.

Использовать данный класс-шаблон в качестве манипулятора можно так:

```
short a = -2;
cout << binary<unsigned short>(a) << endl;
cout << binary<unsigned char>(128)<< endl;
```

В результате на экран выводится следующее:

```
1111111111111110
10000000
```

Как видим, на экране ровно столько битов, сколько занимают типы `unsigned short` и `unsigned char`. Заметим, что использовать в качестве аргумента шаблона знаковый тип нельзя — программа заикнется!

Применение средств форматирования

Рассмотрим, как применять средства форматирования для создания форматированного отчета о наличии товаров. Допустим, у нас есть подготовленный в текстовом редакторе файл с перечнем товаров, в котором одна запись занимает 2 строки: на первой строке — название товара, на второй — количество и цена.

Пусть файл с именем `tovar.dat` записан на дискете. Требуется записать новый файл-ведомость, который потом можно будет выводить на принтер. В одной строке файла 5 полей:

1. Номер по порядку, ширина поля — 3 символа.
2. Название товара — 40 символов.
3. Цена — 7 символов, два знака после запятой.
4. Количество — 6 символов, целое число.
5. Стоимость = количество * цена — 12 символов, 2 знака после запятой.

Поля разделяются символом `|` (вертикальная черта). Название нужно выравнивать влево, а если оно занимает больше 40 символов, то его надо обрезать. Числа нужно выравнивать вправо, причем числа, обозначающие деньги, выводить с фиксированной точкой и двумя знаками после запятой. Десятичная система установлена по умолчанию, а ширину поля приходится задавать для каждого поля (для каждой операции вывода) заново. После всех записей нужно вывести итоговую сумму. Записывать файл будем тоже на дискету (листинг 14.15).

Листинг 14.15. Форматирование ведомости

```
#include <fstream>
#include <iomanip>
#include <string>
using namespace std;
int main()
{
    struct Tovar
    {
        string name;           // наименование товара
        unsigned int HowMany;   // количество
        double Price;           // цена
    };
    Tovar tmp;
    ifstream inf("a:/tovar.dat"); // исходный файл
    ofstream to("a:/summa.dat");   // файл-ведомость
    int i = 1;    double Summa = 0;
    // чтение первой записи
    getline(inf, tmp.name);
    inf >> tmp.HowMany; inf >> tmp.Price; inf.ignore();
    while(!inf.eof())           // пока не конец файла
    {
        to << right << setw(3) << i++ << '|';
        to << left << setw(40) << tmp.name.substr(0,40) << '|';
        to.precision(2);
        to << right << fixed
            << setw(6) << tmp.HowMany << '|'
            << setw(7) << tmp.Price << '|'
            << setw(12) << tmp.Price*tmp.HowMany << '|'
            << endl;
        getline(inf, tmp.name);
        inf >> tmp.HowMany; inf >> tmp.Price; inf.ignore();
    };
    to << setw(72) << Summa << endl;
    return EXIT_SUCCESS;
}
```

После чтения чисел потребовалось вставить вызов `ignore()` для пропуска символа конца строки, иначе ввод работал бы неправильно. Максимально используются манипуляторы, что делает программу существенно понятнее.

Пусть исходный файл `tovar.dat` содержит следующие данные:

```
Авторучки перьевые
1031 257.50
Авторучки шариковые
2143 6.50
Лампы настольные люминесцентные белые
300 350
Папки для бумаг
10677 12.67
```

Тогда результирующий файл `summa.dat` будет выглядеть так:

1 Авторучки перьевые	1031 257.50	265482.50
2 Авторучки шариковые	2143 6.50	13929.50
3 Лампы настольные люминесцентные белые	300 350.00	105000.00
4 Папки для бумаг	10677 12.67	135277.59
		519689.59

Строковые потоки

Строковые потоки — это универсальные средства перевода внутренних двоичных значений в строки (выходной строковый поток) и обратно (входной строковый поток). Строковые потоки объектно-ориентированной библиотеки (см. п. 27.7 в [1]) могут быть и двунаправленными. Кроме того, при работе с ними нам нет необходимости резервировать память для строк. И наконец, со строковыми потоками можно использовать все средства форматирования или собственные написанные манипуляторы.

Чтобы использовать строковые потоки, мы должны прописать в программе оператор

```
#include <sstream>
```

После этого в программе можно в качестве объектов объявлять строковые потоки трех видов:

- входной `istringstream` (см. п. п. 27.7.2 в [1]);
- выходной `ostringstream` (см. п. п. 27.7.3 в [1]);
- двунаправленный `stringstream` (см. п. п. 27.7.4 в [1]).

Чтобы продемонстрировать возможности строковых потоков, напомним несколько примеров. Первый пример (листинг 14.16) демонстрирует возможности выходных строковых потоков.

Листинг 14.16. Выходной строковый поток

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

Листинг 14.16 (продолжение)

```
int main()
{
    char s[] = "computer", c = 'L';
    int i = 35;
    float fp = 1.7320534f;
    ostream os;
    /* Форматирование и вывод данных */
    os << "String: " << s << "; ";
    << "Character: " << c << "; ";
    << "Integer: " << i << "; ";
    << "Real: " << fp << "; ";
    << endl;
    cout << "Output:" << '\n' << os.str() << endl;
    cout << "character count = " << os.str().length() << endl;
    return EXIT_SUCCESS;
}
```

Эта программа выводит на экран следующее:

```
Output:
String: computer; Character: L; Integer: 35; Real: 1.732053;
character count = 61
```

Программисту нет необходимости резервировать память для формируемой строки — строковый поток все делает во внутреннем буфере. Нет также необходимости следить за позицией внутри буфера — все делает сам поток. Внутренний буфер всегда можно получить, вызвав для потока метод `str()`.

Выходной поток может быть создан в режиме дозаписи (как и файл). Тогда он должен быть проинициализирован существующей строкой, например:

```
string s = "Шестнадцатеричное число = ";
ostream os(s, ios::out|ios::app);
os << hex << 123;
cout << os.str() << endl;
```

В результате работы этого фрагмента на экран будет выведено:

```
Шестнадцатеричное число = 7B
```

Сама строка `s` остается без изменений.

Теперь покажем, как использовать двунаправленный поток (листинг 14.17).

Листинг 14.17. Применение двунаправленного строкового потока

```
{
    string names[4] = {"Peter", "Mike", "Shea", "Jerry"};
    string temp[4];
    string name;
    int age;
    long salary;
    stringstream strm; // двунаправленный поток
    srand( (unsigned)time(NULL)); // инициализация датчика случайных чисел
    /* создание данных name, age и salary */
    for (int loop=0; loop < 4; ++loop)
        strm<<names[loop]<<' '<<rand()%10+20<<' '<<rand()%5000+27500L<< endl;
```

```

/* вывод шапки */
cout << setw(4) << "#" << " | "
    << left << setw(20) << "Name" << " | "
    << right << setw(5) << "Age" << " | "
    << right << setw(15) << "Salary" << endl;
cout << " -----\n";
strm.seekg(0); // перевод позиции чтения/записи в начало потока
for (loop=0; loop < 4; ++loop)
{ strm >> name >> age >> salary; // ввод name, age и salary
  cout << right << setw(4) << (loop + 1) << " | "
      << left << setw(20) << name << " | "
      << right << setw(5) << age << " | "
      << setw(15) << salary << endl;
}
}

```

Эта программа сначала пишет в поток `strm`. Потом из того же потока выполняется считывание и вывод на экран.

Перед считыванием устанавливается позиция чтения методом `seekg()` (см. п. п. 27.6.1.3 в [1]). В объектно-ориентированной библиотеке позиционирование можно выполнять и для строковых, и для файловых потоков — этого нельзя делать только для стандартных потоков. Мы рассмотрим методы позиционирования далее.

Входной строковый поток можно использовать, чтобы реализовать считывание чисел из поля фиксированной длины. Не будем писать манипулятор с аргументами, а напишем простой шаблон функции (листинг 14.18).

Листинг 14.18. Ввод из поля фиксированной длины

```

template <class T>
void fixedread(istream &in, T &t)
{
    if (in.width() > 0) // ширина поля установлена
    {
        string field;
        in >> field; // ввод по ширине или до пробела
        istreamstream strm(field); // входной строковый поток
        if (!(strm >> t)) // ввод из строкового потока
            in.setstate(ios::failbit); // ошибки ввода
    }
    else // ширина поля по умолчанию
        in >> t; // ввод в переменную
}

```

Если ширина поля ввода установлена (то есть не равна нулю), то объявляется строка для ввода и ввод выполняется в нее. Чтение выполняется либо до пробела, либо до заявленного количества символов. Затем этой строкой инициализируется строковый поток. Обратите внимание на то, что ввод из строкового потока проверяется на корректность и в случае ошибки устанавливается флаг `failbit` для входного потока-параметра.

Использовать эту функцию можно так:

```

int k = 0; cin.width(3);
fixedread(cin, k);

```

Если мы последовательно нажмем клавиши 1, 2, 3, 4, то в переменную `k` попадет значение 123, а при нажатии, например, клавиш 5, 6, пробел, 4, 5, 6, 7 — значение 56.

Перегрузка операций ввода-вывода

Мы уже сталкивались с перегрузкой операций ввода-вывода (см. п. п. 27.6.1.2.3 и 27.6.2.5.3 в [1]), когда рассматривали дружественные функции и виртуализацию внешних функций (см. листинги 3.12 и 9.12). Однако перегрузка выполнялась в простейшем виде, только для иллюстрации механизмов дружественности и виртуализации. Теперь мы реализуем более развитые операции для ввода и вывода дат с использованием форматирования и строковых потоков. Определим возможные форматы дат:

- три целых числа, разделенных точками; в начале — день, например 31.01.2005;
- целое число, начинающееся с года, например 20050505;
- целое число-день, строка-месяц, целое число-год, например 5-май-2005.

Соответственно, операция ввода даты должна «уметь» настраиваться на вид вводимой даты. И операция вывода тоже должна обеспечивать вывод в одном из этих трех форматов. Это легко можно сделать, написав три разных эффектора по аналогии с эффектором `binary` (см. листинг 14.8). Более того, так как форматирование — это «внутреннее дело» класса даты, эти эффекторы можно объявить вложенными классами класса даты.

Однако возможны и другие пути форматирования даты. Начнем с вывода. Реализуем три манипулятора без аргументов, которые обеспечат нам ту же функциональность:

- манипулятор `pointdate` задает вывод даты в виде чисел, разделенных точками;
- манипулятор `intdate` задает вывод даты в виде целого числа;
- манипулятор `stringdate` задает вывод даты с названием месяца.

Если манипуляторы отсутствуют, то дата выводится в виде `pointdate`. Таким образом, если переменная `d` представляет собой дату, то вывод ее может быть таким:

```
cout << intdate << d;  
cout << pointdate << d;  
cout << stringdate << d;
```

Сразу понятно, что манипуляторы должны где-то в объекте `d` «оставлять следы», а операция вывода `operator<<` должна иметь к этой информации доступ. По аналогии со стандартным вводом-выводом манипуляторы без аргументов могут устанавливать флаг, а операция вывода этот флаг использовать. Таким образом, в класс даты нужно включить поле флагов и определить константы-флаги.

Однако есть одна тонкость: манипулятор без аргументов не получает объект-дату как параметр — аргументом такого манипулятора может быть только поток. Поэтому первое (самое простое) решение — сделать поле флагов статическим. В этом случае установка любого флага будет действовать на вывод всех объектов-дат до установки другого флага. Собственно, так работают и стандартные флаги форматирования: установка, например, флага выравнивания `left` действует до тех пор, пока не будет установлен другой флаг выравнивания.

Очевидно, что манипуляторы должны быть дружественными функциями для класса даты, чтобы иметь доступ к закрытому полю флагов. Таким образом, класс даты может выглядеть так, как показано в листинге 14.19.

Листинг 14.19. Класс TDate, обеспечивающий форматирование дат

```
class TDate
{
public:
    typedef unsigned int fmtflags;
    // константы-флаги
    static const fmtflags pointDate = 0x00;
    static const fmtflags intDate   = 0x01;
    static const fmtflags stringDate = 0x02;
    // методы доступа к флагам
    fmtflags getflags() const { return fmt; }
    void pointdate() { fmt = pointDate; }
    void intdate()   { fmt = intDate; }
    void stringdate(){ fmt = stringDate; }
    // конструкторы даты
    TDate():date(0) {}
    TDate(unsigned long date):date(date){}
    TDate(unsigned int d, unsigned int m, unsigned int y)
        :date(y*10000+m*100+d) {}
    TDate(unsigned int d, string month, unsigned int y);
    TDate(const TDate &d):date(d.date){}
    // ввод-вывод
    friend istream& operator>>(istream& is, TDate &data);
    friend ostream& operator<<(ostream& os, const TDate &data);
    // манипуляторы
    friend ostream& intdate (ostream &os) { fmt = intDate; return os; }
    friend ostream& pointdate (ostream &os) { fmt = pointDate; return os; }
    friend ostream& stringdate(ostream &os) { fmt = stringDate; return os; }

private:
    unsigned long date;
    static fmtflags fmt;
    static const string m[12];
};

TDate::fmtflags TDate::fmt;
const string TDate::m[12] =
{ "янв", "фев", "мар", "апр", "май", "июн", "июл", "авг", "сен", "окт", "ноя", "дек" };
```

Еще несколько замечаний о классе. Дата представлена единственным полем, в котором дата хранится в виде *год 10000 + месяц 100 + день*.

Так как поле флагов статическое, то конструкторы его не инициализируют. Помимо манипуляторов, класс даты, естественно, предоставляет методы установки и доступа к флагам.

Операция вывода даты может быть реализована так, как показано в листинге 14.20.

Функция, в зависимости от установленного флага, «собирает» в строковом потоке нужный вид даты, а потом выводит буфер строкового потока в поток-параметр. Все форматы даты выводятся с ведущими нулями. Нужно подключить библиотеку `<iomanip>`, так как используются манипуляторы с аргументами.

Листинг 14.20. Форматированный вывод даты

```
ostream& operator<<(ostream& os, const TDate &data)
{
    ostringstream s;           // строковый поток
    int day   = data.date%100;
    int year  = data.date/10000;
    int month = data.date/100%100;
    switch(TDate::fmt)          // флаг-переключатель
    { case TDate::pointDate:    // формат dd.mm.yyyy
        s << setfill('0')
          << setw(2) << day   << '.'
          << setw(2) << month << '.'
          << setw(4) << year;
        break;
      case TDate::intDate:      // формат yyyymmdd
        s << data.date;
        break;
      case TDate::stringDate:   // формат dd-mmm-yyyy
        s << setfill('0')
          << setw(2) << day << '-'
          << TDate::m[month-1] << '-'
          << setw(4) << year;
        break;
    }
    return os << s.str();
}
```

Операция ввода, естественно, должна быть «симметричной», то есть обеспечивать ввод любого из этих форматов. Очевидно, необходимо реализовать те же манипуляторы ввода как дружественные классу TDate функции:

```
friend istream& intdate (istream &is) { fmt = intDate; return is; }
friend istream& pointdate (istream &is) { fmt = pointDate; return is; }
friend istream& stringdate(istream &is) { fmt = stringDate; return is; }
```

Кроме того, при вводе необходимо контролировать правильность задания месяца и дня. Очевидно, это можно сделать, используя входной строковый поток и ограничив размер поля ввода. Не особо задумываясь о производительности и оптимальности кода, реализуем функцию ввода самым простым способом (листинг 14.21). Так как функция служит только для ввода даты и «не знает» ничего о входном потоке, в случае любой ошибки будем устанавливать флаг `ios::badbit`. Таким образом, после первой же ошибки последующие операции ввода выполняться не будут. Кроме того, это даст нам возможность «повесить» на этот флаг прерывание ввода, вызвав метод `extensions()`, например, в конструкторе. Принимать решение о продолжении или завершении ввода будет вызывающая программа.

Листинг 14.21. Функция ввода даты

```
istream& operator>>(istream& in, TDate &data)
{
    string field;              // строка для ввода
    istringstream s;           // входной строковый поток
    int day, month, year;
    int days[12] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
    switch(TDate::fmt)         // флаги-переключатели
```

```

{   case TDate::pointDate:           // формат dd.mm.yyyy
    // ввод дня
    in.width(2);                     // количество цифр дня
    in >> field;                      // ввод в строку
    s.str(field);                     // строку -> в поток
    if (!(s >> day))                  // неверные символы в дне
        in.setstate(ios::badbit);
    if (!(day > 0))                    // день = 0
        in.setstate(ios::badbit);
    if (in.peek()!='.')               // следующий символ - не точка
        in.setstate(ios::badbit);
    else in.ignore();                 // если точка - пропускаем
    // ввод месяца
    in.width(2); in >> field; s.str(field);
    s.clear();                        // сброс eofbit
    if (!(s >> month))                // неверные символы в месяце
        in.setstate(ios::badbit);
    if (!(0<month)&&(month<13))        // неправильный месяц
        in.setstate(ios::badbit);
    if (day>days[month-1])            // неправильный день
        in.setstate(ios::badbit);
    if (in.peek()!='.')               // следующий символ - не точка
        in.setstate(ios::badbit);
    else in.ignore();                 // если точка - пропускаем
    // ввод года
    in.width(4); in >> field; s.str(field); s.clear();
    if (!(s >> year))                 // неверные символы в годе
        in.setstate(ios::badbit);
    data.date = year*10000+month*100+day;
    break;
case TDate::intDate:                 // формат ууууmmdd
    in >> data.date;                  // вводим как целое число
    if (data.date > 0)                // если что-то ввелось
    { day = data.date%100;             // получаем день
      if (!(day > 0))                  // если день = 0
        in.setstate(ios::badbit);
      month = data.date/100%100;      // получаем месяц
      if (!(0<month)&&(month<13))     // неправильный месяц
        in.setstate(ios::badbit);
      if (day>days[month-1])         // неправильный день
        in.setstate(ios::badbit);
    }
    break;
case TDate::stringDate:              // формат dd-mmm-yy
    // ввод дня - аналогично первому оператору case
    in.width(2); in >> field; s.str(field);
    if (!(s >> day)) in.setstate(ios::badbit);
    if (!(day > 0)) in.setstate(ios::badbit);
    if (in.peek()!='-') in.setstate(ios::badbit);
    else in.ignore();
    // ввод месяца
    in.width(3); in >> field;
    month = 0;
    for (int i = 0; i<12; ++i) if (TDate::m[i]==field) month = i;

```

Листинг 14.21 (продолжение)

```

        if (! (0 < month) && (month < 12)) in.setstate(ios::badbit);
        if (day > days[month-1])          in.setstate(ios::badbit);
        if (in.peek() != '-')             in.setstate(ios::badbit);
        else in.ignore();
        in.width(4); in >> field; s.str(field); s.clear();
        if (! (s >> year))                 in.setstate(ios::badbit);
        data.date = year*10000+month*100+day;
        break;
    }
    return in;
}

```

Мы не проверяем февраль високосного года — и без этой проверки функция оказалась достаточно сложной. В функции нужно обратить внимание на несколько моментов:

- При вводе даты в первом формате мы проверяем наличие символа точки между числами. Метод `peek()` (см. п. п. 27.6.1.3 в [1]) позволяет просмотреть первый непрочитанный символ потока, не извлекая его оттуда. В третьем варианте мы точно так же проверяем наличие символа - (дефис).
- Установка ширины поля ввода для входного потока делается перед каждой операцией ввода, так как предыдущая операция ввода сбрасывает это значение в ноль.
- Перед очередной операцией ввода из входного строкового потока `s` нужно сбросить флаги потока, так как операция ввода выставляет флаг `eofbit`. Если этого не сделать, то ввод из потока не выполняется.

Проверим работу наших функций ввода-вывода и манипуляторов:

```

TDate D(07, 05, 2005);
TDate R(D);
cout << D << endl;                // вывод в формате dd.mm.yyyy
cout << intdate << R << endl;       // вывод в формате yyyymmdd
cout << D << endl;                // вывод в формате yyyymmdd
cout << stringdate << R << endl;    // вывод в формате dd-mmm-yyy
ofstream out("c:/TextFiles/Data.fmt");
out << pointdate << R << endl;      // запись в файл
out.close();
ifstream in("c:/TextFiles/Data.fmt");
TDate T;
in >> pointdate >> T;              // чтение из файла
cout << "T=" << T << endl;
cout << "T=" << stringdate << T << endl;

```

Этот короткий фрагмент выводит на экран следующие строки:

```

07.05.2005
20050507
20050507
07-май-2005
T=07.05.2005
T=07-май-2005

```

Внимательно просмотрев последовательность операций вывода, убеждаемся, что все операции работают совершенно корректно. В результате в каталоге `TextFiles` создается текстовый файл `Data.fmt`, в котором записана дата 07.05.2005.

Вместо манипуляторов для переключения режима представления даты можно использовать методы, например:

```
R.stringdate(); cout << R << endl;
D.intdate();    cout << D' << endl;
```

Однако совершенно очевидно, что использование манипуляторов делает программу более короткой и понятной.

Произвольный доступ

Как уже отмечалось, C++ позволяет выполнять позиционирование в любых потоках, кроме стандартных. Методы позиционирования приведены в табл. 14.5. Методы позиционирования различаются для входных и выходных потоков: для входных потоков (см. п. п. 27.6.1.3 в [1]) имена методов заканчиваются символом «g» (от слова *get*), для выходных (см. п. п. 27.6.2.4 в [1]) — символом «p» (от слова *put*).

Таблица 14.5. Методы позиционирования

Метод	Описание
<code>pos_type tellg()</code>	Получить текущую (абсолютную) позицию чтения
<code>istream& seekg (pos_type p)</code>	Установить абсолютную позицию чтения
<code>istream& seekg (off_type p, ios::seekdir s)</code>	Установить относительную позицию чтения
<code>pos_type tellp()</code>	Получить текущую (абсолютную) позицию записи
<code>istream& seekp (pos_type p)</code>	Установить абсолютную позицию записи
<code>istream& seekp (off_type p, ios::seekdir s)</code>	Установить относительную позицию записи

Функции `tellp()` и `tellg()` возвращают абсолютное смещение от начала потока. Символы потока нумеруются, как и индексы массива, начиная с нуля. Однако тип `pos_type` не является целым типом. Поэтому получать текущую позицию чтения в файловом потоке нужно так:

```
ios::pos_type p = file.tellg();
```

Можно сохранить текущую позицию и в переменной типа `streampos`, например:

```
streampos pos = file.tellp();
```

Переход к позиции, сохраненной в переменной `pos`, делается так:

```
file.seekp(pos);
```

Несмотря на то что тип значения позиции в потоке не является целым значением, целые константы можно использовать в качестве аргумента в методах установки позиции — выполняется преобразование по умолчанию. Символы в потоке нумеруются, начиная с нулевого, поэтому позиционирование в начало потока можно выполнить, например, так:

```
file.seekg(0);
```

После этого опять можно будет прочитать первый символ потока.

Методы установки относительной позиции очень похожи на функцию `fseek()` из библиотеки `<stdio>`. В классе `ios_base` определены следующие константы (см. п. п. 27.4.2.1.5 в [1]):

```
static const seekdir beg,    // позиционирование от начала потока
                  cur,      // позиционирование от текущей позиции
                  end;      // позиционирование от конца потока
```

Первый аргумент, который обычно задается целым числом (выполняется преобразование по умолчанию), является смещением (в символах) от указанной позиции. Положительное число означает смещение вперед — ближе к концу файла, отрицательное — смещение назад, к началу файла. Например, позиционироваться в конец файла можно так:

```
file.seekg(0, ios::end);
```

При позиционировании нужно следить, чтобы позиция оставалась внутри файла. Попытка позиционироваться до начала потока или после конца приводит поток в состояние `bad()`. Небольшой пример¹ показывает, как работает относительное позиционирование (листинг 14.22).

Листинг 14.22. Относительное позиционирование

```
int main()
{
    fstream strm("c:/binfiles/oonumber2.bin", ios::binary | ios::in | ios::out);
    int a = 0;
    // читаем числа из файла и выводим
    strm.seekg(2*sizeof(int), ios::beg);    // выставляем от начала
    strm.read((char *)&a, sizeof(int));    // читаем
    cout << a << ' ' << '\n';
    // записываем новое число на ту же позицию
    a = 12021;
    strm.seekp(2*sizeof(int), ios::beg);    // выставляем от начала
    strm.write((char *)&a, sizeof(int));    // записываем
    // возвращаемся, читаем снова и выводим
    strm.seekg(2*sizeof(int), ios::beg);    // выставляем от начала
    strm.read((char *)&a, sizeof(int));    // опять читаем
    cout << a << ' ' << '\n';
    return EXIT_SUCCESS;
}
```

В примере открывается существующий (см. листинг 14.7) двоичный файл в режиме чтения/записи. Оба флага (`ios::in | ios::out`) задавать обязательно, иначе

¹ В Visual Studio.NET 2003.

по умолчанию файловый поток открывается только для записи. Сначала читается третье от начала число, затем выполняется возврат в ту же позицию и записывается новое число. Результат можно наблюдать на экране.

С помощью методов позиционирования можно вычислить размер файла (листинг 14.23).

Листинг 14.23. Функция, вычисляющая длину файла

```
long filesize(istream &stream)
{
    streampos curpos = stream.tellg();    // сохраняем текущую позицию
    stream.seekg(0, ios::end);            // выставляем на конец файла
    streampos last = stream.tellg();       // получаем позицию = длина
    stream.seekg(curpos);                  // возвращаем текущую позицию.
    return last;
}
```

Несмотря на то что используются переменные типа `streampos`, в результате выполняется преобразование по умолчанию в тип `long`. Применять эту функцию можно как с двоичными файлами, так и с текстовыми, например:

```
ifstream g;
g.open("c:/textfiles/result.files");
cout << filesize(g) << endl;
g.close();
g.open("c:/binfiles/bin2.bin", ios::binary);
cout << filesize(g) << endl;
g.close();
```

Сначала вычисляется длина текстового файла `result.files`, который образовался в каталоге `textfiles` в результате вызова функции `includeFile()` (см. листинг 14.5). Затем с помощью той же функции вычисляется длина двоичного файла из листинга 14.8.

Методы позиционирования облегчают обработку двоичных файлов, записанных любыми другими средствами, например средствами языка Pascal. В C++, в отличие от языка программирования Pascal, отсутствуют типизированные файлы с записями (file of type). В языке Pascal такие файлы являются двоичными. Записи нумеруются, начиная с нуля, что обеспечивает прямой доступ к записи. Такие файлы в программе на языке Pascal по умолчанию открываются в режиме чтения/записи.

Мы вполне можем обработать такие файлы и обеспечить прямой доступ к записям по номерам средствами C++. Для этого нужно объявить в программе структуру, имеющую поля соответствующих типов, заданные в том же порядке, что в записи в программе на Pascal. Здесь нужно внимательно отнестись к проблеме выравнивания — структура в C++ должна иметь точно такой же размер, как и запись в Pascal. Далее нужно открыть файл как двоичный в режиме чтения/записи. Чтение из файла выполняется оператором

```
strm.read((char *)&record, sizeof(T));
```

А запись выполняется оператором

```
strm.write((char *)&record, sizeof(T));
```

Здесь `sizeof(T)` — размер структуры в C++, который должен совпадать с размером записи в Pascal.

Методы относительного позиционирования позволяют реализовать прямой доступ к записям этого двоичного файла. Например, позиционирование на k -ю запись для чтения выполняется оператором

```
strm.seekg(k*sizeof(T), beg);
```

Если необходимо сместиться вперед от текущей позиции на одну запись, можно воспользоваться любой из следующих инструкций:

```
// эквивалентные вызовы seekg
stream.seekg(stream.tellg() + sizeof(T)); // абсолютная позиция
stream.seekg(sizeof(T), ios::cur);       // относительная позиция
```

Сместиться назад на одну запись можно так:

```
stream.seekg(stream.tellg() - sizeof(T)); // абсолютная позиция
stream.seekg(-sizeof(T), ios::cur);       // относительная позиция
```

Буферизация

Как уже упоминалось, ввод-вывод по умолчанию является буферизованным (см. рис. 14.1). Для выходных потоков это фактически означает, что поток направляет символы буферу, а реальный вывод осуществляет уже буфер (см. п. 27.5 в [1]). Входной поток получает символы от буфера, а реальный ввод с устройства осуществляет буфер. Отменить буферизацию можно, установив флаг `unitbuf`.

Если в библиотеке `<stdio>` буфер представляет собой просто область памяти и функции ввода-вывода работают с этой памятью, то в объектно-ориентированной библиотеке буферы — это полноценные объекты. Конкретный тип буфера зависит от вида потока, но базовым типом для любых буферов является тип `streambuf` (см. п. п. 27.5.2 в [1]). Это обеспечивает единый для любых типов потоков интерфейс с буферами. С объектом типа `streambuf` можно обращаться непосредственно, используя многочисленные предоставляемые методы, например читать и записывать символы. Однако знать детали интерфейса класса `streambuf` необходимо только разработчикам систем ввода-вывода.

Каждый объект-поток содержит указатель на некоторый объект-буфер. Один из важнейших методов потока — метод `rdbuf()` (см. п. п. 27.4.4.2 в [1]), который возвращает этот указатель. С помощью данного метода можно записать функцию копирования (см. листинг 14.4) в одну строку (листинг 14.24).

Листинг 14.24. Копирование потоков

```
void filecopy(istream &in, ostream &out)
{ out << in.rdbuf(); }
```

Мы просто выводим целиком весь буфер одного потока в другой. Таким же способом мы можем вывести текстовый файл в стандартный поток `cout`, например:

```
istream f("c:/textfiles/number.txt");
cout << f.rdbuf();
```

В классах `basic_istream` и `basic_ostream` есть конструкторы, принимающие в качестве аргумента указатель на буфер. Это позволяет объявить несколько разных потоков, связанных с одним буфером. Например, мы можем объявить и настроить отдельный поток для вывода в `cout` целых в шестнадцатеричной форме:

```
ostream hout(cout.rdbuf());  
hout.setf(ios::hex, ios::basefield);  
hout.setf(ios::showbase);
```

Пример:

```
hout << "hout=" << 77 << ' ' ;  
cout << "cout=" << 77; hout << endl;
```

Эти операторы выведут на экран следующее:

```
hout=0x4d cout=77
```

С помощью того же метода `rdbuf()` выполняется и перенаправление потоков, например:

```
ofstream file("c:/textfiles/cout.txt");  
streambuf *b = cout.rdbuf();  
cout.rdbuf(file.rdbuf());  
cout << "привет от русской строки!" << endl;  
cout.rdbuf(b);  
cout << "привет от русской строки!" << endl;
```

В этом фрагменте объявляется поток `file`, связанный с текстовым файлом. Затем сохраняется указатель на буфер стандартного потока. Непосредственно перенаправление стандартного потока в файл выполняется в третьей строке. После этого вывод в `cout` направляется в файл. Затем мы восстанавливаем буфер стандартного потока, и следующий оператор демонстрирует вывод уже на экран. Примечательно, что открыв файл `cout.txt` в Блокноте, мы обнаружим, что текстовая константа записана на кириллице без всякой перекодировки!

Широкие потоки

Широкие (wide) потоки работают с широкими символами `wchar_t` (см. п. п. C.2.2.1 в [1]). Если значение `sizeof(char)` по стандарту равно 1, то размер `wchar_t` зависит от реализации (см. п. п. 5.3.3/1 в [1]). Обычно размер `sizeof(wchar_t)` равен двум (например, в системе Visual C++.NET 2003). Все стандартные классы объектно-ориентированной библиотеки реализованы и в «узком», и в широком виде. В стандартном пространстве имен `std` прописаны следующие операторы (см. п. 27.2 в [1]):

<code>typedef basic_streambuf<wchar_t></code>	<code>wstreambuf;</code>
<code>typedef basic_istream<wchar_t></code>	<code>wistream;</code>
<code>typedef basic_ostream<wchar_t></code>	<code>wostream;</code>
<code>typedef basic_iostream<wchar_t></code>	<code>wiostream;</code>
<code>typedef basic_stringbuf<wchar_t></code>	<code>wstringbuf;</code>
<code>typedef basic_istringstream<wchar_t></code>	<code>wistringstream;</code>
<code>typedef basic_ostringstream<wchar_t></code>	<code>wostringstream;</code>


```

typedef basic_stringstream<wchar_t>      wstringstream;
typedef basic_filebuf<wchar_t>           wfilebuf;
typedef basic_ifstream<wchar_t>          wifstream;
typedef basic_ofstream<wchar_t>          wofstream;
typedef basic_fstream<wchar_t>           wfstream;

```

При этом заголовки нужно подключать те же самые, что и для обычных (узких) потоков. Каждый узкий поток имеет пару — широкий поток. Например, стандартные широкие потоки обозначаются так:

- `wcin` — объект класса `wistream`, соответствующий стандартному вводу (`stdin`); по умолчанию связан с клавиатурой;
- `wcout` — объект класса `wostream`, соответствующий стандартному выводу (`stdout`); по умолчанию связан с экраном;
- `wclog` — объект класса `wostream`, соответствующий стандартному выводу для ошибок (`stderr`); по умолчанию связан с экраном;
- `wcerr` — объект класса `wostream`, соответствующий стандартному выводу для ошибок (`stderr`); по умолчанию связан с экраном; в отличие от `wclog` он не буферизуется.

Для широких потоков перегружены операции ввода (`operator>>`) и вывода (`operator<<`). В стандартных широких потоках для встроенных типов все средства форматирования, все методы работают точно так же, как и для узких. Например, ввод целой переменной и вывод ее в шестнадцатеричном виде выполняется как обычно:

```

int i;
wcin >> i;
wcout << hex << i << endl;

```

Ввод-вывод булевых значений в виде `true` и `false` тоже не отличается от обычного:

```
wcout << boolalpha << true << endl;
```

Ввод и вывод широких символов

Широкие символьные константы объявляются следующим образом:

```

wchar_t ch = L'a';
wchar_t s[] = L"Hello world!";
wstring ws = L"Hello, my students!";

```

Вывод таких англоязычных констант сложностей не представляет:

```

wcout << ch << endl;
wcout << s << endl;
wcout << ws << endl;

```

ПРИМЕЧАНИЕ

В системе Visual C++.NET 2003 в широкие потоки прекрасно можно выводить и узкие англоязычные константы. Никаких ошибок не диагностируется, и строка выводится именно так, как написана. Однако узкие переменные типа `string` и `char` выводить нельзя — возникают ошибки трансляции.

Отдельный символ можно преобразовать в широкий методом `widen()`. Обычно этот метод используют для преобразования специальных символов, например:

```
wcout << wcout.widen('\n') << endl;
```

Код широкого символа можно вывести после явного преобразования типа, например:

```
wcout << int(ch) << endl;
```

В данном случае символ `ch` не обязательно должен быть широким — выводится-то целое число.

Для вывода указателя на широкую строку его тоже необходимо приводить к типу `void *`:

```
wchar_t *p = L"Hello, world!";  
wcout << p << endl;           // вывод строки  
wcout << (void *)p << endl;    // вывод указателя
```

Все, что было сказано о вводе узких латинских символов и строк, справедливо и для широких. Ввод широкого символа можно выполнить с помощью любой формы метода `get()`, например:

```
wcin.get(ch);  
ch = wcin.get();  
wcin.read(&ch, 1);
```

В последнем случае нужно задать именно единицу, так как вводится один символ (не байт!).

Ввод строк из `wcin` тоже выполняется до пробела. Если мы хотим вводить с пробелами, то нужно использовать описанные ранее функции `getline()`:

```
wcin.getline(s, 99);           // ввод в широкий символьный массив с пробелами  
getline(wcin, ws);             // ввод в широкую строку с пробелами
```

Кириллица и локальный контекст

Для ввода-вывода символов и строк на кириллице необходимо установить кириллицу в качестве *локального контекста*, или *локаля* (locale). Локальный контекст (см. п. п. 22.1.1 в [1]) — это объект, инкапсулирующий национальные особенности внешнего представления данных вроде символа-разделителя целой и дробной частей в числах, формата даты и тому подобных «мелочей» (см. п. п. 22.1.1.1 в [1]). Заодно в локальном контексте задается и нужная кодировка. Вообще-то объекты-контексты устроены довольно сложно — в стандарте им посвящено почти 50 страниц (см. п. 22 в [1]), но мы рассмотрим только простейшие применения для работы с потоками.

Для работы с контекстами нужно прописать в программе оператор

```
#include <locale>
```

После этого можно объявлять в программе объекты-контексты, например:

```
locale american;           // установки по умолчанию
```

Установки по умолчанию ориентированы, естественно, на США. Однако ни объявление контекстов, ни стандартные американские установки нас не интересуют. Локальный контекст устанавливается для потока. Чтобы выводить символы кириллицы на консоль, мы должны установить русский локальный контекст для потока `wcout`, а чтобы вводить символ кириллицы, — русский контекст для потока `wcin`. Это делается с помощью метода `imbue()` потока (см. п. п. 27.4.2.3 в [1]), имеющего следующий прототип:

```
locale imbue(const locale& L);
```

Параметром служит объект-контекст, но его не обязательно объявлять в программе — можно использовать временный анонимный объект, например:

```
wcout.imbue(locale("rus_rus.866"));
```

В данном случае русский контекст установлен для стандартного широкого потока `wcout`. Аргумент контекста — строка `"rus_rus.866"`. Эта строка называется именем локального контекста, и ее вид зависит от реализации¹. Это имя можно получить в виде строки типа `string` с помощью метода `name()` класса `locale`:

```
locale loc ("rus_rus.866");  
cout << loc.name( ) << endl;
```

В результате на экран будет выведено следующее:

```
Russian_Russian.866
```

Это — полное имя контекста, мы же писали сокращенное. Первое слово «Russian» означает русский язык, второе — страну, число 866 — кодовую страницу.

После установки русского контекста на экран в консольное окно нормально выводятся русскоязычные сообщения, например:

```
wstring s(L"Привет!");  
wcout << L"Снова привет!" << endl;  
wcout << s << endl;
```

Однако правильно выводятся только широкие русскоязычные строковые константы — в отличие от англоязычных строковых констант, узкие русскоязычные строковые константы выводятся неправильно.

ПРИМЕЧАНИЕ

Можно установить русский контекст и для узких потоков. Однако в системе Windows консольные программы работают в специальном консольном окне, для которого система по умолчанию устанавливает кодовую страницу 866 как для ввода, так и для вывода. Когда же мы набираем текст программы в окне интегрированной среды, то установлена кодовая страница 1251 и русскоязычные константы записываются в программу в этой кодировке. Поэтому даже при установлении русского контекста для потока `cout` русскоязычные текстовые константы выводятся неверно.

¹ В системе Visual C++.NET 2003 под управлением Windows.

Для ввода кириллицы нужно точно таким же способом задать российский контекст для входного широкого потока `wcin`:

```
wcin.imbue(locale("rus_rus.866"));
```

После этого можно будет вводить русскоязычные строки и символы обычными способами, которые мы уже рассматривали, например:

```
wstring s;
getline(wcin, s);           // ввод широкой строки с пробелами
wchar_t g[100];
wcin.getline(g, 99);        // ввод символьного массива с пробелами
wcin.get(ch);               // ввод широкого символа
ch = wcin.get();            // ввод широкого символа
```

Все введенные символы нормально выводятся на экран.

Текущий локальный контекст можно сохранить как объект-локаль с помощью метода `getloc()`, который имеет прототип

```
locale getloc() const;
```

Можно установить «классический» локальный контекст `C` с помощью метода

```
static const locale& classic( );
```

Можно установить некоторый контекст по умолчанию методом

```
static locale global(const locale& L);
```

Применять эти методы можно так:

```
locale current = wcout.getloc(); // сохранение текущего контекста
wcout.imbue(locale::classic()); // установка классического контекста
wcout.imbue(locale("C"));       // тоже установка классического контекста
wcout.imbue(current);           // восстановление прежнего контекста
locale::global(current);        // установка контекста по умолчанию
```

Широкие строковые потоки

Не представляет особого труда перейти от обычных строковых потоков к широким. Перепишем пример из листинга 14.16, используя широкие потоки и символы кириллицы (листинг 14.25).

Листинг 14.25. Широкие строковые потоки

```
#include <iostream>
#include <sstream>
#include <locale>
using namespace std;
int main()
{ char s[] = "computer", c = 'L';
  int i = 35;
  float fp = 1.7320534f;
  wstringstream os;
  // установка русского контекста для строкового потока и для вывода
  os.imbue(locale("rus_rus.866"));
```

Листинг 14.25 (продолжение)

```

wcout.imbue(locale("rus_rus.866"));
/* Форматирование и вывод данных */
os << L"Строка: " << s << L"; "
  << L"Символ: " << c << L"; "
  << L"Целое: " << i << L"; "
  << L"Дробное: " << fp << L"; "
  << endl;
wcout << L"Вывод:" << endl << os.str() << endl;
wcout << L"Количество символов=" << os.str().length() << endl;
return EXIT_SUCCESS;
}

```

Для того чтобы осуществлять вывод по-русски, мы использовали широкие строковые константы и установили русский контекст для строкового потока и для стандартного выходного потока. Выводимый на экран результат выполнения программы:

Вывод:

Строка: computer; Символ: L; Целое: 35; Дробное: 1,73205;
Количество символов=59

Обратите внимание, что ни строку «computer», ни символ «L» мы не стали писать в широком виде — все прекрасно сработало и так.

Широкие файловые потоки

Использовать широкие текстовые потоки для работы с текстовыми файлами не представляет труда. Напишем вариацию на тему примера из листингов 14.3 и 14.4 (листинг 14.26).

Листинг 14.26. Текстовые файлы с широкими текстовыми потоками

```

#include <fstream>
#include <iostream> // требуется в Visual C++.NET 2003
#include <ctime>
using namespace std;
// функция копирования файлов
void filecopy (wifstream &in, wofstream &out)
{ wchar_t ch;
  while(in.get(ch)) // читать все символы, в том числе пробельные
    out.put(ch);
}

int main()
{
  srand((unsigned)time(NULL)); // датчик случайных чисел
  wofstream strm; // выходной поток-объект
  strm.open("c:/textfiles/woonumber.txt"); // открываем
  if (strm.is_open()) // проверка открытия
  { for(int i = 0; i < 10; i++)
      strm << rand()%10 << endl; // выводим 10 чисел
    strm.close(); // закрываем выходной поток-файл
    // суммирование чисел записанных в файле
    // открываем тот же текстовый файл для чтения
    wifstream strm("c:/textfiles/woonumber.txt");

```

```

    if (strm)
    {
        int number, summa = 0, count = 0;
        while(strm >> number)
        {
            ++count;
            summa+=number;
        }
        wcout << summa << L"; " << count;
        strm.close();
    }
}

wifstream instrm ("c:/textfiles/woonumber.txt");
wofstream outstrm("c:/textfiles/woonumber.new");
if (instrm) filecopy(instrm, outstrm);
instrm.close();
outstrm.close();
return EXIT_SUCCESS;
}

```

Сначала программа объявляет широкий поток `strm`, который при открытии связывается с текстовым файлом `woonumber.txt`. Затем в файл выводятся 10 случайных чисел, и поток закрывается. На этой стадии в нашем каталоге `TextFiles` можно обнаружить текстовый файл размером 30 байт. Далее тот же файл открывается как входной, числа читаются и суммируются — все работает точно так же, как и в примере из листинга 14.3. Затем существующий файл копируется с помощью функции копирования `filecopy()`, параметрами которой являются широкие потоки. На этой стадии программы в каталоге `TextFiles` можно увидеть 2 файла: `woonumber.txt` и `woonumber.new`.

Широкие потоки в функции `filecopy()` использовать совершенно необязательно — вполне корректно работают и обычные, открытые в двоичном режиме:

```

ifstream instrm ("c:/textfiles/woonumber.txt", ios::binary);
ofstream outstrm("c:/textfiles/woonumber.new", ios::binary);

```

При этом не происходит преобразования формата символа новой строки `'\n'`. То есть считывание и запись можно выполнять побайтно — независимо от того, сколько и каких байтов в файл записано.

Размеры файлов показывают, что символы-цифры выводятся в «широкий» файл в однобайтовом виде¹. Это, кстати, явно указано в стандарте (см. п. п. 28.8.1/2 в [1]). Проверим, как выводятся широкие символы латиницы и кириллицы. Для этого создадим два широких выходных файловых потока. Для «чистоты» эксперимента установим для одного из них русский контекст и выведем в поток русскоязычную и англоязычную константу. Для второго потока контекст устанавливать не будем. Добавим в конец предыдущего примера следующие строки:

```

wofstream rstrm("c:/textfiles/woorussian.txt");
rstrm.imbue(locale("rus_rus.866"));
rstrm << L"Проверка вывода русских букв в файл" << endl;
rstrm << L"aaaaaaaa vvvvvv rrrrrr cccc i ffff" << endl;
rstrm.close();

```

¹ Программа работает совершенно одинаково и в Visual C++.NET 2003, и в C++ Builder 6.

```
wofstream estrm("c:/textfiles/woenglish.txt");
estrm << L"aaaaaaaa vvvvvv rrrrrr cccc i ffff" << endl;
estrm << L"Проверка вывода русских букв в файл" << endl; // не выводятся
estrm.close();
```

И в «русский», и в «английский» потоки символы выводятся в однобайтной кодировке. Если русский контекст не установлен, то буквы кириллицы просто не выводятся в файл — как и все остальные символы, расположенные следом! Ничего не меняется, если мы используем вместо широкой константы-строки широкую переменную:

```
wstring ws = L"Проверка вывода русских букв в файл";
rstrm << ws << endl;
```

Размер русскоязычной строки в файле остается тем же самым.

ПРИМЕЧАНИЕ

Если широкие потоки, связанные с текстовыми файлами, работают правильно, то при связывании их с двоичными файлами возникают малопонятные проблемы. Во всяком случае, и в Visual C++.NET 2003, и в C++ Builder 6 методы `read()` и `write()` для широких потоков работают неправильно.

Резюме

Ввод-вывод в C++ основан на концепции потоков, которые были реализованы еще в C в библиотеке `<stdio.h>`. Библиотека `<iostream>` предоставляет объектно-ориентированную реализацию потоков. Объектно-ориентированный подход надежнее и проще в использовании.

В C++ реализован достаточно богатый набор средств ввода-вывода: стандартные консольные потоки, строковые потоки, файловые потоки. Объектно-ориентированные потоки более универсальны, чем процедурные, и «умеют» работать со строками типа `string`.

Для ввода-вывода информации в библиотеке перегружены операции сдвига влево и вправо. Кроме того, реализовано множество методов обмена данными между программой и потоком. В объектно-ориентированной библиотеке реализовано значительное количество средств форматирования: флаги, манипуляторы и методы. Все средства форматирования работают единообразно для всех видов потоков: стандартных, строковых и файловых.

Объектно-ориентированные средства ввода-вывода являются расширяемыми, то есть мы можем создавать собственные средства форматирования на основе стандартных. Перегрузка операций ввода и вывода обеспечивает их единообразный вид как для встроенных типов, так и для реализованных классов.

Строковые потоки являются универсальным средством преобразования чисел в строки и обратно. Строковые потоки бывают входными, выходными и двусторонними. Для строковых потоков работают методы, обеспечивающие произвольный доступ к любому байту потока.

Файловые потоки бывают входными, выходными и двунаправленными. Файловые потоки могут быть форматируемыми и неформатируемыми. Файловые потоки связываются с внешними файлами на диске либо при создании потока, либо при открытии. Файловый поток может быть открыт в текстовом режиме (и связан с текстовым файлом) или в двоичном режиме (и связан с двоичным файлом). При уничтожении или закрытии потока связь разрывается.

Для файловых потоков работают методы, обеспечивающие произвольный доступ к любому байту внешнего файла. В частности, с помощью этих методов легко узнать размер файла.

Потоки по умолчанию буферизованы, и метод `gdbuf()` часто используется для копирования и перенаправления потоков.

Помимо обычных потоков, в библиотеке реализованы широкие потоки, которые работают с широкими символами типа `wchar_t`.

Контрольные вопросы

1. Что такое «поток»? Дайте определение.
2. Как классифицируются потоки, реализованные в библиотеках ввода-вывода C++?
3. Что такое буферизация и зачем она нужна?
4. Какие библиотеки ввода-вывода реализованы в C++ и чем они различаются?
5. Перечислите стандартные потоки и объясните их назначение.
6. Зачем нужен процесс форматирования и когда он выполняется?
7. Каковы особенности ввода строк?
8. Каким образом ограничить набор вводимых символов при вводе?
9. Объясните, для чего нужны строковые потоки. Почему строковые потоки — всегда форматируемые?
10. Можно ли использовать тип `string` (и каким образом) со строковыми потоками?
11. Объясните, в чем различие между текстовым и двоичным файлами.
12. Объясните, что означает «открыть» файл и «закрыть» файл?
13. Каким образом внешний файл связывается с потоком?
14. Можно ли один и тот же поток связать с разными файлами? А один и тот же файл — с разными потоками?
15. В каких случаях необходимо следить за ситуацией конца файла? Каким способом это делается?
16. Можно ли текстовый файл открыть как двоичный? А двоичный — как текстовый?
17. Какие операции и методы ввода-вывода требуются для обмена с текстовыми файлами?

18. Перечислите методы ввода-вывода для работы с двоичными файлами.
19. Объясните, что означает «перенаправление» потока? Какие потоки можно перенаправлять и куда?
20. В чем преимущества объектно-ориентированной библиотеки ввода-вывода по сравнению с процедурной?
21. В каких состояниях может находиться поток? Каким образом отслеживается состояние конца потока?
22. Каким образом строковые потоки можно использовать для ограничения ширины поля ввода?
23. Перечислите средства форматирования объектно-ориентированной библиотеки.
24. Каким образом ввести строку типа `string` с пробелами?
25. Каково назначение флагов форматирования? Какие средства реализованы в библиотеке для работы с флагами форматирования?
26. Что такое «манипулятор»? В чем преимущества манипуляторов перед флагами форматирования?
27. Как связываются файлы с потоками в объектно-ориентированной библиотеке?
28. Можно ли файлы, записанные независимо от C++, прочитать объектно-ориентированными средствами ввода-вывода C++? А наоборот?
29. Перечислите режимы открытия объектно-ориентированных файловых потоков. Каким образом комбинируются режимы открытия файловых потоков?
30. Обязательно ли закрывать файл, связанный с объектно-ориентированным файловым потоком? А открывать?
31. Каким образом открыть файловый поток для чтения и записи одновременно?
32. Как открыть файловый поток для дозаписи?
33. Можно ли вывести значение переменной в двоичном виде и как это сделать?
34. Разрешается ли наследовать от классов библиотеки ввода-вывода?
35. Каким образом можно перенаправить объектно-ориентированный поток?
36. Как используется буфер потока для копирования потока?
37. Какими операциями выполняется форматированный ввод в файловые потоки и вывод из них? А неформатированный?
38. Реализованы ли в объектно-ориентированной библиотеке средства прямого доступа к файловым потокам?
39. С какими объектно-ориентированными потоками разрешается, а с какими не разрешается использовать средства прямого доступа?
40. Покажите, каким образом можно выполнить перегрузку операций ввода-вывода для нового типа данных.
41. Как выполняется обработка ошибок ввода-вывода в объектно-ориентированной библиотеке?
42. Какое стандартное исключение генерируется при ошибках ввода-вывода? Обязательно ли оно генерируется?

43. Чем стандартные широкие потоки отличаются от узких?
44. Что такое «локальный контекст», и каково его назначение?
45. Как установить русскоязычный шрифт при выводе в консольное окно?
46. Чем отличается ввод-вывод широких файловых потоков от узких?

Упражнения

1. Написать функцию, которая записывает в текстовый файл 100 случайных целых чисел в диапазоне от -30 до $+70$ по одному в строке. Используя этот файл как входной, сформировать выходной текстовый файл, добавив к каждому числу входного файла последнее число файла.
2. Выполнить упражнение 1, сделав входной файл двоичным. Для получения последнего числа файла использовать методы прямого доступа. Для проверки входного файла написать функцию вывода двоичного файла на экран.
3. Выполнить упражнение 1 с широкими потоками.
4. Выполните упражнение 1 с одним двоичным файлом, перезаписав файл на месте. Для проверки результата написать функцию вывода двоичного файла на экран.
5. Добавить в класс `ListPerson` (см. упражнение 10 в главе 4) метод `save()`, записывающий данные на диск, и метод `load()`, обеспечивающий загрузку данных, сохраненных методом `save()`. Методы должны получать файловый поток в качестве параметра.
6. Выполнить упражнения 2 и 3 из главы 11, добавив в классы `TSimpleArray` и `TDeque` метод `save()`, сохраняющий данные в файл, и метод `load()`, обеспечивающий загрузку данных, записанных методом `save()`. Методы должны получать файловый поток в качестве параметра.
7. Запись о преподаваемой дисциплине представляется следующей структурой: код дисциплины в учебном плане, наименование дисциплины, фамилия преподавателя, код группы, количество студентов в группе, количество часов лекций, количество часов практики, наличие курсовой работы, вид итогового контроля (зачет или экзамен). Зачет — 0,35 часа на одного студента; экзамен — 0,5 часа на студента. Написать функцию, с помощью которой осуществляется первичный ввод информации с клавиатуры в двоичный файл. Написать функцию, преобразующую данные исходного файла в текстовый файл в виде ведомости с графой общих итогов. С файлом выполняются операции поиска, добавления, удаления и замены записи. Удаление и замену записей выполнять перезаписью исходного файла в новый. Поиск вести по фамилии преподавателя, коду группы, наличию курсовой работы, виду итогового контроля. При поиске сохранять выбранные записи в новом файле.
8. Выполнить упражнение 7, написав вместо функции класс, обеспечивающий нужные операции. Для форматирования записи выходного текстового файла написать класс-манипулятор. Использовать широкие потоки.

9. Выполнить упражнение 7, реализовав замену записи по месту старой записи без перезаписи файла. Операцию удаления выполнять тоже без перезаписи файла. Для этого добавить в структуру поле-признак удаления (`true` — удалена). Реализовать операцию `раск()`, выполняющую уплотнение файла посредством перезаписи исходного файла с физическим изъятием помеченных на удаление записей.
10. Для упражнения 7 разработать класс `Index`, выполняющий построение простейшего индексного массива. Индексный массив — динамический, состоящий из пар «ключ-адрес». Массив должен быть отсортирован по возрастанию ключа. Ключ — это одно из полей, по которому производится поиск. Поиск нужной записи выполняется по индексу. Класс должен обеспечить индексирование по любому из требуемых полей класса. Класс должен включать в себя методы `save()` и `load()`.
11. Выполнить упражнение 1, используя строковые потоки. Исходный файл является текстовым, выходной — двоичным. Входное число считывается как строка и преобразуется во внутренний формат с помощью входного строкового потока.
12. Выполнить упражнение 11 при условии, что входной файл является двоичным, а выходной — текстовым. Выходное число преобразуется в строку с помощью выходного строкового потока.

Приложение

Строки в C++

Строки являются, пожалуй, самым используемым типом данных (возможно, после целых чисел) при программировании на C++. Мы неоднократно задействовали строки в различных примерах программ. В C++ в настоящее время поддерживаются два вида символьных строк: строки в стиле C и строки в стиле C++. Строки в стиле C — это символьные массивы, строки в стиле C++ — это строки класса `basic_string<>`, который определен в стандарте (см. п. 21 в [1]).

Строковые константы являются строками в стиле C и имеют тип константного символьного массива:

- `const char[]` — для обычных символов;
- `const wchar_t[]` — для широких символов.

Обычные строковые константы записываются как последовательность символов в кавычках, например:

```
"строка"  
"это очень длинная строка с управляющим символом в конце\n"
```

Широкие строковые константы пишутся с символом `L` перед последовательностью символов, например:

```
L"широкая строка"  
L"это очень длинная строка с управляющим символом в конце\n"
```

Символьные константы по умолчанию завершаются нулевым символом.

Обработка строк в стиле C

Обработка строк, представляющих собой символьные массивы, выполняется функциями, прототипы которых находятся в системном заголовке `<cstring>`. Этот заголовок подключается к программе следующим образом:

```
#include <cstring>
```

В стандартное пространство имен в Visual C++.NET 2003 входят следующие функции (всего 21 имя) для обработки обычных символьных массивов (см. табл. 47 в [1]):

```
namespace std {
    using ::memchr;      using ::memcmp;
    using ::memcpy;      using ::memmove;
    using ::memset;
    using ::size_t;      // имя типа, а не функции
    using ::strcat;      using ::strchr;
    using ::strcmp;      using ::strcoll;
    using ::strcpy;      using ::strcspn;
    using ::strlen;      using ::strncat;
    using ::strncmp;     using ::strncpy;
    using ::strpbrk;     using ::strrchr;
    using ::strspn;      using ::strstr;
    using ::strtok;      using ::strxfrm;
}
```

Имена функций (тоже 21 имя) для обработки широких символьных массивов, входящие в стандартное пространство имен (см. табл. 48 в [1]), определены в заголовке <wchar>:

```
namespace std {
    using ::wscat;      using ::wcschr;
    using ::wscmp;      using ::wcscoll;
    using ::wscpy;      using ::wcscspn;
    using ::wcslen;     using ::wcsncat;
    using ::wcsncmp;    using ::wcsncpy;
    using ::wcpbrk;     using ::wcsrchr;
    using ::wcssp;      using ::wcsstr;
    using ::wcstok;     using ::wcsxfrm;
    using ::wmemchr;    using ::wmemcmp;
    using ::wmemcpy;    using ::wmemmove;
    using ::wmemset;
}
```

ПРИМЕЧАНИЕ

В разных интегрированных средах, несмотря на наличие стандартов C и C++, реализуется много нестандартных функций для работы со строками. В системе Visual C++.NET 2003 такие функции всегда начинаются с символа подчеркивания. В системе Borland C++ Builder 6 имена многих функций, не регламентированных стандартом C, никак не выделены.

Строки в стиле C обрабатываются набором функций, которые «достались» C++ по наследству от C. При использовании строковых констант в качестве аргументов функций обычно выполняют преобразование из типа символьного массива в тип указателя на символ:

```
const char[]->const char*;
const wchar_t[]->const wchar_t*;
```

Функции для работы с массивами типа `char[]`

Далее перечислены пять функций, оперирующих произвольными областями памяти, в частности, выполняющих операции с символьными массивами, не отслеживая завершающий нулевой байт.

- ❑ Функция `memchr()` возвращает указатель на первое вхождение младшего байта аргумента `ch` в массиве `buf` длиной `count` байтов:

```
const void* memchr(const void* buf, int ch, size_t count);  
void* memchr(const void* buf, int ch, size_t count);
```

- ❑ Функция `memcmp()` сравнивает первые `count` байтов символьных массивов `buf1` и `buf2` и длиной `count` байтов в лексикографическом порядке:

```
int memcmp(const void *buf1, const void *buf2, size_t count);
```

Функция возвращает:

- `-1`, если содержимое `buf1` меньше содержимого `buf2`;
 - `0`, если содержимое `buf1` совпадает с содержимым `buf2`;
 - `+1`, если содержимое `buf1` больше содержимого `buf2`.
- ❑ Функция `memcpy()` копирует `count` байтов из символьного массива `src` в символьный массив `dest` и возвращает указатель на `dest`:

```
void * memcpy(void *dest, const void *src, size_t count);
```

- ❑ Функция `memmove()` копирует `count` байтов из символьного массива `src` по адресу `dest`:

```
void * memmove(void *dest, const void *src, size_t count);
```

Массивы могут перекрываться. Функция возвращает указатель на `dest`.

- ❑ Функция `memset()` заполняет первые `count` байтов символьного массива `dest` символом, взятым из младшего байта `ch`:

```
void * memset(void *dest, int ch, size_t count);
```

Функция возвращает указатель на `dest`.

Следующие функции выполняют операции с символьными массивами как со строками в стиле C и учитывают наличие нулевого байта в конце строки.

- ❑ Функция `strlen()` возвращает количество символов в строке `s` (длину строки) без учета нулевого байта:

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

- ❑ Функция `strcat()` добавляет строку `Source` в конец строки с `Dest` и возвращает указатель на `Dest`:

```
char * strcat(char *Dest, const char *Source);
```

Символьный массив `Dest` должен иметь размер не менее `strlen(Dest) + strlen(Source) + 1` байтов.

- ❑ Функция `strncat()` добавляет не более `n` символов из строки `Source` в конец строки `Dest` и возвращает указатель на `Dest`:

```
char * strncat(char *Dest, const char *Source, size_t n);
```

В массиве `Dest` должно быть не менее `strlen(Dest) + n + 1` байтов.

- ❑ Функция `strchr()` выполняет поиск символа с кодом `ch` слева направо в строке `string`;

```
const char* strchr(const char* string, int ch);  
char* strchr(const char* string, int ch);
```

Функция возвращает указатель на первое вхождение символа. Если символ не обнаруживается, возвращает `NULL` (нулевой указатель).

- ❑ Функция `strrchr()` делает то же самое, что и функция `strchr()`, но справа налево:

```
const char* strrchr(const char* string, int ch);  
char* strrchr(const char* string, int ch);
```

- ❑ Функция `strstr()` выполняет поиск строки `Search` в строке `string`:

```
const char* strstr(const char* string, const char* Search);  
char* strstr(const char *string, const char *Search);
```

Функция возвращает указатель на первое вхождение `Search`. Если строка не обнаружена, то возвращает нулевой указатель (`NULL`).

- ❑ Функция `strcspn()` возвращает указатель на первое вхождение любого символа из строки `CharSet` в строке `string` или `NULL`, если такого символа не обнаружено:

```
size_t strcspn(const char *string, const char *CharSet);
```

- ❑ Функция `strspn()` возвращает указатель на первый же символ из строки `CharSet`, который не входит в строку `string`, или `NULL` в противном случае:

```
size_t strspn(const char *string, const char *CharSet);
```

- ❑ Функция `strpbrk()` возвращает указатель на символ, являющийся первым вхождением любого из символов строки `CharSet` в строку `str`:

```
const char* strpbrk(const char* str, const char* CharSet);  
char* strpbrk(const char *str, const char *CharSet);
```

Если символ не найден, функция возвращает `NULL`.

- ❑ Функция `strtok()` возвращает указатель на следующую лексему из строки `Token`, отделенную любым из символов-разделителей строки `Delim`:

```
char * strtok(char *Token, const char *Delim);
```

Первый вызов и последующие вызовы различаются. Следующий пример показывает применение этой функции:

```
#include <cstring>  
#include <iostream>
```

```

using namespace std;
int main()
{
    char input[100] = "tokens,teacher,count,word";
    char *p = strtok(input, ",");          // ищет первую запятую
    if (p) cout << p << endl;             // выводит "abc"
    while(p!=NULL)
    {
        p = strtok(NULL, ",");            // ищет следующую запятую
        if (p) cout << p << endl;        // выводит следующую лексему
    }
    return 0;
}

```

Программа выведет на экран следующее:

```

tokens
teacher
count
word

```

Функция `strtok()` чрезвычайно полезна для выделения из строки, например, идентификаторов.

- ❑ Функция `strcpy()` копирует строку `Source` в другую строку `Dest` и возвращает указатель на `Dest`:

```
char * strcpy(char *Dest, const char *Source);
```

В массиве `Dest` должно быть не менее `strlen(Source) + 1` байтов.

- ❑ Функция `strncpy()` копирует не более чем `count` символов из строки `Source` в другую строку `Dest` и возвращает `Dest`:

```
char * strncpy(char *Dest, const char *Source, size_t count);
```

В массиве `Dest` должно быть не менее `n + 1` байтов.

- ❑ Функция `strcmp()` лексикографически сравнивает строки `string1` и `string2`:

```
int strcmp(const char *string1, const char *string2);
```

Функция возвращает:

- `-1`, если содержимое `string1` меньше содержимого `string2`;
- `0`, если содержимое `string1` равно содержимому `string2`;
- `+1`, если содержимое `string1` больше содержимого `string2`.

- ❑ Функция `strncmp()` сравнивает строку `string1` и первые `n` символов строки `string2`:

```
int strncmp(const char *string1, const char *string2, size_t n);
```

Функция возвращает результат аналогично функции `strcmp()`.

Представленные далее функции требуют установки локального контекста с помощью C-функции `setlocale()`.

- ❑ Функция `strcoll()` сравнивает строки как функция `strcmp()`, учитывая параметры локализации:

```
int strcoll(const char *string1, const char *string2);
```


- Функция `strxfrm()` преобразует строку `Source` и помещает ее в строку `Dest` на основе текущего локального контекста:

```
size_t strxfrm(char *Dest, const char *Source, size_t count);
```

Преобразуется не более `count` символов. Функция возвращает полученную длину строки `Dest` без учета завершающего символа.

Функции для работы с массивами типа `wchar_t[]`

Следующие функции выполняют обработку «широких» символьных массивов и «широких» строк в стиле C.

- Функция `wscat()` аналогична функции `strcat()`:

```
wchar_t * wscat(wchar_t *Dest, const wchar_t *Source);
```

- Функция `wcschr()` аналогична функции `strchr()`:

```
const wchar_t* wcschr(const wchar_t* string, wchar_t c);  
wchar_t* wcschr(const wchar_t* string, wint_t c);
```

- Функция `wscmp()` аналогична функции `strcmp()`:

```
int wscmp(const wchar_t *string1, const wchar_t *string2);
```

- Функция `wscoll()` аналогична функции `strcoll()`:

```
int wscoll(const wchar_t *string1, const wchar_t *string2);
```

- Функция `wscopy()` аналогична функции `strcpy()`:

```
wchar_t * wscopy(wchar_t *Dest, const wchar_t *Source);
```

- Функция `wscspn()` аналогична функции `strcspn()`:

```
size_t wscspn(const wchar_t *string, const wchar_t *CharSet);
```

- Функция `wcslen()` аналогична функции `strlen()`:

```
size_t wcslen(const wchar_t *string);
```

- Функция `wcsncat()` аналогична функции `strncat()`:

```
wchar_t * wcsncat(wchar_t *Dest, const wchar_t *Source, size_t n);
```

- Функция `wcsncmp()` аналогична функции `strncmp()`:

```
int wcsncmp(const wchar_t *string1, const wchar_t *string2, size_t n);
```

- Функция `wcsncpy()` аналогична функции `strncpy()`:

```
wchar_t * wcsncpy(wchar_t *Dest, const wchar_t *Source, size_t n);
```

- Функция `wspbrk()` аналогична функции `strpbrk()`:

```
const wchar_t* wspbrk(const wchar_t* string, const wchar_t* CharSet);  
wchar_t* wspbrk(const wchar_t* string, const wchar_t* CharSet);
```

- Функция `wcsrchr()` аналогична функции `strrchr()`:

```
const wchar_t* wcsrchr(const wchar_t* string, wchar_t ch);  
wchar_t* wcsrchr(const wchar_t* string, wchar_t ch);
```

- Функция `wcsspn()` аналогична функции `strspn()`:
`size_t wcsspn(const wchar_t *string, const wchar_t *CharSet);`
- Функция `wcsstr()` аналогична функции `strstr()`:
`const wchar_t* wcsstr(const wchar_t* string, const wchar_t* CharSet);`
`wchar_t* wcsstr(const wchar_t* string, const wchar_t* CharSet);`
- Функция `wcstok()` аналогична функции `strtok()`:
`wchar_t * wcstok(wchar_t *Token, const wchar_t *Delim);`
- Функция `wcsxfrm()` аналогична функции `strxfrm()`:
`size_t wcsxfrm(wchar_t *Dest, const wchar_t *Source, size_t n);`
- Функция `wmemcmp()` аналогична функции `memcmp()`:
`int wmemcmp(const wchar_t * buf1, const wchar_t * buf2, size_t n);`
- Функция `wmemchr()` аналогична функции `memchr()`:
`const wchar_t* wmemchr(const wchar_t * buf, wchar_t ch, size_t n);`
`wchar_t* wmemchr(const wchar_t * buf, wchar_t ch, size_t n);`
- Функция `wmemcmp()` аналогична функции `memcmp()`:
`wchar_t * wmemcmp(wchar_t *s1, const wchar_t *s2, size_t n);`
- Функция `wmemmove()` аналогична функции `memmove()`:
`wchar_t * wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);`
- Функция `wmemset()` аналогична функции `memset()`:
`wchar_t * wmemset(wchar_t *s, wchar_t c, size_t n);`

Строки в стиле C++

При разработке стандарта, наконец, было покончено с «самодеятельностью» фирм-производителей и программистов: стандартная библиотека шаблонов содержит полноценный строковый класс. В стандарте строкам посвящена отдельная глава 21 — настолько востребован этот тип данных в программировании.

Все строковые типы и функции определены в библиотеке `<string>`, которую мы уже неоднократно упоминали и подключаем:

```
#include <string>
```

Здесь определяется базовый шаблон `basic_string`. Так как в C++ два символьных типа, то определены и две специализации базового шаблона:

```
typedef basic_string <char> string;  
typedef basic_string <wchar_t> wstring;
```

Все различия, связанные с типом символов, собраны в классе-шаблоне свойств символов `char_traits<>`, который является вторым параметром базового шаблона `basic_string`. Этот класс свойств включает определения типов и наиболее

часто используемые операции с последовательностями символов. Естественно, в библиотеке присутствуют специализации свойств и для обычных символов типа `char` (см. п. п. 21.1.3.1 в [1]), и для широких символов типа `wchar_t` (см. п. п. 21.1.3.2 в [1]). Стандарт определяет только интерфейсы классов свойств, оставляя реализацию на усмотрение разработчиков компилятора.

Инкапсуляция различий в классе свойств приводит к тому, что пользователь практически не замечает разницы при манипулировании обычными и широкими строками. Единственное различие заключается в объявлении типа строки: `string` и `wstring`. Все операции со строкой совершенно не зависят от типа символов. Базовый шаблон¹ выглядит следующим образом:

```
template <class charT,                // тип символов
          class traits = char_traits<charT>, // класс свойств
          class Allocator = allocator<charT> > // распределитель памяти
class basic_string {
public:
    // Типы
    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;
    typedef typename Allocator::pointer iterator;
    typedef typename Allocator::const_pointer const_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    static const size_type npos = -1;
    // Конструкторы и деструктор
    explicit basic_string(const Allocator& = Allocator());
    basic_string(const basic_string<charT, traits, Allocator>&);
    basic_string(const basic_string&, size_type, size_type = npos,
                 const Allocator& a = Allocator());
    basic_string(const charT*, size_type, const Allocator& = Allocator());
    basic_string(const charT*, const Allocator& = Allocator());
    basic_string(size_type, charT, const Allocator& = Allocator());
    template <class InputIterator>
    basic_string(InputIterator, InputIterator, const Allocator& = Allocator());
    ~basic_string();
    // Операция присваивания
    basic_string& operator=(const basic_string&);
    basic_string& operator=(const charT*);
    basic_string& operator=(charT);
    // Итераторы
    iterator begin();
    const_iterator begin() const;
```

¹ Именно этот класс послужил основой для разработки класса `TString` (см. листинг 4.2).

```

iterator      end();
const_iterator end() const;
reverse_iterator      rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator      rend();
const_reverse_iterator rend() const;
// Размеры
size_type      size() const;
size_type      length() const;
size_type      max_size() const;
void      resize(size_type, charT);
void      resize(size_type);
size_type      capacity() const;
void      reserve(size_type = 0);
bool      empty() const;
// Доступ к элементам
const_reference operator[](size_type) const;
reference      operator[](size_type);
const_reference at(size_type) const;
reference      at(size_type);
// Сцепление строк
basic_string& operator+=(const basic_string&);
basic_string& operator+=(const charT*);
basic_string& operator+=(charT);
basic_string& append(const basic_string&);
basic_string& append(const basic_string&, size_type, size_type);
basic_string& append(const charT*, size_type);
basic_string& append(const charT*);
basic_string& append(size_type, charT);
template<class InputIterator>
    basic_string& append(InputIterator, InputIterator);
// Присваивание строк
basic_string& assign(const basic_string&);
basic_string& assign(const basic_string&, size_type, size_type);
basic_string& assign(const charT*, size_type);
basic_string& assign(const charT*);
basic_string& assign(size_type, charT);
template<class InputIterator>
    basic_string& assign(InputIterator, InputIterator);
// Модификаторы
basic_string& insert(size_type, const basic_string&);
basic_string& insert(size_type, const basic_string&, size_type, size_type);
basic_string& insert(size_type, const charT*, size_type);
basic_string& insert(size_type, const charT*);
basic_string& insert(size_type, size_type, charT);
iterator insert(iterator, charT = charT());
void insert(iterator, size_type, charT);
template<class InputIterator>
    void insert(iterator, InputIterator, InputIterator);
basic_string& erase(size_type = 0, size_type= npos);
iterator erase(iterator);
iterator erase(iterator, iterator);
basic_string& replace(size_type, size_type, const basic_string&);
basic_string& replace(size_type, size_type, const basic_string&,
    size_type, size_type);

```

```

basic_string& replace(size_type, size_type, const charT*, size_type);
basic_string& replace(size_type, size_type, const charT*);
basic_string& replace(size_type, size_type, size_type, charT);
basic_string& replace(iterator, iterator, const basic_string&);
basic_string& replace(iterator, iterator, const charT*, size_type);
basic_string& replace(iterator, iterator, const charT*);
basic_string& replace(iterator, iterator, size_type, charT);
template<class InputIterator>
    basic_string& replace(iterator, iterator, InputIterator, InputIterator);
size_type copy(charT*, size_type, size_type = 0) const;
void swap(basic_string<charT, traits, Allocator>&);
// получение символического массива из строки    const charT* c_str() const;
const charT* data() const;
const allocator_type& get_allocator() const;
// Поиск
size_type find(const basic_string&, size_type = 0) const;
size_type find(const charT*, size_type, size_type) const;
size_type find(const charT*, size_type = 0) const;
size_type find(charT, size_type = 0) const;
size_type rfind(const basic_string&, size_type = npos) const;
size_type rfind(const charT*, size_type, size_type) const;
size_type rfind(const charT*, size_type = npos) const;
size_type rfind(charT, size_type = npos) const;
size_type find_first_of(const basic_string&, size_type = 0) const;
size_type find_first_of(const charT*, size_type, size_type) const;
size_type find_first_of(const charT*, size_type = 0) const;
size_type find_first_of(charT, size_type = 0) const;
size_type find_last_of(const basic_string&, size_type = npos) const;
size_type find_last_of(const charT*, size_type, size_type) const;
size_type find_last_of(const charT*, size_type = npos) const;
size_type find_last_of(charT, size_type = npos) const;
size_type find_first_not_of(const basic_string&, size_type = 0) const;
size_type find_first_not_of(const charT*, size_type, size_type) const;
size_type find_first_not_of(const charT*, size_type = 0) const;
size_type find_first_not_of(charT, size_type = 0) const;
size_type find_last_not_of(const basic_string&, size_type = npos) const;
size_type find_last_not_of(const charT*, size_type, size_type) const;
size_type find_last_not_of(const charT*, size_type = npos) const;
size_type find_last_not_of(charT, size_type = npos) const;
// Выделение подстроки
basic_string substr(size_type = 0, size_type = npos) const;
// Сравнение
int compare(const basic_string&) const;
int compare(size_type, size_type, const basic_string&) const;
int compare(size_type, size_type, const basic_string&,
            size_type, size_type) const;
int compare(size_type, size_type, charT*) const;
int compare(charT*) const;
int compare(size_type, size_type, const charT*, size_type) const;
};

```

Назначение методов понятно из названий:

- assign() — присвоить;
- append() — добавить («прицепить»);
- insert() — вставить;

- ❑ `erase()` — удалить;
- ❑ `replace()` — заменить;
- ❑ `compare()` — сравнить;
- ❑ `find()` — искать.

Строковый класс обеспечивает прямой доступ к символам строки (операция индексирования `[]`) и метод `at()`, а также последовательный доступ — с помощью прямых (методы `begin()` и `end()`) и обратных (методы `rbegin()` и `rend()`) итераторов.

Можно видеть большое разнообразие перегрузок одной и той же функции — это связано с тем, что требуется выполнять операции и со строками, и с символьными массивами (строками в стиле C). Поэтому практически каждый метод реализован как минимум в двух вариантах: с параметрами типов `basic_string` и `charT*`.

Создавать объекты-строки можно разнообразнейшими способами:

```
string s;                // создает пустую строку
string s(cstr);          // создает строку из строки в стиле C
string s(cstr, len);     // создает строку из len символов строки в стиле C
string s(num, ch);       // создает строку из num символов ch
string s(str);           // создает строку-копию из строки str
string s(str, ind);      // создает строку из символов строки str,
                        // начиная с индекса ind
string s(str, ind, len); // создает строку длиной не более len символов
                        // строки str, начиная с индекса ind
string s(beg, end);      // создает строку из интервала [beg, end)
                        // символов другой строки
```

Как видим, строка не может инициализироваться единичной символьной константой, например `'a'`. Но, естественно, один из конструкторов является конструктором преобразования `const charT* -> string`. Обратное преобразование можно выполнять только явным образом с помощью трех методов:

- ❑ `data()` — возвращает содержимое строки в виде массива символов (в конце массива отсутствует завершающий символ `'\0'`);
- ❑ `c_str()` — возвращает содержимое строки в виде строки в стиле C (с завершающим символом `'\0'`);
- ❑ `cory()` — копирует содержимое строки в символьный массив-параметр метода (завершающий символ `'\0'` отсутствует);

Функции `data()` и `c_str()` возвращают указатель на внутренний буфер строки, поэтому вызывающая сторона не должна изменять символы или освобождать память.

Операция сцепления `operator+` реализована как внешняя дружественная функция со всеми возможными сочетаниями типов параметров:

```
// Сцепление
template <class charT, class traits, class Allocator>
basic_string operator+ (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
basic_string operator+ (const charT*, const basic_string&);
```

```
template <class charT, class traits, class Allocator>
    basic_string operator+ (charT, const basic_string&);
template <class charT, class traits, class Allocator>
    basic_string operator+ (const basic_string&, const charT*);
template <class charT, class traits, class Allocator>
    basic_string operator+ (const basic_string&, charT);
```

Кроме того, реализован полный набор операций сравнения; каждая операция реализована в трех вариантах со всеми возможными сочетаниями параметров. Например, сравнение на равенство обеспечивается следующими тремя функциями:

```
template <class charT, class traits, class Allocator>
    bool operator== (const basic_string&, const basic_string&);
template <class charT, class traits, class Allocator>
    bool operator== (const charT*, const basic_string&);
template <class charT, class traits, class Allocator>
    bool operator== (const basic_string&, const charT*);
```

Как обычно, реализована функция обмена `swap()`:

```
template <class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& a,
         basic_string<charT,traits,Allocator>& b);
```

И наконец, в виде внешних дружественных функций реализованы операции ввода-вывода строк, которые мы рассматривали в главе 14:

```
template<class charT, class traits, class Allocator>
    basic_istream<charT, traits>& operator>>(istream&, basic_string&);
template <class charT, class traits, class Allocator>
    basic_ostream<charT, traits>& operator<<(ostream&, const basic_string&);
template <class Stream, class charT, class traits, class Allocator>
    basic_istream<charT, traits>& getline(Stream&, basic_string&, charT);
```

Литература

Читайте книги — некоторые из них специально для этого написаны.

Михаил Генин

С и C++

1. International Standart ISO/IEC 14882:2003(E), Programming languages — C++.
2. *Страуструп Б.* Язык программирования C++, спец. изд.: Пер. с англ. — М.: Издательство «Бином»; СПб.: Невский Диалект, 2001.
3. *Лишнер Р.* C++. Справочник. — СПб.: Питер, 2005.
4. *Керниган Б., Ритчи Д.* Язык программирования Си: Пер. с англ., 3-е изд., испр. — СПб.: «Невский Диалект», 2001.

Основы программирования на С и C++

5. *Липпман С. Б.* Основы программирования на C++. Серия C++ *In-Depth*. Т. 1: Пер. с англ. — М.: Издательский дом «Вильямс», 2002.
6. *Кениг Э., Му Б. Э.* Эффективное программирование на C++. Серия C++ *In-Depth*. Т. 2: Пер. с англ. — М.: Издательский дом «Вильямс», 2002.
7. *Хенкеманс Д., Ли М.* Программирование на C++: Пер. с англ. — СПб.: Символ-Плюс, 2002.
8. *Дейтел П. Дж., Дейтел Х. М.* Как программировать на C++. Введение в объектно-ориентированное проектирование с использованием UML: Пер. с англ. — М.: Издательство «Бином», 2002.
9. *Лафоре Р.* Объектно-ориентированное программирование в C++. Классика Computer Science. 4-е изд. — СПб.: Питер, 2003.
10. *Лантев В. В.* C++. Экспресс-курс. — СПб.: БХВ-Петербург, 2004.

11. Эккель Б. *Философия C++. Введение в стандартный C++*. 2-е изд. — СПб.: Питер, 2004.
12. Эккель Б., Эллисон Б. *Философия C++. Практическое программирование*. 2-е изд. — СПб.: Питер, 2004.
13. Павловская Т. А. *C/C++*. Программирование на языке высокого уровня. — СПб.: Питер, 2002.
14. Павловская Т. А., Щупак Ю. А. *C/C++*. Структурное программирование: Практикум. — СПб.: Питер, 2002.
15. Павловская Т. А., Щупак Ю. А. *C++*. Объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2004.
16. Труб И. И. *Объектно-ориентированное моделирование на C++*: Учебный курс. — СПб.: Питер, 2006.

Профессиональное объектно-ориентированное программирование на C++

17. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. *Приемы объектно-ориентированного проектирования. Паттерны проектирования*. — СПб.: Питер, 2001.
18. Шаллоуэй А., Тротт Д. *Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию*: Пер. с англ. — М.: Издательский дом «Вильямс», 2002.
19. Влиссидес Д. *Применение шаблонов проектирования. Дополнительные штрихи*: Пер. с англ. — М.: Издательский дом «Вильямс», 2003.
20. Александреску А. *Современное проектирование на C++*. Серия *C++ In-Depth*. Т. 3: Пер. с англ. — М.: Издательский дом «Вильямс», 2002.
21. Саттер Г. *Решение сложных задач на C++*. Серия *C++ In-Depth*. Т. 4: Пер. с англ. — М.: Издательский дом «Вильямс», 2002.
22. Саттер Г. *Новые сложные задачи на C++*: Пер. с англ. — М.: Издательский дом «Вильямс», 2005.
23. Мейерс С. *Эффективное использование C++. 50 рекомендаций по улучшению наших программ и проектов*: Пер. с англ. — М.: ДМК Пресс, 2000.
24. Мейерс С. *Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению наших программ и проектов*: Пер. с англ. — М.: ДМК Пресс, 2000.
25. Эджер Дж. *C++: библиотека программиста*. — СПб.: ЗАО «Издательство «Питер», 1999.

26. *Коплиен Дж.* Программирование на C++. Классика Computer Science. — СПб.: Питер, 2005.
27. *Бадд Т.* Объектно-ориентированное программирование в действии: Пер. с англ. — СПб.: Питер, 1997.

Шаблоны и STL

28. *Вандевурд Д., Джосаттис Н.* Шаблоны C++: Справочник разработчика: Пер. с англ. — М.: Издательский дом «Вильямс», 2003.
29. *Джосьютис Н.* C++. Стандартная библиотека: Пер. с англ. — СПб.: Питер, 2004.
30. *Остерн М. Г.* Обобщенное программирование и STL: Использование и наращивание стандартной библиотеки шаблонов C++: Пер. с англ. под ред. А. Махоткина и И. В. Романовского. — СПб.: Невский Диалект, 2004.
31. *Аммерааль Л.* STL для программистов на C++: Пер. с англ. — М.: ДМК, 1999.
32. *Халтерн П.* Стандартная библиотека C++ на примерах: Пер. с англ. — М.: Издательский дом «Вильямс», 2001.
33. *Мейерс С.* Эффективное использование STL. Библиотека программиста. — СПб.: Питер, 2002.
34. *Москвин П. В.* Азбука STL. — М.: Горячая линия — Телеком, 2003.
35. *Плаугер П., Степанов А., Ли М., Массер Д.* STL — стандартная библиотека шаблонов C++: Пер. с англ. — СПб.: БХВ-Петербург, 2004.

Алгоритмы и структуры данных

36. *Коллинз У. Дж.* Структуры данных и стандартная библиотека шаблонов. — М.: ООО «Бином-Пресс», 2004.
37. *Сэджвик Р.* Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ. — Киев.: Издательство «ДиаСофт», 2001.
38. *Сэджвик Р.* Фундаментальные алгоритмы на C++. Алгоритмы на графах: Пер. с англ. — СПб.: ООО «ДиаСофтЮП», 2002.
39. *Каррано Ф. М., Причард Дж. Дж.* Абстракция данных и решение задач на C++. Стены и зеркала, 3-е издание: Пер. с англ. — М.: Издательский дом «Вильямс», 2003.
40. *Браунси Кен.* Основные концепции структур данных и реализация в C++: Пер. с англ. — М.: Издательский дом «Вильямс», 2002.

41. Фридман А., Кландер Л., Михаэлис М., Шилдт Х. С/С++. Архив программ — М.: ЗАО Издательство «Бином», 2001.
42. Хэзфилд Р., Кирби Л. и др. Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста: Пер. с англ. — К.: Издательство «ДиаСофт», 2001.

Дополнительная литература

43. Страуструп Б. Дизайн и эволюция С++: Пер. с англ.— М.: ДМК Пресс; СПб.: Питер, 2006.
44. Фаронов В. В. Паскаль 7.0. Начальный курс: Учебное пособие. — М.: Нолидж, 1997.
45. Оберг Р., Торстейнсон П. Архитектура .NET и программирование с помощью Visual C++.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002.
46. Олафсен Ю., Скрайбер К., Уайт К. Д. и др. MFC и Visual C++ 6. Энциклопедия программиста.: Пер. с англ. — СПб.: ООО «ДиаСофтЮП», 2003.
47. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows.: Пер. с англ. — СПб.: Питер; М.: Издательско-торговый дом «Русская Редакция», 2001.
48. Круглински Д., Уингоу С., Шеферд Дж. Программирование на Microsoft Visual C++ 6.0 для профессионалов.: Пер. с англ. — СПб.: Питер; М.: Издательско-торговый дом «Русская Редакция», 2001.
49. Шамис В. А. Borland C++ Builder 6. Для профессионалов. — СПб.: Питер, 2003.
50. Шеферд Дж. Программирование на Microsoft Visual C++.NET. Мастер-класс.: Пер. с англ. — 2-е изд. — М.: Издательско-торговый дом «Русская редакция»; СПб.: Питер, 2005.
51. Бокс Д. Сущность технологии COM. — СПб.: Питер, 2001.
52. Трельсен Э. Модель COM и применение ATL 3.0.: Пер. с англ. — СПб.: БХВ-Петербург, 2001.
53. Мартин Р. Быстрая разработка программ: принципы, примеры, практика.: Пер. с англ. — М.: Издательский дом «Вильямс», 2004.
54. Холлингворт Дж., Сворт Б., Кэшман М., Густавсон П. Borland C++ Builder 6. Руководство разработчика.: Пер. с англ. — М.: Издательский дом «Вильямс», 2003.
55. Петцольд Ч. Программирование для Windows 95. В 2-х томах. Т. I.: Пер. с англ. — СПб.: BHV — Санкт-Петербург, 1997.
56. Петцольд Ч. Программирование для Windows 95; в 2-х томах. Т. II.: Пер. с англ. — СПб.: BHV — Санкт-Петербург, 1997.

57. *Румянцев П. В.* Работа с файлами в Win32 API. — М.: Горячая линия — Телеком, 2000.
58. *Харт Дж. М.* Системное программирование в среде Win32: Пер. с англ. — М.: Издательский дом «Вильямс», 2001.
59. *Ганеев Р. М.* Проектирование интерфейса пользователя средствами Win32 API. — М.: Горячая линия — Телеком, 2001.
60. International Standart ISO/IEC 9899:1999(E), Programming languages — C.