Е. В. Пышкин

ТЕОРИЯ И ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

ОСНОВНЫЕ КОНЦЕПЦИИ И МЕХАНИЗМЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

- *Объектная модель и ее развитие в современных языках программирования*
- *Иерархическое проектирование типов и программ*
- *Введение в обобщенное программирование*
- *Управляемые среды проектирования*
- *Основы компонентного программирования*
- *Демонстрационные программы, вопросы и упражнения для самостоятельной работы, примеры заданий практикума*

УЧЕБНОЕ ПОСОБИЕ

bhv

Е. В. Пышкин

ТЕОРИЯ И ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

ОСНОВНЫЕ КОНЦЕПЦИИ И МЕХАНИЗМЫ ОБЪЕКТНО–ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Рекомендовано Учебно-методическим объединением по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению подготовки 553000 – «Системный анализ и управление»

Санкт-Петербург

«БХВ-Петербург»

2005

УДК 681.3.068(075.8)
ББК 32.973.26-018.1я73
П94

Пышкин Е. В.

П94 Основные концепции и механизмы объектно-ориентированного программирования. — СПб.: БХВ-Петербург, 2005. — 640 с.: ил.

ISBN 5-94157-554-8

Рассматривается понятие объектной модели и анализируются механизмы управления вычислительным процессом, лежащие в основе объектно-ориентированного подхода: классы и интерфейсы, динамическое связывание, обработка исключений, пространства имен. Подробно рассматривается конструирование обобщенных типов и библиотека ввода-вывода применительно к программированию на C++. Содержится информация об управляемом коде, свойствах, делегатах, событиях, специализированных атрибутах, отражении, основах компонентной архитектуры.

Размещенные на компакт-диске демонстрационные программы разработаны с использованием современных языков программирования: C++, Java, C#, Visual Basic.

Для студентов и преподавателей технических вузов

УДК 681.3.068(075.8)
ББК 32.973.26-018.1я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Людмила Еремеевская*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Мирошенков*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

РЕЦЕНЗЕНТЫ:
Жабко А. П., доктор физ.-мат. наук, проф., зав. каф. теории управления СПбГУ
Евдокимов В. Е., к. т. н., доцент каф. информационно-измерительных технологий СПбГПУ

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 15.06.05.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 51,6.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-554-8

© Пышкин Е. В., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Оглавление

Предисловие **1**

Преподавание: исторический экскурс 1
Основная задача книги 3
Благодарности 4

Введение **6**

Предпосылки 6
Организация данной книги 6
Исходные тексты программ и листинги программ 9
О соответствии стандарту C++ 9
Одно замечание о терминологии 11

ЧАСТЬ I. ОСНОВАНИЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ **13**

Глава 1. Развитие языков как развитие абстрактных моделей **15**

1.1. Концепции проектирования и языки программирования 15
1.2. Абстрактные модели, лежащие в основе языков программирования 17
Абстрагирование в языках структурного императивного программирования 18
Другие абстрактные модели 19
1.3. Сущность объектно-ориентированного подхода 20
1.4. Вопросы и упражнения 25

Глава 2. Элементы объектной модели **26**

2.1. Смысл абстрагирования как элемента объектной модели 27
2.2. Смысл инкапсуляции как элемента объектной модели 30
2.3. Отношения классов 35
Отношение обобщения 36
Отношение ассоциации 40
Отношение зависимости 46
Отношение реализации 47

2.4. Типы структурных иерархий ... 50
Структурная иерархия "is-part-of" (агрегирование как разновидность ассоциации) ... 50
Структурные иерархии "is-a" и "is-like-a" ... 51
2.5. Иерархии классов и модули: модель C++ ... 58
2.6. Вопросы и упражнения ... 62

ЧАСТЬ II. КОНСТРУИРОВАНИЕ ТИПОВ ... 63

Глава 3. Класс как основной механизм абстракции ... 65

3.1. Структура и организация определения класса ... 65
Элементы-данные и элементы-действия ... 66
Определение класса ... 66
Управление доступом к членам класса ... 67
Класс *QueueInt*: реализация инкапсуляции ... 68
Порядок объявления открытых и закрытых членов класса ... 70
Важное замечание об инкапсуляции ... 71
3.2. Инициализация объектов. Конструкторы ... 72
Конструктор как инструмент гарантированной инициализации ... 72
3.3. Спецификация и реализация класса ... 75
Спецификация класса ... 76
Реализация класса ... 77
3.4. Альтернативные способы инициализации объектов.
Перегрузка конструкторов ... 81
Понятие о конструкторе по умолчанию ... 83
Права доступа в связи с конструкторами ... 84
Конструкторы как неявные преобразования ... 84
3.5. Разрушение среды функционирования объектов. Деструкторы ... 87
3.6. Основные возможности в связи с определением членов класса ... 90
Константные члены класса ... 91
Константные данные ... 91
Константные функции ... 92
Статические константы, переменные и функции ... 92
Члены класса, требующие обязательную инициализацию ... 97
Инициализация статических членов класса ... 99
Понятие о константном указателе *this* ... 100
Встроенные функции-члены класса ... 105
3.7. Перегруженные функции и операции-члены класса ... 106
Перегрузка методов класса ... 106
Перегрузка арифметических операций ... 108
Перегружаемые операции-члены класса ... 108
Перегружаемые операции, не являющиеся членами класса ... 110
Перегружаемые операции-друзья класса ... 111
Перегрузка операций ввода-вывода ... 111
Перегрузка других операций ... 112
Несколько замечаний по поводу перегрузки операций *new* и *delete* ... 118
3.8. Дружественный доступ. Локальные и вложенные классы ... 119
Несколько дополнительных замечаний о дружественном доступе ... 120
Локальные и вложенные классы ... 121

3.9. Копирование объектов класса 123
Проблемы копирования объектов и понятие поверхностного копирования.....123
Реализация корректной семантики копирования объектов.....126
Копирующий конструктор и перегрузка операции присваивания 126
Запрет копирования объектов данного класса 133
3.10. Вопросы и упражнения 135

Глава 4. Наследование и иерархии классов 142

4.1. Наследование как основная форма отношения обобщения 142
Можно ли обойтись без наследования 142
Какие преимущества получает программист, используя отношение наследования классов 144
4.2. Методы производного класса в отношении к методам базового класса 151
Переопределение функций-членов базового класса в производном классе 151
Инициализация объектов производных классов.....153
Наследование конструкторов и операций.....155
Порядок вызова конструкторов и деструкторов в связи с иерархией классов.....156
4.3. Управление доступом к членам класса в связи с наследованием..... 158
Наследование как многократное использование интерфейса 163
4.4. Реализация наследования: модели, отличные от C++ 164
Однокоренная иерархия классов 164
Сокрытие имен.....165
Неизменные методы и классы 167
4.5. Вопросы и упражнения..... 168

Глава 5. Объекты классов и полиморфизм.....173

5.1. Понятие о статическом и динамическом связывании..... 173
Статическое связывание. Установление типов объектов во время компиляции 174
Динамическое связывание. Установление типов объектов во время выполнения.....177
5.2. Виртуальные функции — механизм реализации полиморфизма в C++..... 177
Общие сведения 178
Пример со студентами и аспирантами 178
Пример с геометрическими фигурами 181
Требования к сигнатуре виртуальной функции 184
Таблица виртуальных функций — основной элемент механизма реализации позднего связывания 185
Таблица виртуальных функций..... 185
Выбор корректной версии виртуальной функции 186
Инициализация указателя на таблицу виртуальных функций..... 187
Простой пример для иллюстрации затрат памяти на реализацию полиморфизма..... 188
Чисто виртуальные функции и абстрактные классы 192
Виртуальные функции: некоторые подробности.....193
Разрушение среды функционирования полиморфных объектов. Виртуальные деструкторы..... 193
Вызов виртуальной функции из функции-члена класса..... 197

Можно ли в производном классе определить невиртуальную функцию вместо виртуальной функции базового класса? ... 200
Функции-не члены класса, работающие подобно виртуальным функциям ... 201
5.3. Динамическое связывание и приведение типов ... 204
RTTI (определение типа во время выполнения) и понижающее приведение типов ... 204
RTTI: проблемы использования ... 207
5.4. Несколько замечаний о преобразованиях типа в C++ ... 211
Преобразование типов во время компиляции (преобразование *static_cast*) ... 211
Преобразование типов во время выполнения (преобразование *dynamic_cast*) ... 212
Преобразования для избавления от константности *(const_cast)* ... 213
Преобразование "на свой страх и риск" *(reinterpret_cast)* ... 213
5.5. Вопросы и упражнения ... 214

Глава 6. Обработка ошибок на основе использования механизма исключений ... 219

6.1. Варианты обработки ошибок, не связанные с использованием исключений ... 220
Обработка ошибки на месте ... 221
Использование возвращаемых значений ... 221
Использование глобальных переменных или переменных-членов класса, исполняющих роль индикаторов состояния приложения или объекта ... 224
Использование специально разработанных функций ... 228
Использование функций обратного вызова ... 229
6.2. Обработка исключений ... 234
Основная идея подхода ... 234
Что такое "исключение" и как оно образуется ... 236
Исключения следует перехватывать по ссылке ... 238
Преимущества встраивания в язык обработки исключительных ситуаций ... 241
6.3. Особенности обработки исключений в языке C++ ... 242
Типы исключений ... 242
Использование встроенных типов языка ... 242
Использование типов, определяемых пользователем ... 244
Группирование исключений ... 249
Обработчики исключений ... 252
Порядок записи обработчиков ... 252
Повторная генерация исключения ... 252
Использование обработчика *catch(...)* ... 255
Когда исключение считается обработанным? ... 256
Исключения в связи с ошибкой выделения памяти ... 256
Исключения в конструкторах и деструкторах ... 258
Конструкторы и исключения ... 258
Деструкторы и исключения ... 264
Спецификация исключений ... 268
Неожидаемые и неперехваченные исключения ... 269
Стандартные исключения ... 271

6.4. Особенности обработки исключений в языках Java и C# ... 272
Java: обязательная спецификация и обработка исключений в Java ... 273
C#: необязательная обработка исключений ... 274
6.5. Вопросы и упражнения ... 274

ЧАСТЬ III. ОРГАНИЗАЦИЯ И ВЗАИМОДЕЙСТВИЕ ТИПОВ ... 281

Глава 7. Множественное наследование и интерфейсы ... 283

7.1. Множественное наследование для реализации суммы функциональностей ... 284
Обычное множественное наследование ... 284
Множественное наследование с общим базовым классом ... 292
7.2. Виртуальные базовые классы ... 299
7.3. Дискуссия о множественном наследовании. Понятие интерфейса ... 306
Реализация семантики наследования с интерфейсами на языке C++ ... 309
Реализация семантики наследования с интерфейсами на языке C# ... 312
Конфликты имен и сигнатур при реализации интерфейсов ... 316
7.4. Роль интерфейсов в компонентном программировании ... 318
7.5. Вопросы и упражнения ... 319

Глава 8. Пространства имен в связи с модульностью и иерархией ... 321

8.1. Области действия и пространства имен ... 321
Программный проект: модули и интерфейсы ... 322
8.2. Пространства имен в C++ ... 323
Помещение программных объектов в пространство имен ... 324
Использование программных объектов, объявленных в пространстве имен ... 325
Пространство имен как общая среда реализации программного проекта и пространство имен как внешний интерфейс для пользователей ... 328
Пространства имен: технические подробности ... 329
Разрешение конфликтов имен ... 329
Псевдонимы пространств имен ... 330
Открытость пространств имен ... 330
Объединение пространств имен ... 331
Вложенные пространства имен ... 332
Пространства имен как механизм управления версиями ... 333
Стандартное пространство имен. Пространства имен и код, использующий библиотеки C ... 337
Проблемы пространств имен в C++ ... 338
8.3. Пространства имен в Java ... 341
Именование пакетов ... 341
Создание и использование пакета ... 343
Права доступа в связи с модульной организацией ... 346
8.4. Пространства имен в C#. Введение в архитектуру приложения .NET ... 346
8.5. Вопросы и упражнения ... 350

Глава 9. Введение в обобщенное программирование ... 352

9.1. Обобщенное программирование без использования шаблонных функций ... 352
Определение функции сортировки простыми обменами для сортировки массива целых чисел ... 353
Определение универсальной функции сортировки простыми обменами ... 353

9.2. Шаблонные функции 356
Определение шаблонной функции 356
Использование шаблонной функции. Инстанцирование 358
9.3. Шаблонные классы — основной инструмент параметрического полиморфизма 359
Определение и использование шаблонного класса 359
Параметры шаблонов 363
9.4. Контейнеры и итераторы 364
Проектирование специализированных контейнеров 364
Определение специализированного контейнерного класса 365
Итерируемые контейнеры 365
Использование итератора специализированного контейнера 367
Проектирование стандартных контейнеров 368
Обобщенное представление стандартного контейнера и итератора контейнера 369
Определение контейнерного класса в стиле STL 370
Обобщенные функции для обработки стандартных контейнеров 385
Стандартная библиотека шаблонов (STL) 386
9.5. Контейнеры, построенные на основе однокоренной иерархии классов (модели Java, C#) 388
9.6. Иерархии шаблонных классов 390
Организация размещения данных в памяти 390
Постановка задачи 392
Разработка классов 392
9.7. Вопросы и упражнения 403

ЧАСТЬ IV. РАЗВИТИЕ МОДЕЛЕЙ 405

Глава 10. Организация вычислительного процесса в управляемых средах 407

10.1. Управляемый и неуправляемый код 407
Управляемый код в Java 409
Взаимодействие Java с неуправляемым кодом 412
Ограничения, накладываемые управляемым кодом 417
Управляемый код платформы Microsoft .NET 418
Код, безопасный по отношению к типам 422
Вместо резюме 423
10.2. Встроенные, размерные и ссылочные типы 424
Объекты и встроенные типы 428
10.3. Автоматическое удаление объектов (сборка мусора) 431

Глава 11. Развитие объектно-ориентированной модели управления типами 435

11.1. Реализация полиморфизма 435
Java: полиморфизм по умолчанию 436
C#: спецификатор *new*, версии классов и виртуальные функции 438
11.2. Свойства 445
Поддержка свойств на уровне языка (на примере C#) 446
Особый случай: индексирование 447

11.3. Функции обратного вызова и делегаты ... 450
Понятие делегата и реализация в языке C# ... 451
11.4. Программирование, ориентированное на события ... 455
Обработка сообщений приложением Windows ... 456
Объектно-ориентированная архитектура приложения ... 457
Поддержка модели программирования, ориентированной на события, средствами языка ... 459
11.5. Специализированные атрибуты ... 464
Применение атрибутов, используемых компилятором ... 465
Использование атрибутов в период выполнения ... 468
Атрибуты, определяемые пользователем ... 471
11.6. Отражение (рефлексия) ... 474
Извлечение информации о типе ... 474
Создание экземпляров типов ... 476
Отражение методов класса ... 477
11.7. Интеграция кода и документации ... 478
Документирующие комментарии: модель Java ... 479
Документирующие комментарии: модель C# ... 481

Глава 12. Введение в компонентное программирование ... 483

12.1. Понятие о компонентной архитектуре ... 484
Сокрытие реализации ... 484
Повторное использование кода ... 485
Динамическая компоновка и совместимость интерфейсов ... 487
12.2. Реализация основных элементов компонентной архитектуры на базе COM ... 487
Доступ к компоненту посредством запроса интерфейса ... 488
Идентификация интерфейсов, поддерживаемых компонентом ... 497
Управление временем жизни компонента ... 497
Явное выделение элементов архитектуры компонентного приложения ... 506
12.3. COM-сервер и COM-клиент: действующий макет ... 513
Реализация и использование DLL-компонента ... 513
Фабрики классов и регистрация компонентов ... 517
Преодоление языковой зависимости ... 531
Основные проблемы COM ... 539
12.4. Межъязыковая и межплатформенная интеграция: проблемы и решения ... 542
12.5. Элементы компонентной архитектуры .NET Framework ... 545
Управляемые модули и сборки ... 546
Общая система типов .NET ... 551
Общеязыковая спецификация ... 554
12.6. Вопросы и упражнения ... 557

Заключение ... 558

ПРИЛОЖЕНИЯ ... 559

Приложение 1. Примеры заданий практикума ... 561

Практикум. Задание № 1 ... 561
Цель задания ... 561

Основные умения, которые должны быть продемонстрированы студентом 561
Постановка задачи 562
Оформление результатов выполнения упражнения 562
Практикум. Задание № 2 562
Цель задания 562
Основные умения, которые должны быть продемонстрированы студентом 562
Постановка задачи 563
Оформление результатов выполнения упражнения 563
Практикум. Задание № 3 564
Цель задания 564
Основные умения, которые должны быть продемонстрированы студентом 564
Постановка задачи 564
Оформление результатов выполнения упражнения 565
Практикум. Задание № 4 565
Цель задания 565
Основные умения, которые должны быть продемонстрированы студентом 565
Постановка задачи 565
Оформление результатов выполнения упражнения 566
Практикум. Задание № 5 566
Цель задания 566
Основные умения, которые должны быть продемонстрированы студентом 566
Постановка задачи 567
Рецензия на чужой проект 567
Обсуждение Вашего проекта 567
Подведение итогов 568
Практикум. Курсовой проект 568
Цель задания 568
Основные умения, которые должны быть продемонстрированы студентом 568
Постановка задачи 569
Отчетность по заданию и подведение итогов 571
Рекомендуемые источники 571

Приложение 2. Введение в потоковый ввод-вывод C++ 573

П2.1. Иерархия классов ввода-вывода (заголовочный файл iostream.h) 574
Класс *ios* и производные классы 575
Класс для вывода *ostream* 578
Классы для вывода *ofstream*, *ostrstream*, *ostream_withassign* 578
Класс для ввода *istream* 578
Классы для ввода *ifstream*, *istrstream*, *istream_withassign* 579
Класс для ввода и вывода *iostream* 579
Классы для вывода *fstream*, *strstream*, *stdiostream* 579
П2.2. Вывод данных 579
Вывод данных встроенных типов 580
Вывод символов и строк 580
Буферизация 580
Форматирование вывода. Манипуляторы 582
Вывод для типов, не встроенных в C++ 586

П2.3. Ввод данных 589
Ввод данных встроенных типов 589
Ввод символов и строк 589
Форматированный ввод. Манипуляторы 590
Ввод для типов, не встроенных в C++ 591
П2.4. Файловые потоки 593
Ввод данных из файлового потока *ifstream* 594
Вывод данных в файловый поток *ofstream* 596
Использование потока *fstream* для реализации ввода и вывода при работе с одним и тем же файлом 598
Манипуляторы в связи с файловыми потоками 600
П2.5. Потоковый ввод-вывод в пространстве имен стандартной библиотеки C++ 600
Иерархия классов стандартной библиотеки ввода-вывода 601
Класс *ios_base* 601
Класс *basic_ios* и структура *char_traits* 604
Классы *basic_istream* и *istream* 606
Классы *basic_ostream* и *ostream* 608
Классы *basic_iostream* и *iostream* 609
Организация работы с файловыми потоками 609
Резюме 610

Приложение 3. Описание компакт-диска 611

Список источников 614

Предметный указатель 623

Посвящается моим любимым
Галине и Матвею

Предисловие

Книга, предлагаемая вниманию читателей, основывается на книге "Основные концепции объектно-ориентированного программирования" [Пышкин, 2003-2], представляя ее существенно переработанный и расширенный вариант. Использование материалов книги при проведении занятий со студентами Санкт-Петербургского государственного политехнического университета (СПбГПУ) позволило уточнить и улучшить содержание многих разделов книги, а также дополнить их новой информацией, актуальной в свете развития современных технологий программирования.

С одной стороны, опыт преподавания программирования в рамках конкретного академического курса способствовал написанию книги и выбору обозреваемых разделов, образуя методическую основу книги. С другой стороны, благодаря написанию этой книги, содержание курса, безусловно, стало более стройным и систематическим. При этом справедливым можно считать тезис, что задача учебной книги не только в том, чтобы помочь студентам в освоении нового материала, но и в том, чтобы помочь самому автору более детально разобраться в обсуждаемых вопросах и методике преподавания дисциплины и, как следствие, улучшить и сделать более интересным процесс обучения студентов.

Преподавание: исторический экскурс

В этой книге анализируются основные концепции и механизмы, используемые и развиваемые объектно-ориентированными языками программирования. Рассмотрение отдельных концепций иллюстрируется фрагментами исходных текстов программ, преимущественно написанных на языке C++. Я и мои коллеги, преподаватели СПбГПУ, работающие на кафедре автоматики и вычислительной техники, начали использовать языки C и C++ в качестве базовых при преподавании программирования примерно с середины 1990-х го-

дов. До этого в учебном процессе использовался преимущественно язык Pascal, заместивший, в свою очередь, языки Fortran и PL/1.

Будучи студентом, автор сначала изучал в рамках университетского курса язык Fortran, а затем — именно язык PL/1 (о чем вспоминает с воодушевлением и не без некоторой ностальгии). Язык PL/1 был сконструирован, исходя из принципа, что любая полезная конструкция, которая может быть реализована, должна по возможности включаться в язык. Это обусловило сложность и большой размер языка PL/1, который включает около 200 служебных слов. Многим студентам было нелегко справиться с этой сложностью (хотя автору и удавалось провести "интенсивный двухдневный курс для отстающих" — это было изматывающее предприятие, закончившееся тем, что подопечные сдали экзамен со средним баллом 4.5, — может быть, именно тогда желание заниматься преподавательской деятельностью обрело конкретные очертания).

Многие специалисты полагают, что из перечисленных языков нуждам академического процесса лучше всего соответствует язык Pascal, который, будучи несомненным успехом Вирта, первоначально и был ориентирован на учебные цели. Pascal, являясь удачным сочетанием простоты и выразительности, характеризуется довольно строгой системой типов, жестким контролем преобразований типов и своей очевидной ориентированностью на принципы структурного программирования. Некоторые ограничения, присущие оригинальной версии языка, были преодолены в различных диалектах. Одной из наиболее распространенных и удачных версий многие считают язык Turbo Pascal, реализованный в системах проектирования Borland. Большой популярностью среди программистов продолжает пользоваться объектно-ориентированный язык Object Pascal, положенный в основу системы программирования Delphi (собственно, в наши дни даже более употребительным является наименование "язык Delphi").

При этом следует отметить, что с середины 1990-х годов популярность языка Pascal стала падать как в промышленности, так и в университетах (в особенности в США, что, впрочем, оказало дальнейшее влияние и на Европу, и на Россию) [Sebesta, 2002]. Все большая часть коммерческого программного обеспечения стала разрабатываться на языках C и C++. Таким образом, исходя из нацеленности кафедры на подготовку специалистов-практиков, необходимо было включить в учебные программы язык C++. В отличие от языка Pascal, ни язык C, ни язык C++ не были ориентированы на учебные цели, а создавались их авторами для программистов-профессионалов, поэтому преподавание (а следовательно, и изучение) языка C и особенно языка C++ — не такое уж простое дело.

Долгое время сложность преподавания и изучения языка C++ была связана и с отсутствием общемирового стандарта языка. В то самое время, когда мы начали преподавать язык C++ (т. е. в середине 1990-х годов), программисты

шутили, что процесс стандартизации постоянно находился в состоянии "2 года до завершения" [Зуев, Кротов, Сухомлин, 1996]. При этом Адель Голдберг, бывшая одним из авторов языка Smalltalk, вообще называет C++ большим и сложным языком без целостной философии, за исключением требования сохранить пользовательскую базу языка C: "Одна из основных целей этого языка — сохранение эффективности и особенностей языка C одновременно с обеспечением преимуществ объектно-ориентированного программирования. Некоторые люди интуитивно чувствуют, что эти свойства языка C++ не всегда хорошо согласуются друг с другом" [Sebesta, 2002] (однако продолжают активно использовать его в качестве инструмента проектирования сложных систем и комплексов — Е. П.). Мысли Страуструпа по этому поводу довольно подробно изложены в книге [Stroustrup, 1994].

Основная задача книги

Приступая к работе над книгой, автор стремился избежать стиля книг, упрощающих изучаемый предмет, книг типа "Язык X — для чайников", "Программирование — это просто" и т. д. Несмотря на то, что формат курса, равно как и формат учебного пособия, приводит к почти неизбежной выборочности изучаемых разделов, автор старался не избегать сложных и тонких моментов, а явно указывать на них, одновременно ориентируя студентов в большом объеме издаваемой литературы. Это означает, что, по замечанию Зуева, нельзя о сложных вещах говорить упрощенно, дабы у читателя не сформировался усеченный и выхолощенный образ рассматриваемых инструментов программирования: "Настоящее пособие по сложному языку должно иметь форму, близкую к (...) книге Эллис и Страуструпа [Ellis, Stroustrup, 1990], — комментируемый стандарт. Да, читать и пытаться понять строгий, сложно построенный, местами даже занудный текст весьма непросто — но кто сказал, что профессия программиста проста?" [Зуев, 1997].

Несмотря на то, что большинство примеров реализовано на языке C++, в отдельных главах рассматриваются особенности реализации некоторых концепций и механизмов объектно-ориентированного и компонентного программирования в языках Java и C#. Автор немедленно принимает очевидную критику относительно выбора языков (все они являются, по существу, языками C-группы), и в связи с этим хочет подчеркнуть три момента:

1. Выбор обусловлен наличием личного практического опыта последних лет.
2. Данное учебное пособие нацелено на обеспечение академического курса по объектно-ориентированному программированию. В связи с этим методически удобно ограничиться каким-либо базовым языком и привлекать другие языки либо для иллюстрации концепций, не поддерживаемых ба-

зовым языком, либо для иллюстрации существенных отличий в реализации.

3. Включение в рассмотрение других современных объектно-ориентированных языков (Eiffel, Ada95, Visual Basic, языки платформы Microsoft .NET) является отдельной задачей и требует отдельного решения, которое лучше всего вписывается в рамки курса "Языки программирования", необходимого для обеспечения всестороннего образования в области программных технологий. Не подлежит сомнению, что хороший программный инженер должен знать несколько языков программирования [Meyer, 2001].

Сопровождающие текст примеры в основном довольно просты и непохожи на реальные задачи проектирования. Ценность таких "игрушечных" примеров заключается в том, что, изучая их, студенты смогут понять основные иллюстрируемые ими концепции и механизмы, не оказываясь при этом под давлением объемных исходных текстов.

Отметим, что в данной книге не анализируются во всей полноте достоинства и недостатки объектно-ориентированных технологий, а также вопросы сложности разработки программного обеспечения. Отличные книги, содержащие исследование этой проблематики, упоминаются в списке источников (см., в частности, [Booch, 1994], [Brooks, 1995], [Sebesta, 2002], [Stroustrup, 1994], [Лекарев, 1997]).

Благодарности

Хочу выразить свою благодарность всем коллегам, работающим на кафедре автоматики и вычислительной техники факультета технической кибернетики СПбГПУ, за постоянную поддержку в преподавании и научно-педагогическом творчестве. Благодарю сотрудников учебно-методического объединения по университетскому политехническому образованию за помощь в совершенствовании структуры учебного пособия.

Ряд материалов, использованных при подготовке книги, прошли апробацию в ходе курсов по разработке программного обеспечения в Central Ostrobothnia Polytechnic, Ylivieska, Финляндия. Организация этих занятий была бы невозможна без участия и поддержки моих финских коллег Вейкко Бракса и Юхи Хуумонена. Благодарю профессора Ханспетера Мессенбека из университета г. Линца (Австрия) за лекции, прочитанные им на одной из конференций Microsoft. Систематичность изложения основных элементов программирования на языке C# остается для меня одним из ориентиров в педагогической практике.

Благодарю студентов Андрея Власовских, Александра Григоряна, Алексея Рытенкова, Юрия Викторова, Михаила Веренцова, Андрея Хилько, Алек-

сандра Павлова, Илью Антоненко, Анатолия Карпенко, Ивана Пивоварова, Марата Ахина за их замечания и находки, способствовавшие устранению ошибок и улучшению содержания книги.

Особая признательность — Вере Павловне Тихомировой, в чьем гостеприимном доме в деревне Найдениха Тверской области было написано большинство глав книги.

Спасибо музыке Франца Шуберта, помогавшей мне в плодотворной и комфортной работе над книгой.

Введение

> Специалисты по программному обеспечению — лучшие в мире специалисты по созданию хаоса. Мы должны объяснить студентам, что это непреложный факт и показать им, что следует в связи с этим предпринимать.
>
> *Бертран Мейер*

Предпосылки

Автор исходит из предположения, что читатели книги имеют представление об основных принципах процедурного программирования и, в частности, знакомы с концепциями, лежащими в основе структурного программирования. Необходимой предпосылкой для успешного освоения материала книги является знакомство с основными средствами языка программирования C++, не связанными с объектно-ориентированным программированием.

Восприятию текстов программ, помещенных в книге, читателям поможет знание английского языка. Впрочем, его желательно знать любому квалифицированному специалисту, работающему в области проектирования и использования вычислительных систем.

Для самостоятельного практикума по программированию на C++ читателю потребуются средства разработки на C++. При организации занятий со студентами автор преимущественно использует средства Microsoft Visual Studio 6.0 и Microsoft Visual Studio .NET. На факультативных занятиях студенты используют средства проектирования на языках Java (в рамках изучаемого материала достаточно иметь одну из версий JDK, поддерживающих платформу Java2) и C# (для этого требуется Microsoft .Net SDK или развернутая среда Visual Studio .NET).

Организация данной книги

Книга состоит из введения, четырех частей и заключения, а также содержит три приложения.

Часть I "Основание объектно-ориентированного программирования" включает главы 1—2.

В *главе 1* анализируется развитие языков программирования с точки зрения развития воплощаемых ими концепций проектирования и положенных в основу языков абстрактных моделей. Вкратце представляется генеалогия современных языков программирования и место тех языков, которые используются в книге для иллюстраций.

В *главе 2* представляется понятие объектной модели и анализируются ее основные элементы: абстрагирование, инкапсуляция, иерархия и модульность. В главе рассматриваются отношения между основными абстракциями и простые примеры их использования. Для иллюстрации применяются диаграммы UML (Unified Modeling Language) [Booch, 1999]. При написании первых двух глав частично использованы материалы работы, написанной автором для студентов Central Ostrobothnia Polytechnic (Финляндия) [Пышкин, 2003-1].

Часть II "Конструирование типов" посвящена концепциям и механизмам, лежащим в основе конструирования типов, построения иерархий типов и управления доступом к данным, и включает главы 3—6.

Глава 3 посвящена углубленному изучению классов как основного механизма абстракций в объектно-ориентированном программировании. Рассматривается структура и содержание определения класса, разновидности членов класса и основные принципы конструирования новых типов. Анализируются основные проблемы инициализации и разрушения объектов, перегрузка методов класса и операций, обеспечение правильной семантики копирования объектов.

В *главе 4* представлен основной инструмент построения иерархий классов — наследование. Обсуждаются права доступа к членам классов в связи с наследованием. Анализируется концепция сокрытия имен, рассматриваются различия реализации этой концепции различными языками программирования. Сопоставляются объектно-ориентированные модели организации данных на базе однокоренной иерархии классов и без оной.

Глава 5 посвящена важнейшей концепции объектно-ориентированного программирования — полиморфизму (динамическому связыванию). Анализируются различия механизмов реализации полиморфизма в разных объектно-ориентированных языках и причины этих различий. Студенты получают представление об определении типа времени выполнения и динамическом преобразовании типов.

Глава 6 посвящена анализу механизма исключений как основного инструмента обработки ошибок в объектно-ориентированных языках. Рассматриваются альтернативные варианты реализации обработки ошибок в компьютер-

ной программе и преимущества, получаемые программистом при использовании обработки исключений. Подробно рассматриваются особенности реализации механизма обработки исключений в языке C++, включая разрушение объектов в ходе раскрутки стека, организацию обработки неожидаемых и неперехваченных исключений, исключения, возникающие в ходе ошибок распределения памяти. В главе анализируются задачи спецификации исключений и вкратце рассматривается реализация спецификации исключений в разных языках программирования.

Часть III "Организация и взаимодействие типов" посвящена обсуждению концепций и механизмов, традиционно вызывающих трудности у начинающих программистов, в том числе в связи с реализацией множественного наследования, поддержкой пространств имен и обобщенного программирования. Третья часть включает главы 7—9.

В *главе 7* рассматриваются подробности реализации множественного наследования в C++ и обсуждаются причины, из-за которых многие объектно-ориентированные языки не поддерживают множественное наследование. Студенты получают представление о концепции одиночного наследования с интерфейсами (с примером на языке C#).

В *главе 8* рассматриваются пространства имен как инструмент обеспечения модульности программ на уровне логической организации компонентов. Анализируются присущие C++ ограничения в реализации концепции пространств имен и реализация пространств имен в других языках (Java, C#).

Глава 9 посвящена элементам обобщенного программирования: рассматриваются шаблонные и обобщенные функции, шаблонные классы. Глава содержит введение в обработку контейнерных объектов с помощью обобщенных функций. Рассматриваются основные принципы построения контейнеров, используемые в стандартной библиотеке шаблонов (STL, Standard Template Library). Подробно рассматривается понятие итератора контейнера и приводятся примеры проектирования контейнерных классов в стиле стандартной библиотеки шаблонов.

Часть IV посвящена анализу развития объектно-ориентированных моделей в современных языках и технологиях программирования и включает главы 10—12.

В *главе 10* представляются основы организации управляемого кода и основные механизмы управления типами в средах, построенных на основе концепции управляемого кода.

Глава 11 посвящена развитию моделей управления типами в современных объектно-ориентированных языках, отличных от C++ (анализируются модульность, полиморфизм и версии виртуальных методов, свойства, индекса-

торы, делегаты, специализированные атрибуты, события, документирующие комментарии).

Глава 12 содержит введение в компонентное программирование. В главе анализируется влияние компонентной модели на структуру объектно-ориентированного кода, рассматриваются основные проблемы обеспечения межъязыковой и межплатформенной совместимости компонентов.

Заключение посвящено читателям, дочитавшим эту книгу до конца.

Приложения содержат примеры заданий практикума по объектно-ориентированному программированию, описание основных средств библиотеки ввода-вывода C++, необходимых в работе над учебными проектами, а также описание структуры и содержания материалов, предоставляемых на прилагаемом к книге компакт-диске.

Исходные тексты программ и листинги программ

Исходные тексты примеров программ, использованных в книге, предоставляются на компакт-диске в виде единого пакета. Набор файлов с исходными текстами сопровождается файлом license.txt, содержащим информацию об авторском праве и условиях использования исходных текстов в собственной работе.

Часть представленных в книге исходных текстов пронумерована. Листинги нумеруются в пределах отдельных глав. Все пронумерованные листинги соответствуют программам, распространяемым вместе с книгой в пакете исходных текстов программ. В этом случае листинги, представленные в книге, являются копиями или фрагментами программ с компакт-диска, которые были оттранслированы компиляторами Visual C++ 6.0 и Visual C++ 7.0. Случаи, когда результаты исполнения программы зависят от версии компилятора, оговариваются в комментариях. Незначительные отличия текстов программ, приведенных в книге, от компилируемых текстов на компакт-диске могут быть связаны только с исключением из текста книги подробных заголовочных комментариев и с особенностями форматирования текста, обусловленными требованиями издателя. Такие различия не затрагивают содержание программного кода.

О соответствии стандарту C++

При разработке примеров программ автор стремился обеспечить соответствие получаемых программ требованиям стандартного C++ *(ISO/IEC 14882 "Standard for the C++ Programming Language")*. Исходя из сложившейся на

сегодняшний день организации учебного процесса, в ходе которого студентам сначала представляется процедурная парадигма программирования и концепции структурного императивного программирования *(см. во введении разд. "Предпосылки")*, в некоторых примерах программ могут встречаться отклонения от рекомендаций по следованию стандарту, изложенных, в частности, Страуструпом [Stroustrup, 2000, приложение Б]. Это означает, что некоторые программы, не нарушая стандарта с формальной точки зрения, могут отклоняться от стиля программирования на стандартном C++. Повторяю, что это обусловлено исключительно методическими задачами и особенностями учебного процесса и затрагивает лишь некоторые элементы языка, которые перечисляются ниже.

В частности, в приводимых учебных примерах используются средства стандартной библиотеки C++, определенные в пространстве имен `std`. Поэтому в тех случаях, когда необходимы стандартные средства языка C (например, при работе со строками ANSI C), используются заголовочные файлы без расширения, например

```
# include <cstring>
```

вместо

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

При использовании средств стандартной библиотеки C++ в программу помещается явное указание на используемое пространство имен, например:

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

Это позволяет обеспечить совместимость кода с требованиями стандарта, не вызывая серьезных проблем в восприятии кода даже теми студентами, которые до этого использовали библиотеки языка C. Укажем, что подробности реализации пространств имен в C++ и механизмы обеспечения совместимости со старым кодом *рассматриваются в главе 8*.

В большинстве программ, приводимых до раздела 6.3, при использовании операции резервирования памяти `new` результат ее выполнения проверяется на 0. В соответствии с требованиями стандарта, в старом коде проверки на равенство 0 результата, возвращаемого `new`, должны быть заменены перехватом исключений `bad_alloc` или применением `new (nothrow)` [Stroustrup, 2000, §Б.3.4]. После рассмотрения реализации механизма обработки исключения `bad_alloc` в разделе 6.3, в дальнейших главах при распределении памяти используется версия `new`, генерирующая исключение `bad_alloc` в случае ошибки.

Несмотря на рекомендацию использовать вместо C-строк более высокоуровневый тип `string` (см., например, [Stroustrup, 2000, §20.4.1]), в ряде учебных

проектов продолжают применяться строки в стиле C. Такое решение обусловлено тем обстоятельством, что в этих примерах моделируется процесс создания более высокоуровневых типов на основе существующих механизмов низкого уровня. Это позволяет, в частности, рассмотреть проблемы копирования объектов и резервирования-освобождения динамической памяти в процессе функционирования объектов. В тех случаях, где строки нужны как таковые, используется преимущественно тип `string`.

В целом, автор стремился руководствоваться справедливым замечанием Страуструпа о том, что общее подмножество C и C++ не является лучшим подмножеством для изучения начинающими программистами [Stroustrup, 2000, раздел Б.4]. В связи с этим автор заранее приносит извинения за возможные отклонения от этой рекомендации, которые может заметить внимательный и строгий читатель.

Одно замечание о терминологии

В ряде переводных изданий, посвященных объектно-ориентированному программированию и в особенности программированию на языке C++, переводчики изменили сложившуюся в русском языке терминологическую традицию. Наиболее подробное объяснение этого факта содержится в редакционном предисловии к книге "Язык программирования C++" Страуструпа ([Stroustrup, 2000]). Позволю себе привести пространную цитату из этого текста:

> "Слово "statement" принято переводить на русский язык как "оператор". Мы привыкли к тому, что if, while, case и т. д. — это операторы. Увы, в контексте C++ такой подход неприемлем. Дело в том, что в C++ слово "operator" (которое и переведено как "оператор") имеет совсем другое значение. Оно применяется для обозначения сложения (оператор `+`), разыменования (оператор `*`), выяснения размера (оператор `sizeof()`) и в других подобных случаях. (...) Что же касается термина "statement", то из предлагаемых различными словарями вариантов "утверждение", "предложение", "инструкция" мы выбрали последний, т. к. он, по-видимому, лучше всего соответствует сущности обозначаемых словом "statement" конструкций C++ и, кроме того, периодически встречается в книгах в нужном значении".

По поводу процитированного фрагмента хотелось бы высказать следующие соображения:

1. (Общее соображение). По-моему, в принципе неверно из контекста использования какого-либо термина в ограниченной области, изменять сложившиеся языковые традиции, иначе как общаться с людьми, не работающими в этой ограниченной области?

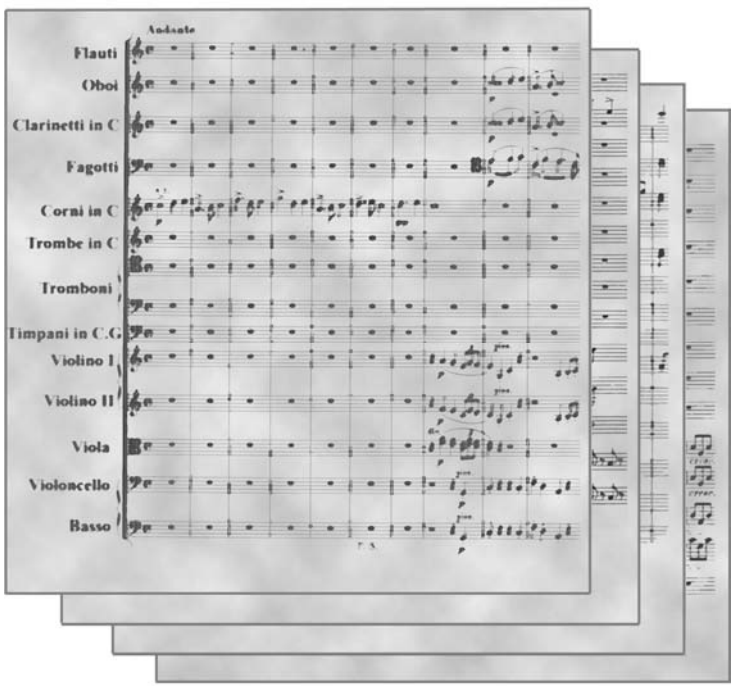
2. (Частное соображение). Не совсем ясно, что означает фраза "в C++ слово "operator" имеет совсем другое значение". Оно имеет то значение, которое и предписано ему английским языком: операция, знак операции! Собственно, вполне корректно сказать, что перегрузка операций является приданием новой семантики знаку операции, и в этом смысле использование служебного слова "operator" довольно точно соответствует существу дела. При этом словари дают вполне адекватный перевод этого термина на русский язык, не идущий вразрез с установившейся традицией.
3. (A propos). Язык C++ включает в себя в качестве подмножества язык C. Одной из существенных сложностей создания и развития C++ является поддержание пользовательской базы языка C. Мне кажется, что пользовательская база включает также и терминологию, используемую в связи с определением конструкций языка. А ведь язык C создавался без служебного слова "operator"!
4. (Наблюдение). Не ссылаясь на авторов, замечу, что мне попадались книги по C++, где понятия "операция" и "оператор" были весьма причудливо перемешаны: видимо, авторы не отнеслись с должным вниманием к означенному редакционному предисловию и использовали эти термины как придется.

На основании изложенных соображений я склонен использовать слово "операция" там, где речь идет об операциях языка (operators). Но чтобы у читателя не возникало излишней путаницы, я буду использовать вполне разумный эквивалент "инструкция" для английского термина "statement". При этом я постараюсь минимизировать использование конфликтного слова "оператор" (хотя кое-где оно все же может встретиться в тексте в своем "исконном" значении, очень уж трудно произносить "инструкция if" вместо "условный оператор" или "инструкция goto" вместо "оператор goto").

> Может быть, программисты и лучшие в мире специалисты по созданию хаоса, но определенно не единственные.

ЧАСТЬ I

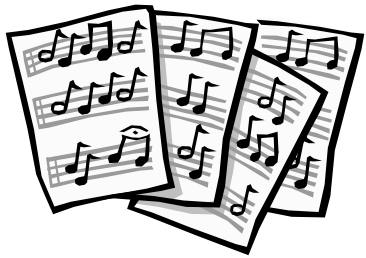
Основание объектно-ориентированного программирования



Глава 1. Развитие языков как развитие абстрактных моделей

Глава 2. Элементы объектной модели

ГЛАВА 1



Развитие языков как развитие абстрактных моделей

> Язык и мышление теснейшим образом связаны. Если язык обеднеет, обеднеет и мышление. Колоссальное значение имеет терминология, грамматические системы, способствующие самовыражению человека.
>
> *Д. С. Лихачев*

Объектная модель представляет собой концептуальную основу объектно-ориентированного программирования, поэтому изучение основных элементов объектной модели и их взаимодействия позволяет лучше разобраться в принципах организации объектно-ориентированных программ, а также понять, в чем сходства и различия поддержки объектно-ориентированной парадигмы различными языками.

Прежде чем перейти к изучению элементов объектной модели как таковых, проанализируем вкратце процесс развития языков программирования. Это позволит нам выяснить концептуальные и методологические различия изобразительных средств, предоставляемых программисту языками программирования, и причины успешности и востребованности объектно-ориентированного подхода.

1.1. Концепции проектирования и языки программирования

Объектно-ориентированное программирование (ООП) является, вероятно, самой распространенной в современной проектной практике парадигмой программирования, которая поддерживается большинством современных языков программирования. При этом реализации концепций и механизмов, состав-

ляющих основание ООП, могут отличаться в разных языках. Поэтому изучение ООП предполагает не столько изучение какого-либо конкретного языка программирования, сколько лежащих в его основе концепций и механизмов как таковых. К самым важным вещам, которые должен знать профессиональный программный инженер, относится набор фундаментальных концепций, которые вновь и вновь используются в его работе и которые в первую очередь следует изучать [Meyer, 2001]. С одной стороны, такой подход к изучению программирования позволяет лучше разобраться в проблемах, возникающих перед разработчиками программного обеспечения (ПО), с другой стороны, позволяет лучше понять причины появления и успешного развития различных языков и технологий программирования.

Основная задача любой технологии заключается в том, чтобы предоставить разработчикам средства решения основных проблем проектирования. Широкое применение технологии (которая всегда является реализацией определенных концепций) доказывает ее успешность. В частности, это и означает, что данная технология (а значит, и лежащие в ее основе концепции, модели, методы проектирования) позволяет разработчикам успешно решать актуальные проблемы проектирования.

Вообще говоря, появление новых технологий программирования в большой степени обусловлено трансформацией проблем проектирования ПО с его растущей сложностью. В свою очередь, рост сложности программного обеспечения обусловлен развитием областей жизнедеятельности человека, где находят применение новейшие информационные и компьютерные технологии. Это становится возможным благодаря развитию средств проектирования и возможностей вычислительной техники, удешевлению аппаратных ресурсов и увеличению продуктивности разработчиков, использующих современные средства проектирования. С другой стороны, сейчас уже никто не удивляется тому, что новейшие технологии проектирования зачастую требуют более мощного аппаратного обеспечения для своей эффективной работы (хотя это иногда и связано с неэффективностью алгоритмов, используемых в ходе разработки программ), подталкивая пользователей к модернизации используемого оборудования и вычислительных средств. Таким образом, новое более мощное оборудование способствует более эффективному использованию программных средств, это порождает новые области применения и дает толчок к возникновению новых задач проектирования, требующих для своего решения новых технологий разработки программ [Пышкин, 2003-1]. Графически описанный процесс представлен на рис. 1.1.

Язык программирования — инструмент, вытекающий из концепций проектирования, инструмент, облекающий в синтаксическую форму основное содержание концепций, их суть. Поэтому изучение языка самого по себе вне связи с моделями проектирования, заложенными в основу языка, не влечет улуч-

шения качества проектирования и не обязательно способствует повышению квалификации разработчика.

Рис. 1.1. Прогресс информационных технологий и рост сложности задачи проектирования

Разумеется, нужно отдавать себе отчет в том, что *качество проектирования*, прежде всего, *определяется квалификацией и творческими способностями проектировщиков*, а не моделью программирования: выдающиеся проекты создают выдающиеся проектировщики [Brooks, 1995]. При этом технология проектирования помогает грамотному разработчику продемонстрировать свои умения и построить качественный программный продукт. Таким образом, *необходимо изучать не языки программирования сами по себе, а концепции, выражаемые этими языками* [Eckel, 2000-1].

1.2. Абстрактные модели, лежащие в основе языков программирования

Языки программирования — это всегда некоторые модели (виртуальные вычислительные машины), позволяющие наиболее эффективно использовать возможности вычислительных средств, существенные для конкретных областей применения. По сути дела, при написании программ ориентируются не на вычислительную машину как таковую, а на некоторую абстрактную модель вычислительного устройства [Minsky, 1967]. Качество абстрактной модели, ее соответствие содержанию решаемых задач во многом определяет качество и эффективность проектирования с использованием этой модели. По меткому замечанию Эккеля, компьютеры не столько машины, сколько средства усиления мысли [Eckel, 2000-1].

Сложность задач, которые возможно решить с применением тех или иных концепций проектирования (читай, выражающих их языков), непосредственно связана с уровнем абстракции как при постановке задачи, так и в ходе ее решения. Абстракция является, вероятно, самым мощным интеллектуальным средством познания из тех, что имеются в распоряжении человека [Dahl, Dijkstra, Hoare, 1972]. Со способностью к абстракции связана и способность человека к творчеству. Кроме того, обнаружение общих абстракций и механизмов значительно облегчает понимание сложных систем [Booch, 1994].

Так, языки ассемблера позволяют избежать необходимости программирования в машинном коде, и в этом смысле ассемблеры являются абстракцией вычислительной машины, для которой были спроектированы [Eckel, 2000-1]. Кстати, и сама вычислительная машина, являясь иерархической системой, может рассматриваться на разных уровнях абстракции [Booch, 1994]. В подобной интерпретации языки, основанные на методологии процедурного программирования, можно считать абстракциями ассемблеров.

Таким образом, язык программирования — не только выразитель концепций проектирования, но и уровня абстракции при представлении задач и технических средств, используемых для решения этих задач. *Развитие языков программирования — это, прежде всего, развитие абстрактных моделей, облегчающих и систематизирующих проектирование.*

Абстрагирование в языках структурного императивного программирования

Методология структурного императивного программирования воплощает подход, характеризующийся принципом последовательного изменения состояния вычислителя пошаговым образом с поддержкой концепции структурного программирования. Императивное программирование является первой методологией программирования, поддержанной на аппаратном уровне и ориентированной на класс архитектур фон Неймана [Одинцов, 2002].

Примерами языков-выразителей концепций структурного императивного программирования могут служить языки Fortran, Pascal, C, PL/1 (в последнем, впрочем, был реализован и ряд других важных концепций, например, обработка исключений). Все перечисленные языки основаны на парадигме процедурного программирования, основанной на представлении программы в виде иерархии процедур и функций. В принципе, понятия процедурного и структурного программирования не следует отождествлять, поскольку использование процедурной модели совместимо и с объектно-ориентированным программированием, и с функциональным программированием, и с параллельным программированием [Легалов, 2000]. Развитие структурного про-

граммирования позволило реализовать или развить ряд важных методов проектирования:

- функционально-иерархическая декомпозиция;
- структурная организация данных;
- повторное использование проектных решений;
- технология тестирования программного обеспечения.

Основная проблема процедурных языков заключается в том, что достигаемый ими уровень абстракции все еще требует от программиста мышления в большей мере в терминах вычислительной машины (пусть даже и виртуальной), чем в терминах задачи, которую ему приходится решать. Качество проектирования определяется в итоге тем, насколько удачно программисту удалось установить соответствие между пространством понятий, характерных для решаемой задачи, и набором изобразительных средств языка, не всегда позволяющих адекватно отобразить эти понятия в рамках машинной модели. Абстрагирование, достигаемое посредством использования процедур, хорошо подходит для описания абстрактных действий, но не предоставляет адекватных языковых средств для описания абстрактных объектов.

Кинг указывает также на проблемы реального проектирования, возникающие в ходе разработки программ на основе функционально-иерархической декомпозиции [King, 1997]:

- Функциональную модель трудно развивать: эволюция системы, учет новых требований может порождать плохо управляемую архитектуру приложения.
- Функционально-ориентированный код трудно обобщать для многократного использования.

Резюме: *языки императивного программирования, основанные на процедурной парадигме, реализуют систематический подход к разработке программного обеспечения, который во многих случаях основан на концепции структурного программирования. При этом задача программиста — перейти от представления о решаемой задаче в терминах, присущих этой задаче, к терминам, присущим модели вычислительной машины.*

Другие абстрактные модели

В историческом плане развитие абстрактных моделей не всегда означало переход на более высокие уровни абстракции представления вычислительной машины как исполнительного устройства. Специфические особенности некоторых классов задач способствовали появлению альтернативных моделей.

Примерами альтернатив моделированию вычислительной машины могут служить концепции, реализованные в языках функционального и логического программирования. Примером языка функционального программирования может служить язык LISP, основанный на модели, которая в компактной форме характеризуется постулатом "все задачи, в конечном счете, могут быть сведены к работе со списками". Классическим примером языка логического программирования является язык Prolog, основывающийся на принципе: "все задачи могут быть сведены к цепочке логических рассуждений" [Eckel, 2000-1]. В частности, эти языки нашли применение в задачах искусственного интеллекта и прикладной математики.

Каждый из подобных подходов оказывался приемлемым или даже лучшим для определенного класса задач, но не обладал достаточной универсальностью. Во многих случаях более успешными оказывались языки общего назначения. Даже в области искусственного интеллекта многие коммерческие образцы систем, основанных на знаниях, были разработаны на универсальных языках. В разных источниках приводятся примеры систем, спроектированных полностью или частично на языках Pascal, Fortran, C, PL/1 (см., например, [Forsyth, 1984], [Попов и др., 1990]).

Для решения задач логического управления в настоящее время активно развивается подход, основанный на использовании теории автоматов — автоматное программирование [Шалыто, 1998]. Модели представления данных и поведения системы, положенные в основу этого подхода, фокусируются на абстракциях *состояния* (определяющего статику объекта управления) и *события* (определяющего динамику процесса управления). Автоматное программирование находит применение не только при решении задач управления, но и в таких областях, как синтаксический анализ и трансляция, моделирование систем управления, проектирование программируемых логических контроллеров и др.

Приведенные примеры показывают, что *развитие абстрактных моделей связано не только с переходом на более высокие уровни абстракции при описании вычислительной машины, а также и с развитием других концепций абстрагирования, ориентированных на различные классы задач.*

1.3. Сущность объектно-ориентированного подхода

Сущность объектно-ориентированного подхода заключается, прежде всего, в двух моментах.

С одной стороны, он предоставляет разработчику инструмент, который позволяет описать задачу и существенную часть реализации проекта в терми-

нах, характеризующих предметную область, а не компьютерную модель. По замечанию Кинга, объектно-ориентированный анализ начинается с исследования предметов реального мира, являющихся частью решаемой задачи [King, 1997]. Хотя применение объектно-ориентированной парадигмы само по себе не обеспечивает качества проектирования, оно способствует ясному представлению данных и присущих им свойств и действий (поведения) на уровне программного кода. При этом программные структуры на формальном языке соответствуют ментальной модели описания задачи и способов ее решения на языке, естественном для данной предметной области. Эта особенность отличает объектно-ориентированные языки от языков императивного программирования, нацеленных на моделирование действий, выполняемых вычислительной машиной (рис. 1.2).

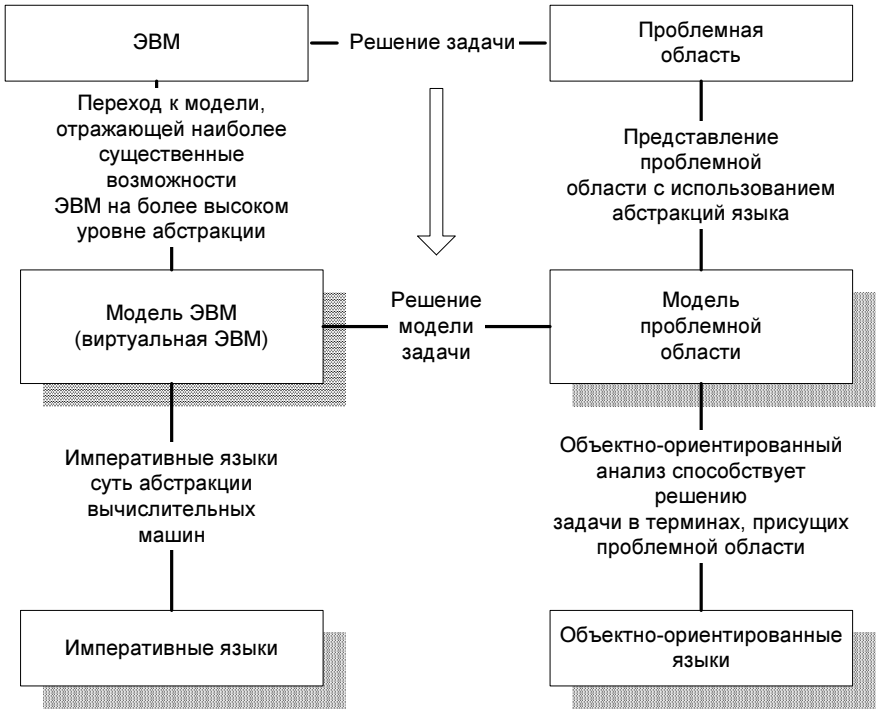


Рис. 1.2. Абстрагирование посредством императивных и объектно-ориентированных языков

С другой стороны, объектно-ориентированное проектирование предоставляет разработчикам гибкий мощный универсальный инструмент, не связанный с каким-то определенным классом задач. По замечанию Буча, "объектный подход зарекомендовал себя как унифицирующая идея всей компьютерной науки, применимая не только в программировании, но также в проектировании интерфейса пользователя, баз данных и даже архитектуры компьютеров"

[Booch, 1994]. При этом использование объектно-ориентированного подхода оправдывает себя не для всех классов задач. Например, для реализации основных вычислительных алгоритмов применение объектно-ориентированного программирования едва ли позволит получить более наглядный и более эффективный код, хотя, разумеется, эти задачи можно использовать в качестве примеров, иллюстрирующих отличия в управлении данными в процедурных и объектно-ориентированных языках [Давыдов, 2001].

Другим классическим примером области, в которой, как раньше считалось, объектная модель вряд ли обеспечивает какие-либо существенные преимущества при проектировании и использовании кода, является синтаксический анализ и трансляция. Однако в ряде случаев объектно-компонентная организация компилятора может быть разумным решением. Здесь можно выделить, по крайней мере, два момента.

Во-первых, разработка и использование элементов транслятора как независимых компонентов (классов) обусловлена развитием интегрированных сред проектирования. Это связано с тем, что реализация различных средств поддержки программирования требует возможности обращения к отдельным компонентам транслятора. Например, подсветка служебных слов и литеральных констант, быстрая навигация по тексту требует вызова лексического анализатора, частичная трансляция фрагментов исходного текста в процессе его набора и информирование программиста о совершаемых ошибках при записи синтаксических конструкций языка требует вызова синтаксического анализатора и т. д. Одной из известных реализаций подобной схемы построения транслятора является так называемая обобщенная инфраструктура компилятора (Common Compiler Infrastructure), проектируемая как часть Microsoft .NET Framework в подразделении Microsoft Research (рис. 1.3). Данная инфраструктура представляет собой библиотеку классов для создания собственных трансляторов языков .NET.

Во-вторых, структурирование компилятора позволяет облегчить его перенос на другую вычислительную архитектуру. Отделяя кодогенерирующие функции от остальной части компилятора, можно существенно упростить создание семейства компиляторов для различных целевых архитектур [Franz, 1998]. Справедливости ради, отметим, что структурирование и компонентная организация компилятора не обязательно предполагают использование объектно-ориентированных технологий.

Во многих случаях наилучшие результаты проектирования обеспечиваются совместным применением множества моделей программирования. Собственно, язык C++ и создавался как язык, поддерживающий множество парадигм проектирования, предоставляя программистам мощный и гибкий механизм для конструирования абстрактных типов, который подходил бы для большинства современных вычислительных машин [Stroustrup, 1998]. C++ под-

держивает много стилей программирования и сочетает эффективность и гибкость языка C для системного программирования с возможностями объектно-ориентированной организации кода и данных. Таким образом, многие специалисты соглашаются с тем, что преимущества объектно-ориентированной технологии можно реализовать быстрее и в более полном объеме, если использовать ее в сочетании с традиционными методами [Лекарев, 1997].

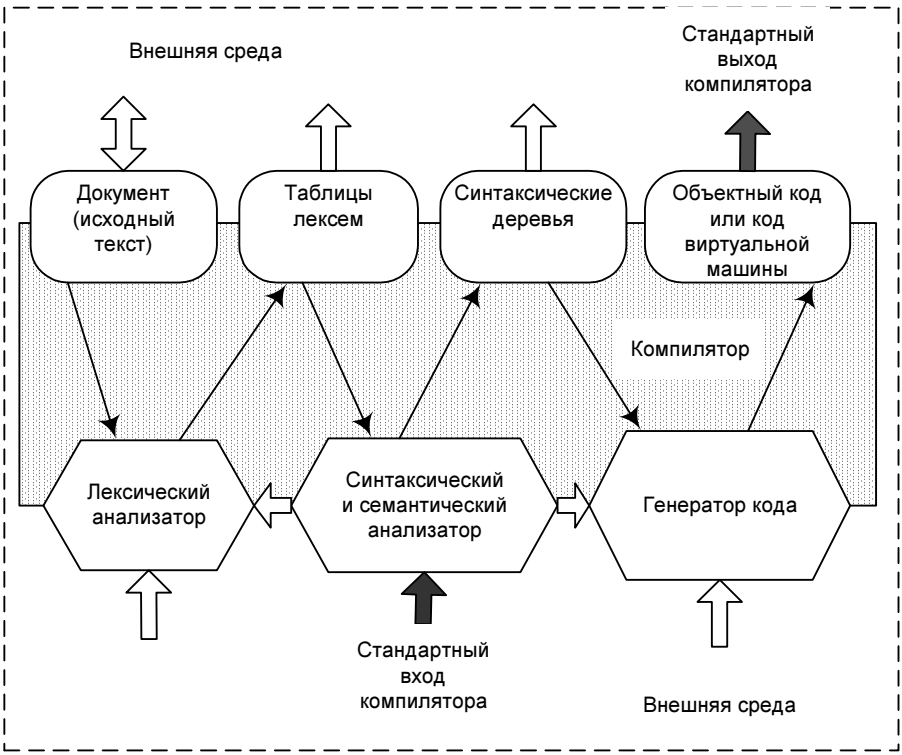


Рис. 1.3. Компонентная организация компилятора

Будучи составной частью множества технологий проектирования, объектно-ориентированная модель программирования поддерживается многими языками (Simula-67, Ada, Smalltalk, Object Pascal, C++, Eiffel, Java, C# и др.). На рис. 1.4 графически представлена генеалогия наиболее распространенных процедурных, объектных и объектно-ориентированных языков. Рисунок конспективно представляет развитие основополагающих концепций программирования в связи с развитием языков программирования.

Подведем итог. *Объектно-ориентированное программирование улучшает проектирование, фокусируясь на данных как более стабильном элементе вычислительной системы. Объектно-ориентированный подход концентриру-*

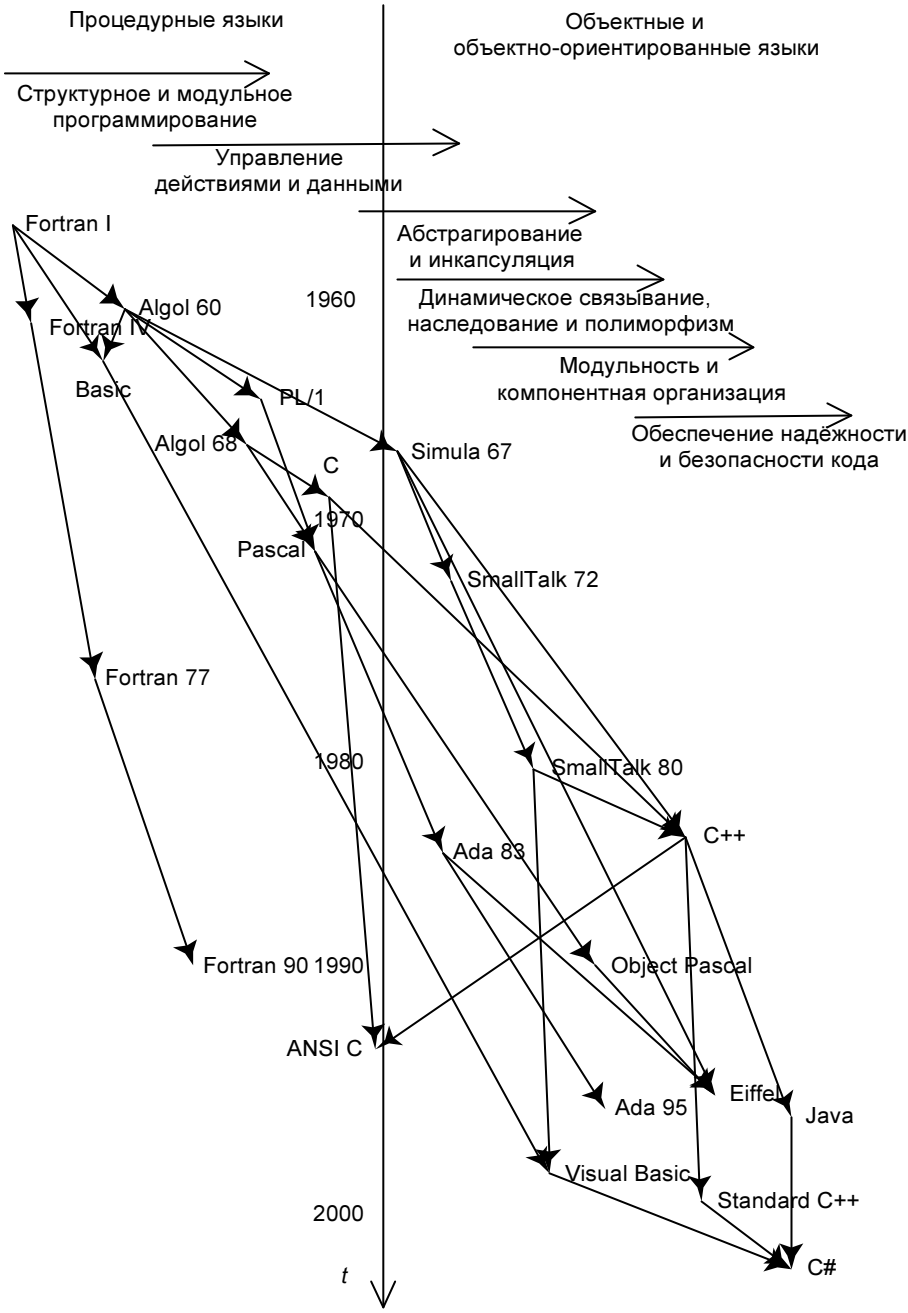
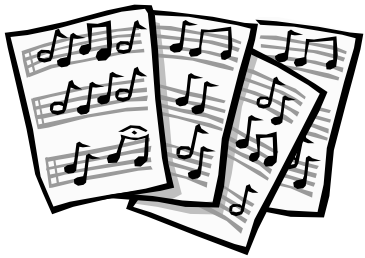


Рис. 1.4. Генеалогия распространенных языков программирования и развитие основных концепций

ется на разработке кода, нацеленного на повторное использование. Объектно-ориентированная модель обеспечивает лучшую масштабируемость проектов. Именно поэтому большинство успешных современных технологий проектирования программных систем предполагает преимущественное или исключительное использование объектно-ориентированной парадигмы.

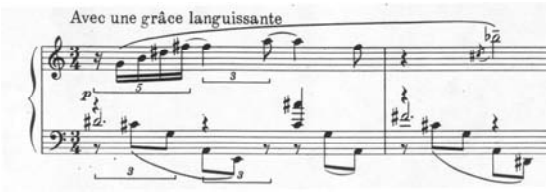
1.4. Вопросы и упражнения

1. Каким образом в рамках процедурного программирования решается вопрос о разграничении доступа к функциям и данным?
2. Какие средства структурного программирования обеспечивают построение моделей данных, присущих предметной области? Как связываются данные и функции для обработки этих данных?
3. Какие преимущества с точки зрения представления данных обеспечивает объектно-ориентированное программирование?
4. Напишите процедурную программу, обеспечивающую вычисление задаваемых в некотором исходном файле арифметических выражений, оперирующих комплексными данными. Выражения могут содержать операции сложения, вычитания, умножения и деления. Синтаксис представления комплексного числа продумайте самостоятельно.
5. Как Вы понимаете упомянутый в *разд. 1.3* термин "масштабируемость программных систем". Обсудите этот термин с коллегами или преподавателем.
6. Почему обеспечение возможности повторного использования является важной задачей проектирования? В связи с этим, как Вы могли бы пояснить смысл тезиса Мейерса *"Программируйте в будущем времени"* [Meyers, 1996]?



ГЛАВА 2

Элементы объектной модели



Александр Скрябин. "Гирлянды", op.73 No.1

По замечанию Буча, каждый стиль программирования имеет свою концептуальную базу. Для объектно-ориентированного стиля концептуальная база — это *объектная модель*.

Объектную модель составляют четыре главных элемента:

- ❒ абстрагирование (abstraction);
- ❒ инкапсуляция (encapsulation);
- ❒ модульность (modularity);
- ❒ иерархия (hierarchy).

Абстрагирование позволяет выделить существенные характеристики некоторого объекта, отличающие его от всех других видов объектов. Абстракция четко определяет концептуальные границы объекта с точки зрения наблюдателя.

Инкапсуляция — это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция также служит для того, чтобы изолировать внешнее поведение объекта от его внутреннего устройства.

Иерархия — это упорядочение абстракций, средство классификации объектов, систематизация связей между объектами.

Модульность — это представление системы в виде совокупности обособленных сегментов, связь между которыми обеспечивается посредством связей между классами, определяемыми в этих сегментах.

Теперь рассмотрим назначение и взаимодействие этих элементов более подробно.

2.1. Смысл абстрагирования как элемента объектной модели

Абстрагирование является одним из основных методов, используемых для решения сложных задач. Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя [Booch, 1994]. В этом определении все элементы важны. Например, любой автомобиль имеет корпус и колеса, причем корпуса автомобилей могут весьма существенно отличаться друг от друга. Тем не менее мы вряд ли спутаем автомобиль с телегой, которая, вообще говоря, тоже имеет корпус и колеса, т. е. абстракции автомобиля и телеги отличаются друг от друга, несмотря на то, что они имеют некоторые общие свойства.

Аналогично понятиям теории абстрактных систем (см. [Mesarovic, Takahara, 1989]), под абстрактными моделями, используемыми в объектно-ориентированном проектировании, понимается не что-то воображаемое или не имеющее отношения к действительности, а модели, характеризуемые наиболее общими свойствами, актуальными для практических случаев.

Объектно-ориентированное программирование характеризуется наличием двух основных видов абстракций:

- Тип данных объектной природы (класс) — определяемое программистом расширение исходных типов языка.
- Экземпляр класса (объект) — переменная класса. Объектов в сложной системе обычно гораздо больше, чем классов. Объект обладает состоянием, поведением и идентичностью.

Состояние объекта характеризуется набором его свойств (атрибутов) и текущими значениями каждого из этих свойств (автомобиль заведен, фары включены).

Состояние объекта — результат его поведения, т. е. выполнения в определенной последовательности характерных для него действий (при включении переключателя указателя поворотов лампочка указателя начинает мигать).

Идентичность — это такое свойство (или набор свойств) объекта, которое позволяет отличить его от всех прочих объектов того же типа. Идентичность

не обязательно связана с именованием или состоянием объекта: два совершенно одинаковых автомобиля, стоящих в одном и том же гараже, все же являются разными объектами. Свойством, отвечающим за идентичность, в этом случае, может считаться VIN автомобиля (Vehicle Identification Number — идентификационный номер автотранспортного средства).

В качестве простейшего примера абстракции, интуитивно понятного всем, Эккель в своих книгах (см., например, [Eckel, 2000-1]) приводит электрическую лампочку, для которой определены, как минимум, два присущих ей состояния: включена (светит) и выключена (не светит) (рис. 2.1).

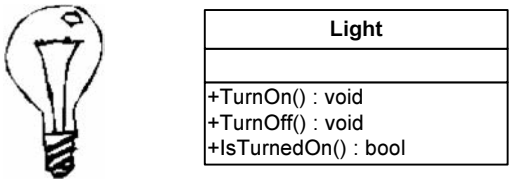


Рис. 2.1. Абстракция электрической лампы

Процесс включения и выключения лампочки описывает поведение объекта. Включению лампы на уровне абстрактной модели соответствует функция `TurnOn()`, выключению — функция `TurnOff()`. Для того чтобы узнать, светится лампа или нет, нам достаточно посмотреть на нее. На уровне абстрактной модели этот процесс представлен специальной функцией `IsTurnedOn()`, позволяющей установить текущее состояние модельного объекта. На диаграмме класса знак "плюс" рядом с именем каждой функции показывает, что эти функции составляют открытый интерфейс класса.

Все лампочки обладают примерно таким поведением, хотя их реализации могут отличаться. Принцип работы вольфрамовых, галогенных, газовых ламп различен, однако любая из них распознается нами как осветительное устройство, которое можно включить и выключить. Какие физические и химические процессы участвуют в ходе перехода лампы из выключенного состояния во включенное, конкретного потребителя может не интересовать. Ему может быть достаточно представления о том, где и как правильно использовать это устройство. Конструируя абстрактную модель, программист в первую очередь пытается выразить наиболее общие свойства моделируемых объектов.

Объявление на языке C++, которое соответствует данной модели, может выглядеть следующим образом:

```
class Light { // Класс Light
public:
    void TurnOn();
```

```
    void TurnOff();
    bool IsTurnedOn();
};
```

Класс определяет общие свойства объектов (т. е. тип). Когда покупатель приходит в магазин и просит дать ему электрическую лампочку, то, выражаясь терминами объектно-ориентированного программирования, он оперирует именем класса (указывает тип интересующего его изделия). Продавец выдает конкретную лампочку — объект класса. Ее-то и можно вкрутить в патрон. Переходя на уровень программной модели, организацию работы можно представить следующим образом: для того чтобы воспользоваться функциональностью открытого интерфейса класса, необходимо создать объект данного класса (который и является настоящей моделью лампы). Для управления объектом в программе определяется соответствующая переменная:

```
// Фрагмент кода на C++ (объект размещается в стеке приложения)
Light light1; // Объект класса Light

light1.TurnOn(); // "включить"
light1.TurnOff(); // "выключить"
```

Что хранится в памяти, выделяемой для переменной `light1`, вообще говоря, зависит от особенностей конкретного языка. В C++ объектная переменная, определенная таким образом, является действительным экземпляром класса, т. е. хранит информацию, характеризующую сам объект. C++ предоставляет также возможность разместить объект в динамической памяти, при этом адрес объекта хранится в переменной типа указатель на класс:

```
// Фрагмент кода на C++ (объект размещается в динамической памяти)
Light *pLight1; // Указатель на объект класса Light
pLight1 = new Light; // Размещение объекта в динамической памяти

pLight1->TurnOn(); // "включить"
pLight1->TurnOff(); // "выключить"
```

Таким образом, решение о том, где будет размещаться объект, принимается программистом. В языке Java объявление объектной переменной всегда определяет не сам объект, а ссылку на него. Все объекты Java размещаются в динамической памяти — для размещения объекта используется специальное предложение языка:

```
// Фрагмент кода на Java
Light light1 = new Light(); // Ссылка, связанная с объектом
                            // класса Light

light1.TurnOn(); // "включить"
light1.TurnOff(); // "выключить"
```

Рис. 2.2 иллюстрирует различия основных механизмов размещения объектов в памяти, используемых языками C++ и Java.

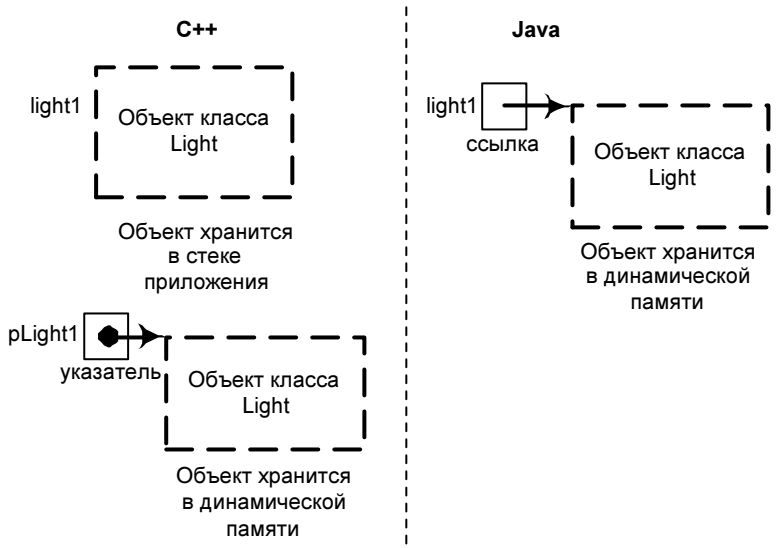


Рис. 2.2. Различия механизмов размещения объектов, используемых языками C++ и Java

Итак, абстрагирование отвечает за внешнее поведение объекта ("интерфейс") — так ведут себя все объекты данного класса, иными словами, *абстрагирование направлено на наблюдаемое поведение объекта (а не на внутреннюю реализацию).*

2.2. Смысл инкапсуляции как элемента объектной модели

Инкапсуляция тесно связана с абстракцией. Если абстрагирование направлено на наблюдаемое поведение объекта, то инкапсуляция занимается внутренним устройством и поэтому определяет четкие границы между различными абстракциями. Механическая коробка передач для Audi не подходит к "Жигулям" из-за различий в реализации, несмотря на общий внешний интерфейс.

Другое важное свойство инкапсуляции — возможность скрытия реализации (hiding an implementation). Инкапсуляция всегда предполагает возможность ограничения доступа к данным класса. С одной стороны, это позволяет упростить интерфейс класса, показав наиболее существенные для внешнего пользователя данные и методы. С другой стороны, скрытие реализации обеспечивает возможность внесения изменений в реализацию класса без изменения

других классов. Данный аспект очень важен с точки зрения дальнейшего сопровождения и модернизации программного кода.

Для простейшей абстракции электрической лампы уровень реализации связан с тем, каким образом обеспечивается представление состояния лампы. Например, это можно сделать с использованием булевской переменной (см. диаграмму на рис. 2.3). Знак "минус" у атрибута `turnedOn` на диаграмме показывает, что доступ к нему извне закрыт. Эта означает, что изменить значение переменной `turnedOn` можно только используя функции, составляющие внешний интерфейс (в нашем случае — функции `TurnOn()` и `TurnOff()`). Таким образом, если попытаться найти на диаграмме наиболее точный графический эквивалент инкапсуляции, то, вероятно, следует считать им черту, отделяющую закрытый для доступа извне атрибут `turnedOn` от открытых методов класса!

Уточненное определение класса `Light` на C++ выглядит следующим образом:

```
class Light {
private: // спецификатор, указывающий на недоступность извне
    bool turnedOn; // недоступно извне
public: // спецификатор, указывающий на открытость членов класса
    void TurnOn();
    void TurnOff();
    bool IsTurnedOn();
};
```

Light
-turnedOn : bool = false
+TurnOn() : void +TurnOff() : void +IsTurnedOn() : bool

Рис. 2.3. Отделение интерфейса от реализации

Реализация функций-членов класса `Light` может выглядеть следующим образом:

```
void Light::TurnOn() {
    turnedOn = true;
}
void Light::TurnOff() {
    turnedOn = false;
}
```

```
bool Light::IsTurnedOn() {
    return turnedOn;
}
```

Таким образом, закрытый член класса `turnedOn` доступен только из функций-членов класса. Нет легальной возможности изменить переменную `turnedOn` извне. В данном примере функции-члены класса работают с единственной переменной-членом класса, которая и отвечает за состояние объекта, а объект класса `Light` хранит единственное значение булевского типа (рис. 2.4).

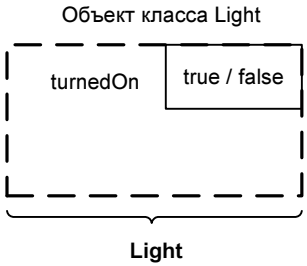


Рис. 2.4. Объект класса `Light`

Листинг 2.1 содержит законченную программу, содержащую определение класса `Light` и пример создания и использования объектов класса.

Листинг 2.1. Создание и использование объектов класса

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

/*
 * Определение класса
 */
class Light {
private: // спецификатор, указывающий на недоступность извне
    bool turnedOn; // недоступно извне
public: // спецификатор, указывающий на открытость членов класса
    void TurnOn() {
        turnedOn = true;
    }
    void TurnOff() {
        turnedOn = false;
    }
```

```
    bool IsTurnedOn() {
        return turnedOn;
    }
};

/*
 * Основная программа: иллюстрирует создание и использование
 * объектов класса
 */
void main() {

    Light light1; // Создаем объект
                  // (автоматическая переменная — хранится в стеке)
    light1.TurnOn();
    if( light1.IsTurnedOn() ) cout << "Light1: On" << endl;
    else                      cout << "Light1: Off" << endl;

    light1.TurnOff();
    if( light1.IsTurnedOn() ) cout << "Light1: On" << endl;
    else                      cout << "Light1: Off" << endl;

    Light *pLight2 = new Light;
                  // Определяем указатель на объект
                  // и связываем с ним объект, размещаемый в
                  // динамической памяти

    pLight2->TurnOn();
    if( pLight2->IsTurnedOn() ) cout << "Light2: On" << endl;
    else                        cout << "Light2: Off" << endl;

    pLight2->TurnOff();
    if( pLight2->IsTurnedOn() ) cout << "Light2: On" << endl;
    else                        cout << "Light2: Off" << endl;
}
```

В программе листинга 2.1 определения функций-членов класса располагаются непосредственно внутри определения класса. В данном случае это оправдано, т. к. программа состоит из одного модуля. Однако такое определение методов класса приемлемо не во всех случаях. *Полное представление об организации определений класса и методов класса в программе читатель получит после ознакомления с информацией, содержащейся в гл. 3—5, 9.*

Заметим, что инкапсуляция не обеспечивает полной закрытости классов. Она, по замечанию Страуструпа, "защищает от ошибок, а не от жульничества". Рассмотрите в качестве иллюстрации пример, представленный в листинге 2.2.

Листинг 2.2. Инкапсуляция защищает от ошибок, а не от жульничества

```
// Код на C++
// Внимание! Данный пример иллюстрирует возможность, которую
//           нельзя рекомендовать для применения на практике!
#include <iostream>
using namespace std;
class Test {
private:
    int x;
public:
    Test() {
        x = 1;
    }
    void PrintX() {
        cout << x << endl;
    }
};

void main() {
    Test *t = new Test;
    int *pi = (int *) t;
    t->PrintX(); // Будет напечатано: 1

    *pi = 100; // "Нелегальный" доступ к закрытым данным

    t->PrintX(); // Будет напечатано: 100
}
```

Реализация условий доступа к закрытым данным отличается в разных языках программирования. Например, в Smalltalk все данные закрыты, а все методы открыты: защита от прямого доступа к данным обеспечивается на стадии компиляции.

В Object Pascal доступ к закрытым данным возможен из всех классов, расположенных в том же модуле (unit), что и анализируемый класс. Таким образом, полной закрытости нет.

В C++ управление доступом и видимостью реализовано достаточно гибко. Члены класса могут быть объявлены открытыми для всех внешних объектов (public), полностью закрытыми для доступа извне (private) и защищенными, т. е. видимыми для всех экземпляров данного класса и его подклассов (protected). *Детально права доступа к различным элементам класса обсуждаются в гл. 3, 4.*

Заметим, что закрытые данные закрыты, в том числе и для объектных переменных данного класса. Кроме того, в C++ существует понятие дружественных классов и функций (friend). Дружественным функциям и классам могут быть доступны и закрытые данные.

Резюме: *внутренняя реализация описывает, каким образом достигается желаемое поведение объекта. Инкапсуляция изолирует интерфейс от реализации. Инкапсуляция связана с управлением доступом к данным класса. Существуют отличия в реализации управления доступом к данным класса в разных языках программирования.*

Отметим, что в связи с учебным характером этой книги, в данном разделе понятие инкапсуляции вводится несколько упрощенно. В действительности, обеспечение инкапсуляции не всегда связано с сокрытием реализации. Мейерс показывает, что во многих реалистичных проектных ситуациях реализация части функциональности в связи с обработкой объектов класса в виде внешних функций (а не методов класса) может даже улучшать инкапсуляцию класса (см. [Meyers, 1995], [Meyers, 2000]).

2.3. Отношения классов

Если рассматривать ООП как программирование, сфокусированное на данных, а вычислительный процесс как моделирование поведения [Pohl, 1997], то объектом изучения должны, прежде всего, стать формы организации структур данных, присущих объектно-ориентированному программированию, и формы взаимодействия объектов внутри программной системы, построенной с использованием ООП.

Основные отношения классов. Ориентируясь на спецификацию UML, можно определить четыре базовых вида отношений между объектно-ориентированными абстракциями [Booch, Rumbaugh, Jacobson, 1999].

- Отношение *обобщения* (generalization) описывает отношение между общей сущностью и ее конкретным воплощением (например, один класс является специализацией другого класса).
- Отношение *ассоциации* (association) описывает структурное отношение, показывающее, что объекты одного типа некоторым образом связаны с объектами другого типа (например, находятся в отношении типа "часть/целое").
- Отношение *зависимости* (dependency) описывает отношение использования, при котором изменение в спецификации одного класса может повлиять на класс, его использующий (например, объекты некоторого класса передаются в качестве аргументов функциям-членам другого класса).

- Отношение *реализации* (realization) описывает семантическое отношение, при котором класс гарантирует выполнение контракта, определяемого некоторым интерфейсом. Обычно определяется применительно к интерфейсам и классам, которые реализуют (implement) интерфейсы.

Поясним смысл этих определений с использованием примера конструирования объектной модели для гирлянды из цветных лампочек, подобной тем, что вешают на елку перед новогодним праздником [Пышкин, 2003-1]. Представим приложение, которое должно моделировать создание и использование такой гирлянды. Этот довольно простой и вместе с тем наглядный пример позволит нам проиллюстрировать все основные виды отношений между классами.

Отношение обобщения

Конструирование классов. Итак, наша гирлянда должна состоять не из обычных лампочек, а из цветных. Понятно, что цветная лампочка может считаться разновидностью обычной, однако при этом обладает, по крайней мере, одним дополнительным свойством. Абстрактная модель такой лампочки, представленная в отношении к обычной лампочке, изображена на рис. 2.5.

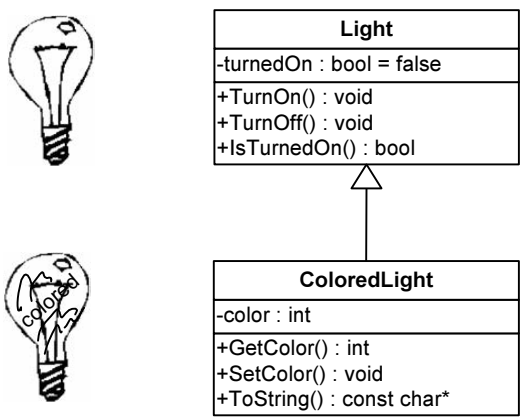


Рис. 2.5. Класс `ColoredLight` расширяет определение `Light`

Класс `ColoredLight` расширяет определение `Light`, наследуя его элементы и добавляя к этим элементам новые. При этом говорят, что класс `ColoredLight` наследует классу `Light`. Отношение наследования является, вероятно, самым распространенным типом отношения обобщения. В соответствии со спецификацией UML, отношение обобщения изображается на диаграмме классов в виде сплошной линии с незакрашенной стрелкой, направленной от дочернего (более специализированного) элемента к родительскому (более общему).

В своей предварительной версии класс `ColoredLight` добавляет к методам, унаследованным из класса `Light`, методы, позволяющие установить или узнать цвет лампочки, а также (для задач последующего тестирования) — функцию, которая представляет цветовую характеристику в виде строкового литерала.

Для упрощения реализации мы ограничились представлением цветовой характеристики в виде целого числа, что не слишком нарушает общность представления подобных данных в виде упакованных наборов двоичных разрядов.

Фрагмент исходного кода, построенного на основе данного отношения, может выглядеть следующим образом:

```
class Light { // Базовый класс: воплощает наиболее общую абстракцию
private:
    bool turnedOn;
public:
    void TurnOn() {
        turnedOn = true;
    }
    void TurnOff() {
        turnedOn = false;
    }
    bool IsTurnedOn() {
        return turnedOn;
    }
};

class ColoredLight : public Light { // Производный класс: наследует
                                    // свойства базового класса
private:
    int color;
public:
    void SetColor( int customColor );
    int  GetColor();
    const char *ToString();
}
```

Используя объект производного класса, можно обращаться к открытым членам базового класса:

```
ColoredLight cl; // Объект-экземплярная переменная
cl.SetColor( 0 );// Обращение к методу класса ColoredLight
cl.TurnOn();     // Обращение к методу класса Light
```

```
ColoredLight pCl = new ColoredLight; // Указатель на объект
                                     // класса ColoredLight
pCl->TurnOn();   // Обращение к методу класса Light
```

Анализ структуры объекта. Теперь перейдем на уровень объектов класса и проанализируем структуру информации, хранящейся в объекте класса `ColoredLight` (рис. 2.6).

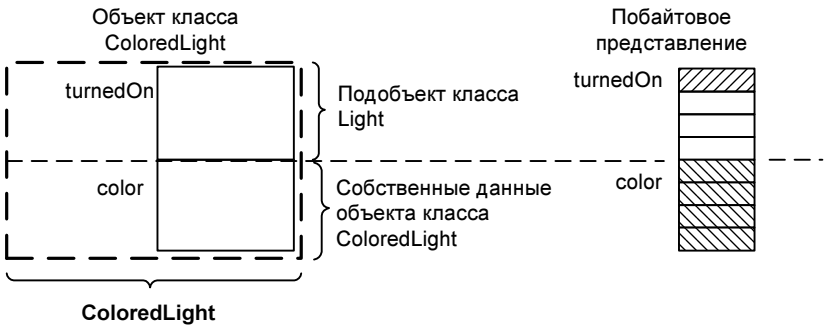


Рис. 2.6. Подобъект базового класса `Light` в составе объекта производного класса `ColoredLight`

Поскольку класс `ColoredLight` наследует классе `Light`, то часть объекта составляют данные, определенные в базовом классе (в нашем примере — это единственная переменная `turnedOn`). Часть объекта, обусловленная базовым классом, обычно называется подобъектом базового класса в составе объекта производного класса. В рассматриваемом примере переменная `turnedOn` недоступна для объектов базового класса, поскольку является закрытым (`private`) членом класса, следовательно она тем более недоступна и для объектов производного класса. Тем не менее ее значение хранится в составе объектной переменной класса `ColoredLight`. Для того чтобы убедиться в этом, рассмотрим простой пример программы, использующей определения обоих классов и вычисляющей размеры объектов этих классов (листинг 2.3).

Листинг 2.3. Размеры объектов базового и производного классов

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

class Light { // Базовый класс: воплощает наиболее общую абстракцию
private:
    bool turnedOn;
public:
    void TurnOn();
```

```
    void TurnOff();
    bool IsTurnedOn();
};

const int LastColor = 4; // Наибольшее значение в ColorEnum
enum ColorEnum { // Перечисление, определяющее константы,
                 // используемые для представления различных цветов
    WHITE  = 0,
    RED    = 1,
    GREEN  = 2,
    YELLOW = 3,
    BLUE   = 4,
};

class ColoredLight : public Light { // Производный класс: наследует
                                    // свойства базового класса
private:
    int color;
public:
    void SetColor( int customColor ) {
        if( customColor < 0 || customColor > LastColor ) {
            cout << "Incorrect color" << endl;
            return;
        }
        color = customColor;
    }

    int  GetColor() {
        return color;
    }
    const char *ToString() {
        switch( color ) {
        case WHITE  : return "white";
        case RED    : return "red";
        case GREEN  : return "green";
        case YELLOW : return "yellow";
        case BLUE   : return "blue";
        default     : return "white";
        }
    }
};
```

```
/*
 * Основная программа: иллюстрирует создание объектов
 * базового и производного классов и определяет
 * размеры объектов класса
 */
void main() {
    Light light1;
    cout << "Size of light1 object: " << sizeof( light1 ) << endl;

    ColoredLight light2;
    cout << "Size of light2 object: " << sizeof( light2 ) << endl;
}
```

При исполнении данной программы на компьютере с 32-разрядной архитектурой будут выведены следующие результаты:

```
Size of light1 object: 1
Size of light2 object: 8
```

Из этого следует, что объект `light1` (имеющий тип `Light`) занимает в памяти 1 байт, требующийся для хранения одного значения булевского типа. Объект `light2` (имеющий тип `ColoredLight`) занимает в памяти 8 байтов. В первом из четырех байтов располагается закрытая булевская переменная `turnedOn`. Следующие три байта обеспечивают выравнивание до 32-разрядного слова (заметим, что, управляя ключами компилятора, выравнивание можно устранить). Наконец, последние 4 байта занимает целочисленная переменная `color`, объявленная в классе `ColoredLight` (см. рис. 2.6).

Замечание

Заметим, что основная задача, решаемая в этом разделе, заключалась в иллюстрации простейшего варианта наследования и выяснении того, как хранится в памяти информация, относящаяся к объектам базового и производных классов. Рассмотрение других отношений классов позволит нам в дальнейшем уточнить и улучшить определение класса `ColoredLight`.

Отношение ассоциации

Конструирование классов. В конечном счете, гирлянда — это набор цветных лампочек. Для нашей модели это означает, что объект класса, воплощающего абстракцию гирлянды (`Garland`), должен предоставлять возможность работать с множеством объектов класса `ColoredLight`. Диаграмма, представленная на рис. 2.7, иллюстрирует возможную организацию класса `Garland`.

Как следует из диаграммы, в качестве одного из элементов класса `Garland` присутствует массив указателей на объекты типа `ColoredLight`. При этом го-

ворят, что класс `Garland` находится в отношении ассоциации к классу `ColoredLight`. Внешний интерфейс класса определяют три метода:

- метод `TurnOn()`, моделирующий включение гирлянды;
- метод `TurnOff()`, моделирующий выключение гирлянды;
- метод `GetLight()` для доступа к очередному объекту класса `ColoredLight`, моделирующему отдельную лампочку в составе гирлянды. Наличие такого доступа дает возможность управлять отдельными лампочками в составе гирлянды, используя открытый интерфейс класса `ColoredLight`.

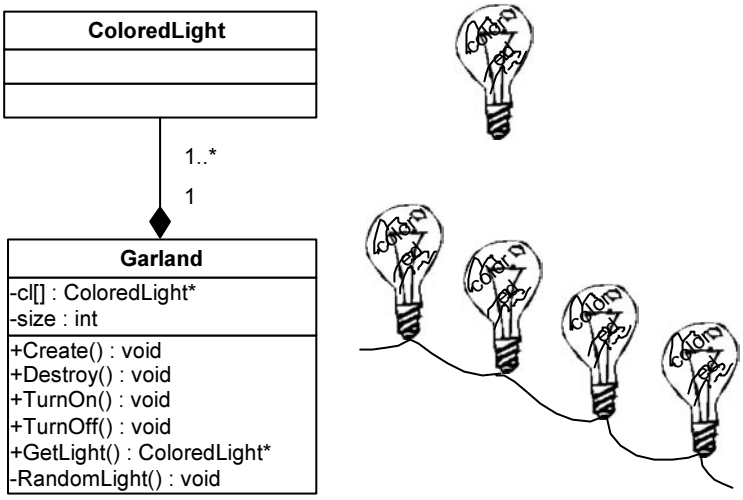


Рис. 2.7. Гирлянда состоит из цветных лампочек

Закрытый метод `RandomLight()` предназначен для случайной генерации объектов `ColoredLight` и должен вызываться в процессе инициализации гирлянды. *Каким образом осуществляется инициализация объектов, мы рассмотрим в гл. 3.*

Фрагмент исходного кода, содержащего определение класса `Garland`, может выглядеть следующим образом:

```
class Garland { // Класс, воплощающий абстракцию гирлянды
private:
    typedef ColoredLight *PColoredLight; // Тип-указатель на ColoredLight
    PColoredLight *cl; // Указатель на начало массива указателей
                       // на объекты типа ColoredLight
    int size;          // Размер гирлянды (число элементов массива cl)
```

```
public:
    void Create( int customSize ); // создать гирлянду заданного размера
    void Destroy(); // разрушить ранее созданную гирлянду
    void TurnOn();  // "включить" гирлянду
    void TurnOff(); // "выключить" гирлянду
    PColoredLight GetLight( int index ); // получить доступ к отдельной
                                         // лампочке
private:
    void RandomLight( PColoredLight& refPLight );
};
```

Анализ структуры объекта. Перейдем к анализу структуры объекта (рис. 2.8).

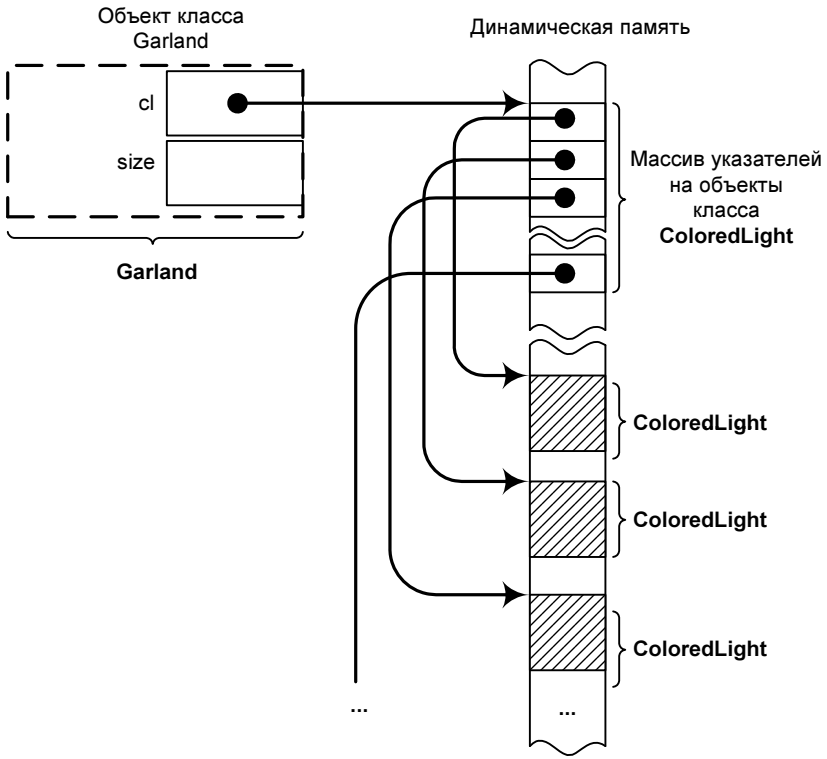


Рис. 2.8. Структура информации в связи с объектом класса `Garland`

Объектная переменная хранит указатель `cl` и целочисленное значение `size`. Таким образом, сам объект занимает в памяти 8 байтов. С указателем `cl` связан размещаемый в ходе инициализации объекта массив указателей на объек-

ты класса ColoredLight. В процессе работы каждый из этих указателей может быть связан с объектами класса ColoredLight. Несложная реализация, обеспечивающая тестирование описанной функциональности, приведена в листинге 2.4.

Листинг 2.4. Реализация класса Garland и использование объектов класса

```
#include <cstdlib>
#include <ctime>
#include <cassert>

#include <iostream>
using namespace std;

class Light {
    // Определение см. в листинге 2.3...
};

const int LastColor = 4;
enum ColorEnum {
    // Определение см. в листинге 2.3...
};

class ColoredLight {
    // Определение см. в листинге 2.3...
};

class Garland {
private:
    typedef ColoredLight *PColoredLight;


    PColoredLight *cl; // Указатель на массив указателей
                       // на объекты класса ColoredLight
    int size; // Размер массива cl (число ламп в гирлянде)

public:
    // Метод, обеспечивающий создание гирлянды
    // задаваемого размера customSize
    void Create( int customSize ) {

        cl = new PColoredLight[ customSize ];
        assert( cl != 0 );
```

```
        for( int ix = 0; ix < customSize; ix++ ) {
            randomLight( cl[ ix ] );
        }
        size = customSize;
    }

    // Метод, обеспечивающий разрушение гирлянды
    // (освобождается занятая динамическая память)
    void Destroy() {

        if( cl != 0 ) {
            // Сначала удаляем объекты класса ColoredLight...
            for( int ix = 0; ix < size; ix++ ) {
                delete cl[ ix ];
            }
            // ... затем — сам массив указателей на объекты
            delete [] cl;
        }
    }

    // Метод возвращает указатель на отдельный объект
    // класса ColoredLight в составе гирлянды
    PColoredLight getLight( int number ) {
        return cl[ number ];
    }

    // "Включение" гирлянды
    void TurnOn() {
        // Последовательно "включаем" все лампы
        for( int i = 0; i < size; i++ )    cl[ i ]->TurnOn();
    }

    // "Выключение" гирлянды
    void TurnOff() {
        // Последовательно "выключаем" все лампы
        for( int i = 0; i < size; i++ )    cl[ i ]->TurnOff();
    }

private:
    // Метод, обеспечивающий генерацию очередного объекта
    // ColoredLight (цвет определяется случайным образом)
    void randomLight( PColoredLight& refPLight ) {
        int randomColor = rand() % ( LastColor+1 );
```

```
        refPLight = new ColoredLight;
        refPLight->SetColor( (ColorEnum ) randomColor);
    }
};

/*
 * Основная программа: иллюстрирует создание объекта
 * класса Garland и его использование
 */
void main() {
    // "Рассеивание" генератора случайных чисел необходимо
    // для корректной работы библиотечной функции rand()
    srand( (unsigned) time( NULL) );

    Garland garland; // объект-гирлянда
    garland.Create( 10 ); // создаем гирлянду из 10 ламп

    // Печатаем информацию о цвете всех ламп
    for( int ix=0; ix < 10; ix++ ) {
        ColoredLight *pCl = garland.getLight( ix );
        cout << pCl->ToString() << endl;
    }

    garland.Destroy(); // разрушаем гирлянду
}
```

Вывод данной программы отличается от запуска к запуску, поскольку "цвета" объектов `ColoredLight` определяются случайным образом, и результат может быть, например, таким:

```
blue
white
red
yellow
yellow
green
green
blue
green
red
```

Об инициализации и разрушении объектов. Некоторый недостаток предложенной реализации заключается в том, что для правильной инициализации гирлянды пользователь класса должен явно вызвать метод `Create()`, равно

как и метод `Destroy()` для разрушения гирлянды. Подобные требования делают возможными такие ошибки, как попытка использования неинициализированного объекта, повторная инициализация объекта, отсутствие корректного разрушения объекта, попытка повторного разрушения объекта. Таким образом, вопрос об обеспечении корректной гарантированной инициализации объекта и его корректного однократного разрушения остается открытым. *Обеспечение этих возможностей средствами языка программирования подробно рассматривается в гл. 3.*

Распространенные типы ассоциаций. Наиболее распространенными ассоциациями являются отношения композиции и агрегирования, при этом композиция, по существу, является вариантом агрегирования с четко выраженным отношением владения. Отличие композиции и простого агрегирования может быть проиллюстрировано следующим примером. Студенты могут учиться в одном или нескольких вузах (таким образом, студент и вуз находятся в отношении простого агрегирования, не являющегося композицией), а факультет может принадлежать только одному вузу (таким образом, факультет и вуз находятся в отношении композиции). Наконец, студент посещает курсы, но, разумеется, не является их составной частью (студент и курс находятся в отношении ассоциации, не являющейся агрегированием) (пример позаимствован из [Booch, 1999]). *Подробнее агрегирование и композиция рассматриваются в разд. 2.4 данной главы.*

В соответствии со спецификацией UML, графически ассоциация изображается в виде сплошной линии, рядом с которой могут присутствовать дополнительные обозначения (например, кратность, имена ролей, имя самой ассоциации). Линия может завершаться стрелкой в форме ромба. Ромб, завершающий стрелку на диаграмме UML, закрашивается, если речь идет о композиции, и не закрашивается в случае простого агрегирования.

Отношение зависимости

Отношение зависимости является таким типом отношений между классами, когда изменение в спецификации или реализации одного класса влияет на спецификацию или реализацию другого класса. Пример отношения зависимости между классами представлен на рис. 2.9.

В данном случае два метода класса `GarlandApp`, определяющего функциональность приложения, принимают аргументы, тип которых содержит имя типа `Garland` (в одном случае — это адрес указателя на объект класса `Garland`, в другом — указатель на объект класса `Garland`). Таким образом, на реализацию методов `CreateGarland()` и `TestGarland()` могут влиять изменения в определении класса `Garland`.

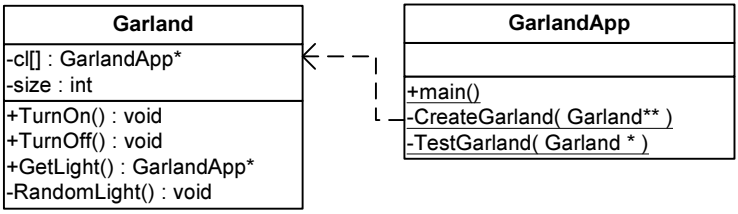


Рис. 2.9. Зависимость `GarlandApp` от реализации `Garland`

Графически отношение зависимости изображается пунктирной линией, направленной от данного элемента диаграммы к тому элементу, от которого он зависит.

Отношение реализации

Чаще всего отношение реализации используется для определения отношения между интерфейсом и классом, реализующим интерфейс. Интерфейс определяет набор элементов (как правило, действий), характерных для объектов, обладающих определенными свойствами.

Поскольку действиям соответствуют методы, то интерфейс определяет сигнатуры методов (т. е. определяет то, как должны называться методы, сколько они должны принимать аргументов, какой тип должны иметь аргументы и возвращаемое значение). Тем самым, интерфейс определяет так называемый контракт. Класс, поддерживающий данный интерфейс, должен обеспечить реализацию контракта, устанавливаемого интерфейсом.

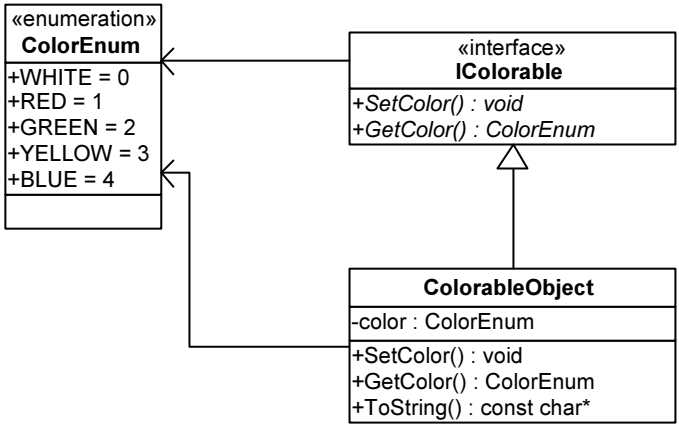


Рис. 2.10. Класс `ColorableObject` предоставляет реализацию интерфейса, определяемого `IColorable`

Рис. 2.10 представляет вариант определения класса `ColorableObject` на основе интерфейса `IColorable`. Проиллюстрируем на этом примере основную идею, лежащую в основе концепции интерфейса. Цветом могут характеризоваться не только лампочки, но самые разнообразные объекты. В рамках программной модели было бы неплохо иметь какой-нибудь вполне унифицированный механизм работы с такими объектами. Интерфейс `IColorable` "в кооперации" с типом-перечислением `ColorEnum` предоставляет простейшую модель такого механизма. Тип `ColorEnum` содержит элементы перечисления, которые мы будем использовать для идентификации цвета. Интерфейс `IColorable` использует этот тип для определения сигнатур двух методов `SetColor()` и `GetColor()`, которые должен поддерживать любой объект, характеризуемый цветом. Интерфейс не предоставляет реализацию методов. То есть интерфейс `IColorable` — это, по существу, чистая абстракция объекта, характеризуемого цветом.

Определив для любого класса отношение реализации интерфейса `IColorable`, разработчик должен предоставить реализацию методов `SetColor()` и `GetColor()`, соблюдая тем самым контракт, установленный интерфейсом (в рассматриваемом примере классом, реализующим интерфейс `IColorable`, является класс `ColorableObject`).

В такой интерпретации `ColoredLight` может трактоваться как разновидность `ColorableObject`. Можно реализовать и другие разновидности `ColorableObject`.

Графически отношение реализации изображается в виде пунктирной линии с незакрашенной стрелкой, направленной от сущности, обеспечивающей реализацию, к сущности, определяющей контракт.

Наличие интерфейсов предоставляет возможность работать с объектами разных типов в той части, которая поддерживает соответствующий интерфейс. Это означает, что программист имеет возможность написать общий код для объектов самых разных типов, если этот код затрагивает только ту функциональность, которая связана с реализуемым интерфейсом, например:

```
void SetWhite( IColorable *ic_supported ) {

    ic_supported->SetColor( WHITE );
}
```

Данной функции можно передать указатель на любой объект, поддерживающий интерфейс `IColorable`. *Механизмы обеспечения такой возможности рассматриваются в гл. 5.*

Отметим, что большинство современных языков поддерживает интерфейсы в виде встроенных средств языка (включая Java, C#, Visual C++ .NET with Managed Extentions). C++ напрямую не поддерживает интерфейсы, однако

эквивалентная им семантика может быть выражена с помощью абстрактных классов.

На основе рассмотренных фрагментов, иллюстрирующих отдельные виды отношений между классами, можно построить полную иерархию классов проекта "Гирлянда из цветных лампочек" (Garland Application) — результирующая диаграмма приведена на рис. 2.11.

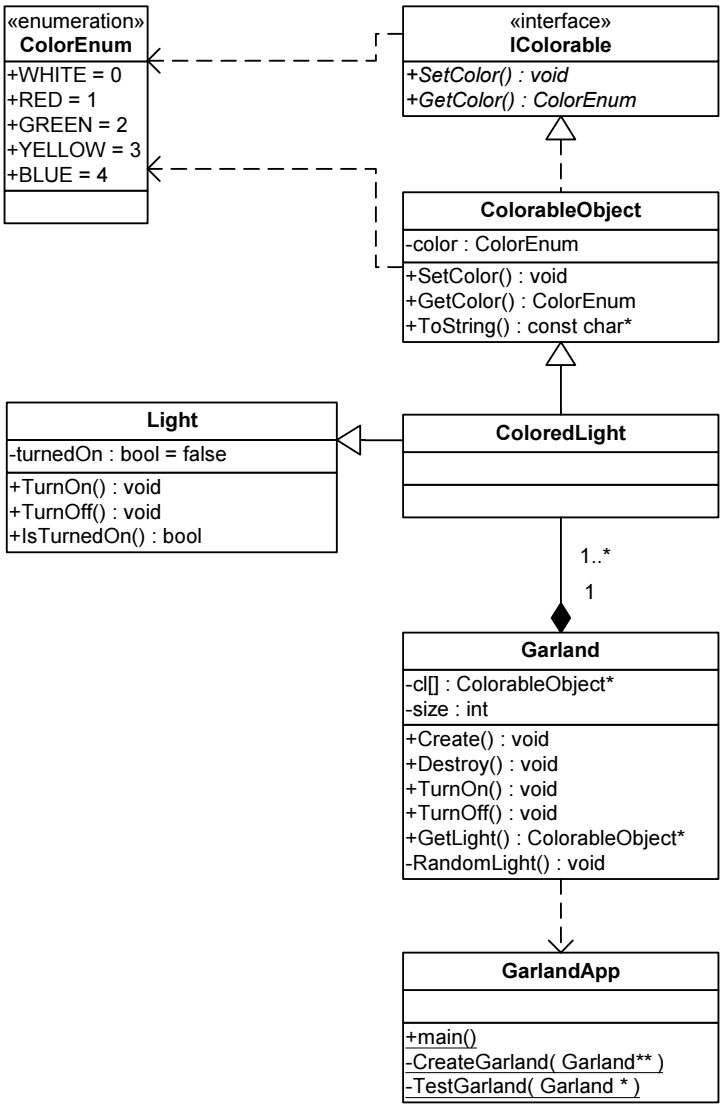


Рис. 2.11. Диаграмма классов Garland Application

Из данной диаграммы видно, что в рамках проекта Garland Application используются все рассмотренные в этой главе отношения между классами. Спецификация класса `ColoredLight` упростилась настолько, что он вообще не содержит собственных элементов, а объединяет функциональности классов `Light` и `ColorableObject`, используя множественное наследование *(подробности см. в гл. 7)*.

Замечание

Приведенная диаграмма является эскизным вариантом, поскольку за скобками пока остались вопросы инициализации объектов, которые будут обсуждаться в *гл. 3*.

2.4. Типы структурных иерархий

Известно, что изучение и организация сложных технических (и не только технических) систем существенно упрощается, если следовать принципам функционально-иерархической декомпозиции. Иерархическая декомпозиция и иерархическая организация ПО образуют один из основных систематических методов преодоления сложности программного обеспечения [Лекарев, 1997]. В этом разделе мы акцентируем внимание на понимании сложной системы как совокупности иерархий и анализируем основные средства реализации структурных иерархий в объектно-ориентированной программе.

Структурная иерархия "is-part-of" (агрегирование как разновидность ассоциации)

Типичным вариантом отношения ассоциации является структурная иерархия, построенная по принципу "is-part-of" ("является частью"), обычно называемая агрегированием. Агрегирование является простейшей реализацией одной из базовых идей, лежащих в основе ООП: созданный класс в идеале должен представлять собой фрагмент полезного кода, доступного для повторного использования. Это не означает, что при проектировании каждого класса нужно обеспечить возможность его использования в любом стороннем коде, однако учет фактора повторного использования в целом способствует проектированию удачных классов, нацеленных на обеспечение решения вполне конкретных задач.

Напомним, что вариант агрегирования, при котором отношение владения четко выражено, а время жизни частей и целого совпадают (таким образом, часть существует только в контексте целого), называется композицией. Простое агрегирование в отличие от композитного агрегирования предполагает, что одна и та же "часть" может одновременно принадлежать разным "целым" (например, одна стена может принадлежать нескольким комнатам). В даль-

нейшем обычно мы будем подразумевать именно композитное агрегирование.

Отношение композиции образуют, например, автомобиль как целое и двигатель, трансмиссия, рулевое управление и тормозная система как его части.

Механизм агрегирования предоставляет программисту гибкое средство организации иерархии классов. Члены создаваемого класса, объявленные как `private`, недоступны для клиентских программ, содержащих классы, агрегирующие данный класс. Это позволяет вносить изменения, затрагивающие особенности внутренней реализации, не влияя на существующий клиентский код.

Структурные иерархии "is-a" и "is-like-a"

Идея наследования вытекает из желания иметь удобное средство, которое позволило бы создавать класс, данные и поведение которого в некоторой части сходны с существующим классом. Производя новый класс (называемый производным) из существующего (называемого базовым), мы получаем возможность сохранить функциональность базового класса, а также дополнить его или, возможно, изменить его поведение. Наследование является частным случаем отношения обобщения. Обобщение означает, что объекты класса-потомка могут использоваться везде, где допустимы объекты родительского класса. Обратное в общем случае неверно.

Создавая базовый тип (базовый класс), программист выражает наиболее общие идеи относительно объектов, из которых конструируется программа. В производных классах программист уточняет различия в реализации конструкций базового класса.

Когда вы конструируете новый класс, наследуя от базового класса, вы создаете новый тип. Этот новый тип включает все данные и методы базового типа (хотя члены базового типа с классом доступа `private` скрыты и недоступны в производном классе). Следовательно, производный класс полностью дублирует интерфейс базового класса, т. е. дублирует наблюдаемое поведение объекта базового класса. Это означает, что все сообщения, которые можно послать базовому классу, можно послать и производному классу. Поэтому часто говорят, что объекты производного класса имеют тот же тип, что и объекты базового класса (точнее речь идет о совместимости типов в направлении от производного класса к базовому — результат достигается стандартным преобразованием). Соответствие типа производного класса типу базового класса является одним из фундаментальных принципов реализации концепции наследования в рамках объектной модели.

Несмотря на указанное соответствие типов, поведение объекта производного класса может отличаться от поведения объекта базового класса. Таким обра-

зом, производный класс может иначе реализовывать (implement) интерфейс базового класса.

В качестве простого (ставшего уже практически каноническим) примера построения иерархии классов, использующей наследование, в различных источниках приводится пример с геометрическими фигурами.

Существуют два основных пути изменить поведение производного класса по отношению к базовому классу. Первый заключается в добавлении новых методов, реализующих возможности, которых нет в базовом классе (рис. 2.12). Так, в классы `Circle`, `Square`, `Triangle` включен ряд элементов, специфических для данных типов фигур и отсутствующих в классе, реализующем наиболее общую абстракцию геометрической фигуры `Shape`. При этом в классах, унаследованных от `Shape`, можно использовать методы `Draw()` и `Erase()`, объявленные в классе `Shape`. Дополнительно в классе `Triangle` представлен метод `Rotate90()`, которого нет в базовом классе, а также в прочих производных от `Shape` классах. Действительно, объекты классов `Circle` и `Square` вряд ли нуждаются в специальном методе для поворота на 90° (изображение от этого не изменится). В том случае, когда производный класс просто расширяет функциональность базового класса, говорят о структурной иерархии типа "is-a".

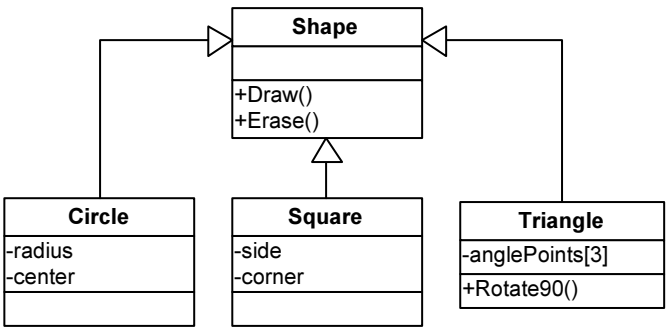


Рис. 2.12. Расширение возможностей, обеспечиваемых производными классами по отношению к базовому классу

Схема определений классов, изображенных на рис. 2.12, выглядит следующим образом (листинг 2.5).

Листинг 2.5. Иерархия классов "is-a"

```
/*
 * Определения классов приведены для иллюстративных целей,
 * поэтому методы класса просто печатают на консоль сообщения,
 * подтверждающие факт выполнения метода
 */
```

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

typedef struct {
    double x;
    double y;
} Point;

class Shape {
public:
    void Draw() {
        cout << "Shape::Draw()" << endl;
    }
    void Erase() {
        cout << "Shape::Erase()" << endl;
    }
};

class Circle : public Shape {
    double radius;
    Point  center;
};

class Square : public Shape {
    double side;
    Point  corner;
};

class Triangle : public Shape {
    Point anglePoints;
public:
    void Rotate90() {
        cout << "Triangle::Rotate90()" << endl;
    }
};

// Пример использования объектов
void main() {
    Shape shapeObj;
    shapeObj.Draw();        // печатается "Shape::Draw()"
    shapeObj.Erase();       // печатается "Shape::Erase()"

    Circle circleObj;
    circleObj.Draw();       // печатается "Shape::Draw()"
    circleObj.Erase();      // печатается "Shape::Erase()"
```

```
    Square squareObj;
    squareObj.Draw();       // печатается "Shape::Draw()"
    squareObj.Erase();      // печатается "Shape::Erase()"

    Triangle triangleObj;
    triangleObj.Draw();     // печатается "Shape::Draw()"
    triangleObj.Erase();    // печатается "Shape::Erase()"
    triangleObj.Rotate90(); // печатается "Triangle::Rotate90()"
}
```

Однако иногда нет необходимости в добавлении новых методов в производном классе, или же этого недостаточно. Взамен может потребоваться изменение реализации уже существующих методов (рис. 2.13). Для этого дочерний класс предоставляет операции, замещающие операции родителя с такой же сигнатурой, т. е. видоизменяющие функциональность базового класса (термин "операция" используется здесь в самом общем смысле — *см. также информацию во введении*). В этом случае говорят о структурной иерархии типа "is-like-a". Механизм изменения реализации метода класса с сохранением интерфейса, определяемого базовым классом, называется *замещением* (overriding) *метода класса*.

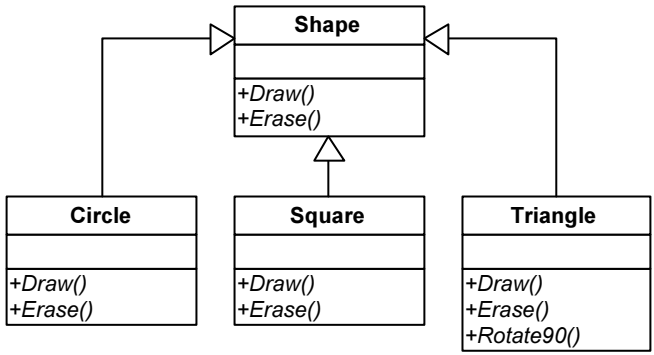


Рис. 2.13. Изменение реализации существующих методов в производных классах

Действительно, алгоритм прорисовки и стирания изображения графического объекта во многом определяется формой изображаемой фигуры, т. е. ее математической моделью. Соответствующие реализации отличаются для разных геометрических фигур. При этом интерфейс разумно сохранять неизменным.

Таким образом, класс `Shape` предоставляет общий интерфейс (задает контракт) для методов `Draw()` и `Erase()` и, возможно, содержит какие-нибудь другие методы. Классы `Circle`, `Square` и `Triangle` предоставляют реализации

этого интерфейса (реализуют контракт). Поведение объектов, на которых вызываются видоизмененные методы, интерфейс к которым заявлен в базовом классе, называют полиморфным, а механизм, обеспечивающий такое поведение, называют *полиморфизмом*. Схема реализации классов C++, содержащих замещающие методы, представлена в листинге 2.6.

Листинг 2.6. Замещающие методы в иерархии классов (иерархия "is-like-a")

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

typedef struct {
    double x;
    double y;
} Point;

class Shape {
public:
    virtual void Draw() {  // Спецификатор "virtual" указывает на то,
                           // что данный метод может быть замещен
                           // (overridden)
        cout << "Shape::Draw()" << endl;
    }
    virtual void Erase() { // Спецификатор "virtual" указывает на то,
                           // что данный метод может быть замещен
                           // (overridden)

        cout << "Shape::Erase()" << endl;
    }
};

class Circle : public Shape {
    double radius;
    Point  center;
public:
    void Draw() { // Замещающий метод
                  // Circle::Draw() замещает (overrides) Shape::Draw()
        cout << "Circle::Draw()" << endl;
    }
    void Erase() { // Замещающий метод
                   // Circle::Erase() замещает (overrides) Shape::Erase()
        cout << "Circle::Erase()" << endl;
    }
};
```

```
class Square : public Shape {
    double side;
    Point  corner;
public:
    void Draw() { // Замещающий метод
                  // Square::Draw() замещает Shape::Draw()
        cout << "Square::Draw()" << endl;
    }
    void Erase() { // Замещающий метод
                   // Square::Erase() замещает Shape::Erase()
        cout << "Square::Erase()" << endl;
    }
};

class Triangle : public Shape {
    Point anglePoints;
public:
    void Draw() { // Замещающий метод
                  // Triangle::Draw() замещает Shape::Draw()
        cout << "Triangle::Draw()" << endl;
    }
    void Erase() { // Замещающий метод
                   // Triangle::Erase() замещает Shape::Erase()
        cout << "Triangle::Erase()" << endl;
    }
    void Rotate90() { // Обычный метод класса
        cout << "Triangle::Rotate90()" << endl;
    }
};

// Пример использования объектов
void main() {
    Shape *pShape = new Shape();
    pShape->Draw();        // печатается "Shape::Draw()"
    pShape->Erase();       // печатается "Shape::Erase()"

    Circle *pCircle1 = new Circle;
    pCircle1->Draw();      // печатается "Circle::Draw()"
    pCircle1->Erase();     // печатается "Circle::Erase()"

    Shape *pCircle2 = new Circle;
    pCircle2->Draw();      // печатается "Circle::Draw()"
    pCircle2->Erase();     // печатается "Circle::Erase()"
}
```

Что же дает такая унификация?

Во-первых, теперь, работая с объектами разных типов, представляющих абстракции различных геометрических фигур, мы можем использовать методы, которые имеют одинаковый интерфейс, что, в конечном счете, упрощает и реализацию программы, и ее восприятие читателем.

Во-вторых, и это даже более существенно, теперь мы имеем возможность разрабатывать общие процедуры обработки объектов — геометрических фигур, например, мы можем определить функцию для перерисовки графического объекта независимо от его типа примерно следующим образом (листинг 2.7).

Листинг 2.7. Использование общего интерфейса для обработки объектов разных типов

```
// Функция принимает указатель на объект типа Shape:
// по правилам совместимости указателя на производный тип
// с указателем на базовый тип, в качестве аргумента
// может быть передан указатель на объект любого типа,
// производного от Shape (т. е. типа, поддерживающего
// интерфейс, определяемый классом Shape)
void Repaint( Shape *pShape ) {
    pShape->Erase();
    // Возможно, еще какие-то действия...
    pShape->Draw();
}

// Следующий фрагмент кода на C++ иллюстрирует
// примеры использования функции Repaint() для перерисовки
// объектов различных типов
void main() {
    Shape *pCircle = new Circle;
    Repaint( pCircle ); // перерисовка объекта класса Circle:
                        // вызываются Erase() и Draw(),
                        // определенные в классе Circle
    delete pCircle;

    Triangle trng;
    Repaint( &trng ); // перерисовка объекта класса Triangle:
                      // вызываются Erase() и Draw(),
                      // определенные в классе Triangle
}
```

В ходе работы функции `Repaint()` полиморфизм обеспечивает вызов методов `Draw()` и `Erase()` в реализации, специфической для класса `Circle`, а не обоб-

щенную версию из класса `Shape`. Таким образом, полиморфизм помогает учесть действительный тип объекта, а не предполагаемый в функции `Repaint()` тип `Shape`. По существу, мы используем практически ту же модель, что была рассмотрена при анализе отношения реализации: `Shape` представляет интерфейс, а `Circle` обеспечивает реализацию интерфейса. В данной главе механизмы наследования и полиморфизма рассмотрены в той мере, в какой это необходимо для более комфортного изучения последующего материала. *Подробности реализации наследования и полиморфизма в объектно-ориентированных языках рассматриваются в гл. 4, 5, 11.*

2.5. Иерархии классов и модули: модель C++

Напомним, что модульность была определена как представление системы в виде совокупности обособленных сегментов, связь между которыми обеспечивается посредством связей между классами, определяемыми в этих сегментах. При этом выделяют два уровня представления модульности, которые условно называются физическим и логическим уровнями.

Физический уровень имеет дело со структурой файлов и каталогов, в которых хранятся отдельные модули, из которых составлен проект. Так, в строгой интерпретации, в C/C++ физическим модулем является отдельный сегмент компиляции, т. е. объектный файл. Физическому модулю соответствует файл с исходным текстом C/C++, к которому подключены необходимые заголовочные файлы, образующие интерфейс между модулями. Поэтому часто (с некоторой долей условности) модулем также называют файл с исходным текстом. Язык C и первые реализации языка C++ не предоставляли других механизмов реализации модульности, поэтому физический модуль являлся одновременно и логическим модулем. Модульная организация проекта в этом случае тесно связана с моделью обработки программы (рис. 2.14).

В ходе трансляции исходных текстов препроцессор обрабатывает директивы `#include`, обеспечивая помещение в область видимости исходного текста необходимых объявлений, например, объявлений функций, реализуемых в других объектных модулях. В результате компиляции исходного текста получаются объектные модули, содержащие некоторые ссылки в символической форме, или так называемые неразрешенные (unresolved) ссылки. Важным элементом проектирования, обеспечиваемым модульностью, является возможность раздельной трансляции модулей. Модули взаимодействуют друг с другом посредством вызовов функций и чтения-записи глобальных данных. Модуль может предоставлять не только доступ к комплектам функций, но также к определениям типов и констант.

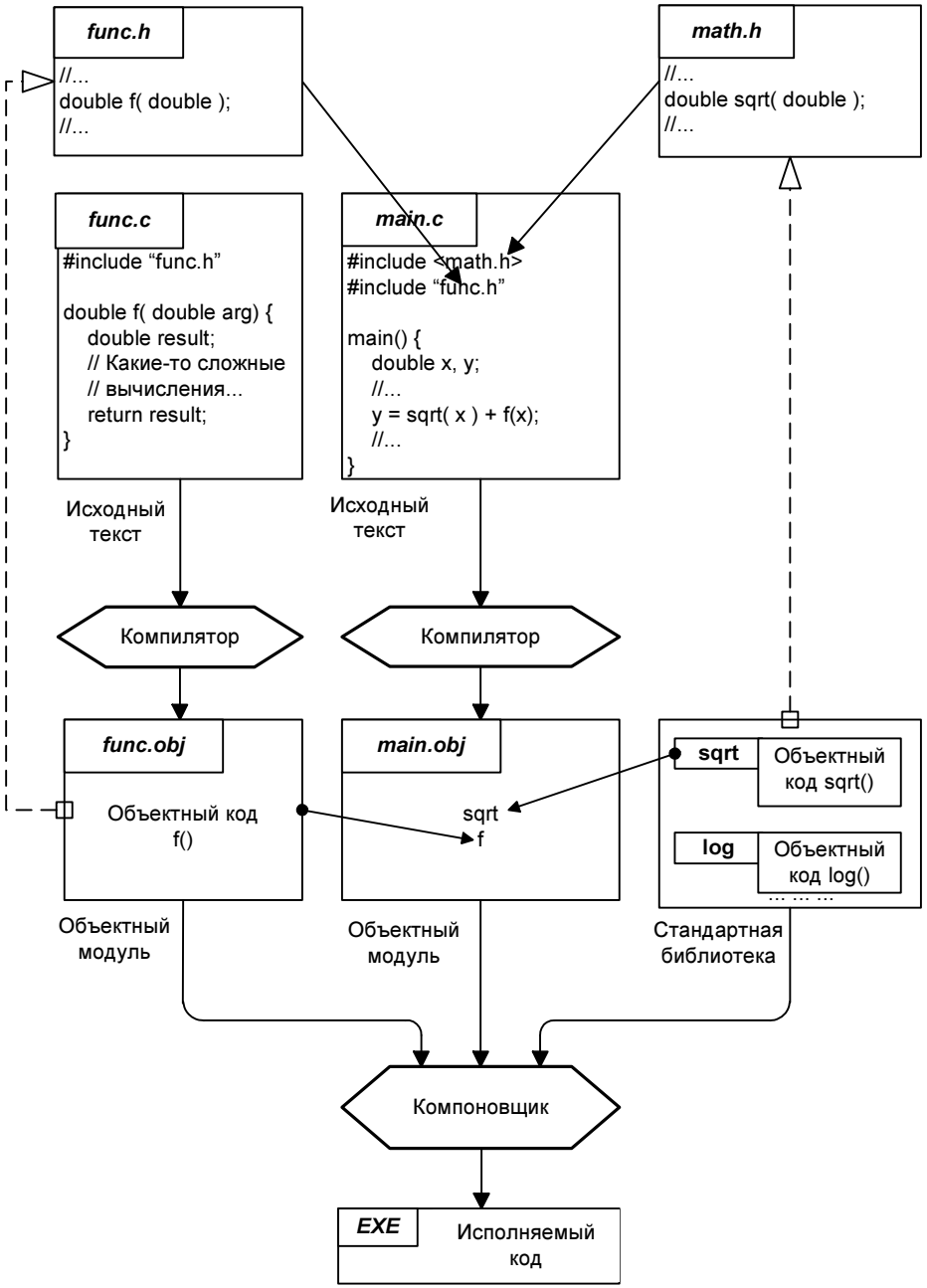


Рис. 2.14. Модульность и обработка программы на языке С

Компоновщик осуществляет связывание объектных модулей, полученных в результате трансляции файлов, и объектных модулей, хранящихся в стандартной библиотеке и в других библиотеках.

Однако работа над проектом, организованным на основе концепции физической модульности, затрудняется с ростом сложности задач и числа программистов, участвующих в разработке. Объектно-ориентированная парадигма с концепцией класса как фрагмента кода, ориентированного на его повторное использование, также предъявляет дополнительные требования к обеспечению модульности средствами языка. Возникает необходимость организации проекта на основе логической взаимосвязи его компонентов, а в связи с этим — обеспечение корректного разрешения возможных конфликтов глобальных имен (переменных, констант, типов, функций), разделение реализации и интерфейсной части определений, содержащихся в модуле, обособление библиотечного и клиентского кода. Это приводит к необходимости реализации механизмов модульности, отражающих логические взаимосвязи между отдельными компонентами. Основной механизм логического группирования программных объектов, поддерживаемый современными языками программирования — это механизм пространств имен. *Реализация пространств имен в C++ рассматривается в гл. 8.* Там же обсуждаются отличия в реализации пространств имен в других языках программирования (на примере языков Java и C#).

В этом разделе ограничимся иллюстрацией возможного варианта модульной организации проекта Garland Application (рис. 2.15).

Данная схема иллюстрирует модульность на логическом уровне. Определения отдельных классов могут физически размещаться в разных файлах. Довольно распространенной в современной практике является модель организации проекта по схеме один физический модуль — один класс, в которой используется следующая стандартная инфраструктура файлов с исходными текстами:

- Заголовочный файл (с расширением h или hpp) содержит определение класса (и, возможно, используемых в этом определении прочих объявлений).
- Файл с расширением cpp содержит реализацию методов класса.

Реализация приложения в соответствии с описанными принципами предлагается читателю для самостоятельного упражнения *(см. упражнение 4 разд. 2.6)*. Читатель может сравнить свое решение с примером, представленным на компакт-диске.

Логическое группирование программных объектов позволяет нам отразить взаимосвязи между программными объектами на более высоком уровне абстракции.

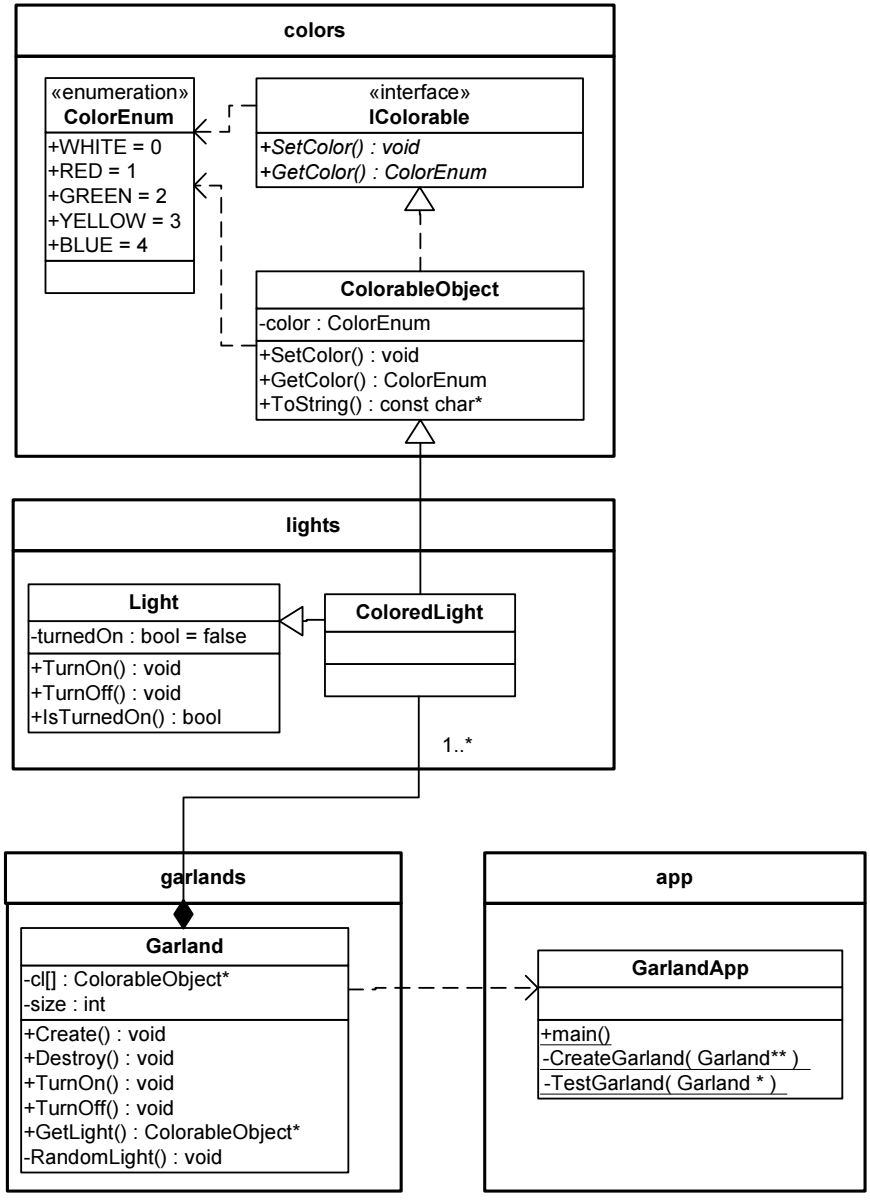


Рис. 2.15. Возможная реализация модульности в проекте Garland Application

На рис. 2.15 выделено 4 логических модуля:

- `colors` — содержит определения типов, интерфейсов и классов для представления абстракций, связанных с цветовыми характеристиками объектов;

- `lights` — содержит определения классов для представления моделей лампочек;
- `garlands` — содержит определения класса для представления гирлянды (если потребуется определить другой класс для представления какой-либо иной модели гирлянды, его также можно поместить в этот модуль);
- `app` — содержит определение класса в связи с приложением.

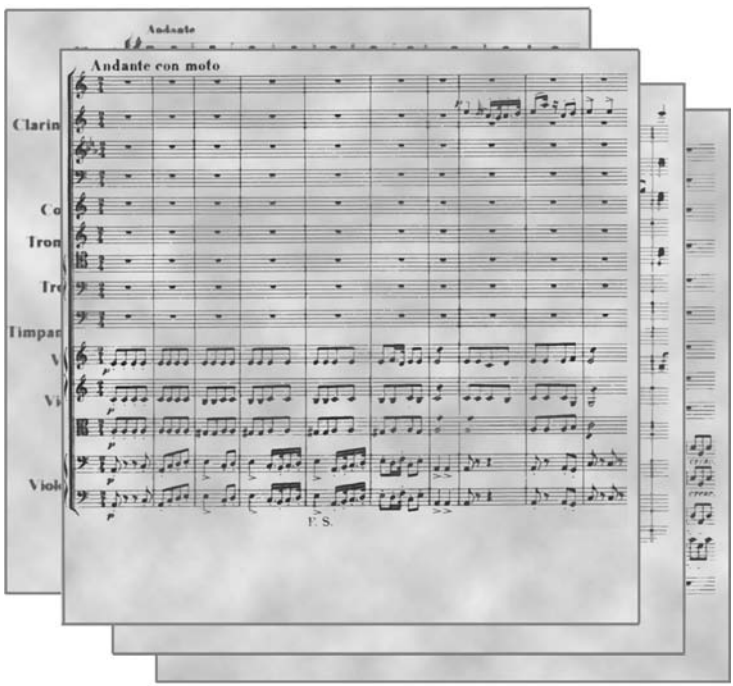
Логические модули могут однозначно соответствовать физическим модулям (как, например, в языке Java) или предоставлять независимый от размещения физических модулей механизм логического группирования программных объектов (как, например, в языке C#).

2.6. Вопросы и упражнения

1. Что такое абстрагирование? Приведите пример абстракции данных.
2. Почему инкапсуляция так важна в процессе объектно-ориентированного программирования? Сформулируйте для себя еще раз, какие основные задачи решает инкапсуляция.
3. В чем неточность следующего утверждения: "инкапсуляция — это процесс интеграции данных и функций для обработки этих данных"?
4. Напишите набросок реализации проекта, основываясь на диаграмме рис. 2.15 (вместо рисования цветных лампочек ограничьтесь печатью текстуальных наименований цвета на консоль). Выясните для себя, детального представления о каких элементах объектно-ориентированного программирования Вам недостает для законченной реализации.
5. Изобразите диаграмму классов, устанавливающую отношения следующих сущностей: вуз, факультет, студент, аспирант, группа студентов, преподаватель, предмет. Какие отношения классов вы использовали? Обсудите с другими студентами различия в полученных диаграммах.
6. Изобразите модульную организацию программы из *упражнения 4 разд. 1.4*. Изобразите набросок модульной организации объектно-ориентированного решения этой задачи.

ЧАСТЬ II

Конструирование типов



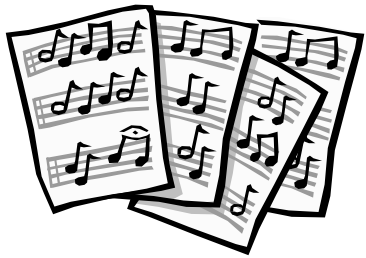
Глава 3. Класс как основной механизм абстракции

Глава 4. Наследование и иерархии классов

Глава 5. Объекты классов и полиморфизм

Глава 6. Обработка ошибок на основе использования механизма исключений

ГЛАВА 3



Класс как основной механизм абстракции

Основная задача классов — предоставить программисту инструмент для создания новых типов с тем, чтобы их можно было без ограничений использовать в программе наряду со встроенными типами.

По замечанию Страуструпа, тип является конкретным представлением некоторой концепции. Например, встроенный тип `int` вместе с множеством операций, применимых к данным этого типа (в частности, сложение, вычитание, умножение, деление, получение остатка от деления), представляет собой конкретное воплощение математической абстракции целого числа. Это конкретное воплощение определяется рядом характеристик, в том числе:

- областью применения соответствующего типа (для типа `int` в языке C++ такой областью являются точные вычисления);
- способом представления данных этого типа в памяти (для типа `int` в языке C++ — дополнительный двоичный код, длина которого зависит от архитектуры — 4 байта для 32-разрядных архитектур);
- множеством операций, разрешенных для данных этого типа;
- множеством совместимых типов и т. д.

Новые типы создаются для воплощения концепций, которые не выражаются непосредственно (или адекватно) встроенными типами.

3.1. Структура и организация определения класса

Общие сведения об организации определения класса и использовании объектов класса *представлены в гл. 2*. В данном разделе устройство класса излага-

ется более детально, а также формируется основа для изучения механизмов использования классов с целью создания новых конкретных и абстрактных типов.

Элементы-данные и элементы-действия

В предварительном обсуждении элементов объектной модели мы отмечали, что класс — это, в первую очередь, инструмент абстракции. Определяя класс, мы создаем программную сущность, позволяющую выделить существенные характеристики некоторого объекта, отличающие его от всех других видов объектов. При этом мы отделяем друг от друга элементы объекта, определяющие его устройство (элементы-данные), от элементов, определяющих его поведение (элементы-действия). Данные, объявленные внутри определения класса, могут, в частности, быть переменными-членами класса и константами-членами класса. Функции, объявленные внутри определения класса, называются функциями-членами класса, или методами класса.

Определение класса

Рассмотрим в качестве простого примера определение типа для работы со стандартной очередью, содержащей в качестве элементов целые числа (листинг 3.1). Под стандартной очередью здесь понимается способ организации объектов, при котором раньше обрабатывается (извлекается из очереди) тот объект, который был раньше занесен в очередь, иначе такой способ управления данными называется очередью типа FIFO: First Input — First Output (обычная очередь в каком-нибудь магазине вполне соответствует этому определению).

Листинг 3.1. Определение класса `QueueInt` (версия 1)

```
/*
 * QueueInt.h
 *
 * Определение класса для работы с очередью
 * Версия 1
 *
 */
#ifndef _QueueInt_h_
#define _QueueInt_h_

class QueueInt {      // Стандартная очередь из целых чисел:
                      // "первым пришел - первым обслужен"
```

```
public:
    //
    // Данные, необходимые для работы с очередью
    //

    enum {
        EMPTY = -1,   // Признак пустоты очереди
        MAXSIZE = 100 // Предельный размер очереди
    };

    int body[ MAXSIZE ];     // Массив для хранения данных
    int size;                // Предельный размер
                             // инициализированной очереди
    int length;              // Текущая длина очереди

    int last;                // Индекс последнего занесенного
                             // элемента очереди
    int next;                // Индекс извлекаемого
                             // элемента очереди

    //
    // Функции для работы с очередью
    //

    void Init( int customSize = MAXSIZE );
                             // Инициализация очереди

    void Enqueue( int );     // Занесение элемента
    int  Dequeue();          // Извлечение элемента

    void Print();            // Печать текущего содержимого
};

#endif
```

Главным недостатком данного определения является общедоступность всех элементов класса и, следовательно, открытый доступ к механизмам представления очереди на уровне реализации. С одной стороны, это оставляет простор для ошибок при работе с объектами класса (например, некорректная модификация элементов массива `body` или значения переменной `length`), с другой стороны, усложняет использование данного класса.

Управление доступом к членам класса

Разграничение доступа к членам класса преследует две основные цели:

- обеспечение связывания функций-членов класса с переменными-членами класса таким образом, чтобы только эти функции непосредственно зависе-

ли от представления класса и имели непосредственный доступ к объектам класса;

- обеспечение инкапсуляции, т. е. отделения внешнего интерфейса класса от его реализации.

Члены класса, объявленные со спецификатором `private` (закрытые члены класса), могут использоваться только методами класса. Как уже было показано выше, эту защиту, разумеется, можно обойти манипулированием с адресами или путем явного преобразования типа, однако следует понимать, что основная задача защиты, обеспечиваемой закрытостью данных, заключается в предотвращении случайных ошибок, а не преднамеренного нарушения правил доступа.

К членам класса, объявленным со спецификатором `public` (к открытым членам класса), разрешен доступ не только функциям-членам класса, но и обращение из-за пределов объявления класса. Поэтому такие члены класса образуют открытый интерфейс класса.

Класс *QueueInt*: реализация инкапсуляции

Исправленное определение класса для работы с очередью, содержащей целые числа, может выглядеть так, как представлено в листинге 3.2.

Листинг 3.2. Определение класса `QueueInt` (версия 2)

```
/*
 * QueueInt.h
 *
 * Определение класса для работы с очередью
 * Версия 2
 *
 */

#ifndef _QueueInt_h_
#define _QueueInt_h_

class QueueInt {      // Стандартная очередь из целых чисел:
                      // "первым пришел - первым обслужен"
private:
    //
    // Закрытые члены класса
    //(доступ только в пределах класса)
    //
```

```
    enum {
        EMPTY = -1,   // Признак пустоты очереди
        MAXSIZE = 100 // Предельный размер очереди
    };

    int body[ MAXSIZE ];    // Массив для хранения данных
    int size;               // Предельный размер
                            // инициализированной очереди
    int length;             // Текущая длина очереди

    int last;               // Индекс последнего занесенного
                            // элемента очереди
    int next;               // Индекс извлекаемого
                            // элемента очереди
public:
    //
    // Открытые члены класса
    //

    void Init( int customSize = MAXSIZE );
                            // Инициализация очереди

    void Enqueue( int );    // Занесение элемента
    int  Dequeue();         // Извлечение элемента

    void Print();           // Печать текущего содержимого
};

#endif
```

Теперь определение класса вполне корректно соответствует различиям между назначением абстрагирования и задачами, решаемыми инкапсуляцией. С точки зрения внешнего интерфейса очередь представляется в глазах пользователя примерно так, как это проиллюстрировано в верхней части рис. 3.1. Нижняя часть рисунка показывает модель очереди времени разработки (которая включает в себя целый набор элементов-членов класса, недоступных извне).

Если для пользователя класса работа с очередью — это, по существу, последовательность операций `Enqueue()` и `Dequeue()` с очередью заданного размера, то разработчик имеет дело с множеством элементов, обеспечивающих простоту и наглядность внешнего интерфейса. Так, реализация метода `Enqueue()` предполагает изменение индекса `last` и занесение по адресу `body+last` задаваемого элемента. Реализация метода `Dequeue()` заключается в чтении эле-

мента массива `body` с индексом `next` и модификацию индекса `next`. Текущее состояние очереди характеризуется числом элементов, хранящихся в очереди, — длиной очереди `length`.

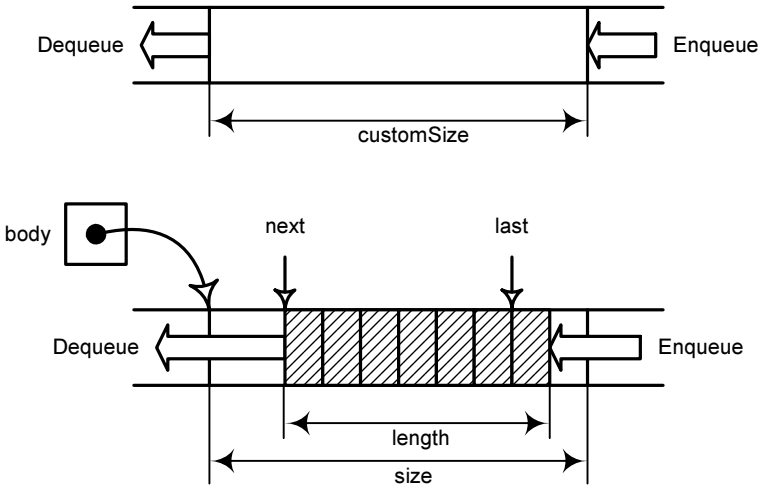


Рис. 3.1. Различия абстрактных моделей пользователя и разработчика

Порядок объявления открытых и закрытых членов класса

C++ не регламентирует порядок следования членов класса в определении. Допускается введение нескольких обособленных секций с одинаковыми правами доступа *(см. пример в листинге 2.4)*.

Необходимость объявления доступа для группы членов класса отчасти вынуждает программиста группировать объявления членов класса с одинаковым доступом, даже если это по каким-то причинам не отвечает представлениям программиста о "правильном" порядке объявлений. Видимо, из-за этого разработчики языков Java и C# решили, что спецификатор доступа следует указывать для каждого имени отдельно. Это дает возможность упорядочивать объявления членов класса по другим признакам (например, в C# таким признаком может быть атрибут члена класса). Однако большинство программистов придерживаются традиции помещать рядом объявления членов классов с одинаковым доступом.

При разработке определения класса необходимо также иметь в виду, что многие программисты следуют стилю объявления, в соответствии с которым сначала объявляются открытые члены класса, а затем — члены класса с ограниченным или закрытым доступом. Это связано с тем обстоятельством, что

объявление новых типов обычно предполагает их повторное использование. Это означает, что требуется обеспечить удобство восприятия кода, а следовательно — в первую очередь представить открытый интерфейс класса.

В рамках учебных проектов мы будем разворачивать определение класса в соответствии с обсуждаемым учебным материалом. В большинстве случаев это будет означать помещение закрытых данных в начале спецификации класса (так удобнее при обсуждении учебных примеров).

Многие объектно-ориентированные языки поддерживают еще один вид доступа — защищенный, позволяющий определить члены класса, к которым возможен доступ из производных классов. При этом защищенные члены класса закрыты для всех объектов. Защищенный доступ устанавливается спецификатором `protected`. *Подробно защищенный доступ рассматривается в гл. 4.*

Важное замечание об инкапсуляции

Мейерс справедливо указывает на то, что инкапсуляция не обязательно предполагает реализацию всех действий над объектом посредством функций-членов класса [Meyers, 2000]. Действительно, можно разделить public-методы класса на два подмножества: методы, требующие доступа к внутренней реализации, и методы, использующие только другие открытые методы (рис. 3.2).

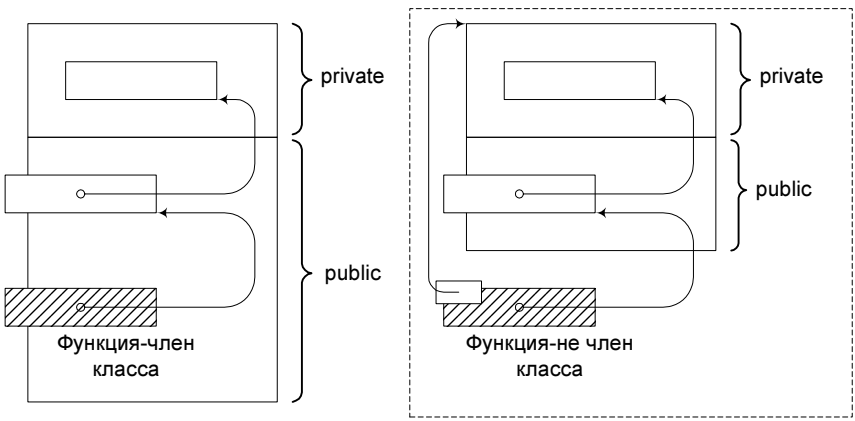


Рис. 3.2. Функции-не члены класса, улучшающие инкапсуляцию

Эти "вторые" методы не требуют доступа к представлению класса, а значит — могут быть реализованы в виде обычных функций, не являющихся членами класса (non-member functions), получающих объект (или ссылку на объект) в качестве явного аргумента. В тех случаях, когда такие функции зависят также и от каких-либо других внешних аргументов, их реализация за

пределами класса оказывается не просто разумной альтернативой, но и в определенном смысле улучшает инкапсуляцию. Ведь изменение этих функций никак не затрагивает определение класса. Для того чтобы указать на связь этих функций с классом, объекты которого они обрабатывают, можно воспользоваться помещением их в одно общее пространство имен *(см. гл. 8)*.

3.2. Инициализация объектов. Конструкторы

Важный вопрос, немедленно возникающий при определении экземпляров класса (объектов), заключается в том, как инициализировать объект. Разумеется, для этого можно использовать специальную функцию (что и предполагает рассмотренное нами объявление класса `QueueInt`), например:

```
void QueueInt::Init( int customSize ) {

    if( customSize < 1 || customSize > MAXSIZE ) {
        cout << "Incorrect size. ";
        cout << "Using default value.\n";

       size = MAXSIZE;
    }
    else size = customSize;

    last = EMPTY;
    next = 0;
    length = 0;
}
```

Этот подход работает, но не только лишен элегантности, но и не вполне надежен. Нигде в объявлении типа не указано, что объект должен быть проинициализирован. Кроме того, некоторые объекты могут не нуждаться в инициализации, а для некоторых инициализация должна быть обязательной. Все это является источником возможных ошибок: программист может или забыть проинициализировать объект, или проинициализировать его более одного раза.

Конструктор как инструмент гарантированной инициализации

Для решения обозначенной проблемы инициализации объектов в объектно-ориентированном программировании реализован подход, предоставляющий программисту возможность явно определить функцию, основное назначение которой и состоит в инициализации объекта. Такая функция называется конструктором. Особая роль конструктора дополнительно подчеркивается тем,

что во многих языках (в том числе и в C++) имя конструктора совпадает с именем класса.

```
QueueInt::QueueInt( int customSize ) {

    if( customSize < 1 || customSize > MAXSIZE ) {
        cout << "Incorrect size. ";
        cout << "Using default value.\n";

        size = MAXSIZE;
    }
    else size = customSize;

    last = EMPTY;
    next = 0;

    length = 0;
}
```

В отличие от других функций, конструктор *никогда* не имеет возвращаемого значения.

Примечание

Последняя фраза, вероятно, нуждается в небольшом комментарии. Многие студенты, начинающие изучать язык C++, ошибочно полагают, что отсутствие возвращаемого значения конструктора эквивалентно определению функции, возвращающей значение типа `void` (т. е. не возвращающей никакого значения). Однако здесь есть существенное семантическое различие. Функция, объявляемая как `void`, не возвращает значения, хотя в принципе могла бы быть определена и иным образом (например, функция `main()` часто определяется и как `void main()`, и как `int main()` — во втором случае наличие инструкции `return` в теле функции обязательно). Конструкторы в принципе не могут иметь возвращаемого значения.

Определение класса `QueueInt` с конструктором имеет следующий вид (листинг 3.3).

Листинг 3.3. Определение класса `QueueInt` (версия 3)

```
/*
 * QueueInt.h
 *
 * Определение класса для работы с очередью
 * Версия 3
 *
 */
```

```
#ifndef _QueueInt_h_
#define _QueueInt_h_

class QueueInt {     // Стандартная очередь из целых чисел:
                     // "первым пришел - первым обслужен"
private:
    //
    // Закрытые члены класса
    //(доступ только в пределах класса)
    //

    enum {
        EMPTY = -1,   // Признак пустоты очереди
        MAXSIZE = 100 // Предельный размер очереди
    };

    int body[ MAXSIZE ];     // Массив для хранения данных
    int size;                // Предельный размер
                             // инициализированной очереди
    int length;              // Текущая длина очереди

    int last;                // Индекс последнего занесенного
                             // элемента очереди
    int next;                // Индекс извлекаемого
                             // элемента очереди
public:
    //
    // Открытые члены класса
    //

    QueueInt ( int customSize = MAXSIZE );
                             // Конструктор:
                             // обеспечивает гарантированную
                             // однократную инициализацию

    void Enqueue( int );     // Занесение элемента
    int  Dequeue();          // Извлечение элемента
    void Print();            // Печать текущего содержимого
};

#endif
```

В отличие от функции `Init()`, вызов конструктора гарантирован: конструктор вызывается всегда при попадании объекта в область видимости (в соответствии с концепцией автоматического распределения памяти) или при размещении объекта в динамической памяти посредством операции `new`, например:

```
#include "QueueInt.h" // (в версии 3)

void SomeFunction() {

    // Автоматическое размещение объекта в стеке
    // => Вызов конструктора при попадании объектной
    //    переменной в область видимости
    QueueInt queue;     // Аргумент по умолчанию (MAXSIZE=100)
    queue.Enqueue(100); // Занесение элемента в очередь
    // ...

    // Объявляется указатель на объект
    // => Нет вызова конструктора!
    QueueInt *dynamicQueue;

    // Резервируется динамическая память
    // => Явный вызов конструктора с аргументом 10
    dynamicQueue = new QueueInt( 10 );
    dynamicQueue->Enqueue(200); // Занесение элемента в очередь
    // ...
}
```

3.3. Спецификация и реализация класса

> Выявление различий между спецификацией и реализацией — постоянная тема для дискуссий. Никто не может спутать настоящий мост с нарисованным или автомобиль с его чертежом. Но в случае программного обеспечения мы постоянно рискуем спутать трубу с ее изображением.
>
> *Бертран Мейер*

Конструкция, подобная той, что приведена в листинге 3.3, называется *определением* класса, потому что она определяет новый тип. По историческим причинам, определение класса часто называют *объявлением* класса. Также как и другие объявления, не являющиеся определениями, определение класса, помещенное в заголовочный файл C++, может быть повторено в другом исходном файле при помощи директивы препроцессора `#include`.

С точки зрения организации и сопровождения кода, содержащего определения классов, удобно различать понятия *спецификации класса* и *реализации класса*.

Спецификация класса

Спецификация класса содержит информацию, достаточную для корректного использования класса. Спецификация для разработчика класса должна содержать объявления всех членов класса. Спецификация для пользователя должна содержать, по крайней мере, объявления открытых и защищенных членов класса. Для корректной трансляции кода компилятор должен иметь информацию о типах закрытых членов класса, хотя они не обязательны в спецификации, ориентированной на пользователя класса.

Рассмотрим в качестве примера спецификацию класса для работы с очередью, определение которого представлено в листинге 3.3.

Предварительная спецификация класса для пользователя могла бы выглядеть следующим образом:

```
class QueueInt

Обеспечивает функциональность для хранения целых чисел
в стандартной очереди, обрабатываемой по принципу:
"первым пришел – первым обслужен".

Определен в заголовочном файле QueueInt.h

Конструктор (является конструктором по умолчанию):
    QueueInt( int customSize = 100 );
        Инициализирует очередь указанного размера.
        Если аргумент имеет некорректное значение,
        печатается диагностическое сообщение
        и устанавливается значение по умолчанию.

        Аргументы конструктора:
        customSize – задаваемый предельный размер очереди,
                     Значение по умолчанию: 100
                     Предельное разрешенное значение: 100

Открытые функции-члены класса:

    void Enqueue( int item );
        Обеспечивает занесение элемента в очередь.
        Если очередь полна, печатает сообщение об ошибке.

        Аргументы:
            item – элемент, заносимый в очередь

        Возвращаемое значение:
            Нет
```

```
    int Dequeue();
        Обеспечивает извлечение элемента из очереди.
        Если очередь пуста, печатает сообщение об ошибке.

        Аргументы:
            Нет

        Возвращаемое значение:
            Значение извлекаемого элемента


    void Print();
        Обеспечивает печать на консоль содержимого очереди.
        Если очередь пуста, печатает диагностическое сообщение.

    Аргументы:
        Нет

    Возвращаемое значение:
        Нет

Конец спецификации класса QueueInt
----------------------------------
```

Понятно, что при условии грамотного подробного документирования определения класса, помещаемого в заголовочный файл, само определение класса вполне корректно считать вариантом спецификации класса. Причем, если подробно документируется назначение открытых (public) и защищенных (protected) членов класса, то речь идет о пользовательской спецификации (как правило, пользователю хорошо спроектированного класса нет необходимости изучать реализацию класса). Если подробно документированы все члены класса, включая закрытые (private), то перед нами — спецификация времени разработки. Неудивительно, что в таких языках, как Java и C#, предусмотрены встроенные средства генерирования спецификаций в виде так называемых документирующих комментариев *(подробнее о встроенной документации см. в разд. 11.7)*.

Реализация класса

Реализация класса состоит из определения класса, реализации методов класса и определений статических данных класса *(о статических членах класса см. разд. 3.6 данной главы)*.

Текст реализации всех методов класса `QueueInt` представлен в листинге 3.4.

Листинг 3.4. Реализация методов класса `QueueInt`

```
/*
 * QueueInt.cpp
 * QUEUE of INTegers: implementation
 *
 * Реализация методов класса QueueInt
 *
 */

#include <iostream.h>

#include "QueueInt.h" // (в версии 3)

// Конструктор
QueueInt::QueueInt( int customSize ) {

    if( customSize < 1 || customSize > MAXSIZE ) {

        cout << "Incorrect size. "
        cout << "Using default value.\n";

        size = MAXSIZE;
    }
    else size = customSize;

    last = EMPTY;
    next = 0;

    length = 0;
}

// Добавление элемента в очередь
void QueueInt::Enqueue( int item ) {

    if( length == size ) {
        cout << "Enqueue error. ";
        cout << "Queue is full." << endl;

        return;
    }

    last = ( last+1 ) % size;
    length++;

    body[ last ] = item;
}
```

```
// Извлечение элемента из очереди
int QueueInt::Dequeue() {

    if( length == 0 ) {
        cout << "Dequeue error. ";
        cout << "Queue is empty." << endl;

        return EMPTY;
    }


    int copy = body[ next ];

    next = ( next+1 ) % size;
    length--;

    return copy;
}

// Печать содержимого очереди
void QueueInt::Print() {

    if( length == 0 ) {
        cout << "Queue is empty." << endl;
        return;
    }

    int count;
    int ix;
    for( ix = next % size, count=1;
         count<=length;
         ix = ( ix+1) % size, count++ ) {

        cout << body[ ix ] << " ";
    }

    cout << endl;
}
```

В основе своей данный код не содержит трудных для понимания элементов. Некоторых пояснений, вероятно, требует лишь работа с индексами элементов массива `body` (рис. 3.3).

Если снова обратить внимание на рис. 3.1, то можно заметить, что при добавлении элементов в очередь рано или поздно мы достигнем конца области памяти, отведенной для массива `body`. В этом случае можно продолжить занесение элементов в очередь, помещая их в начало массива `body` (если, разумеется, текущее значение длины очереди `length` не достигло значения `size`).

Наиболее естественным выглядит решение применить элементы арифметических вычислений по модулю для модификации `last` [Graham, Knuth, Patashnik, 1994, разд. 3.4, 4.6]. По своему содержанию выражение для модификации индекса `last`, использованное в методе `Enqueue()`, эквивалентно следующей последовательности действий:

```
last = last + 1;
if( last >= size ) last = 0;
```

Эта запись содержит 4 операции и инструкцию `if`, в то время как использованная в реализации запись содержит 3 операции (и, по-моему, выглядит более элегантно). Аналогично обрабатывается изменение индекса `next`.

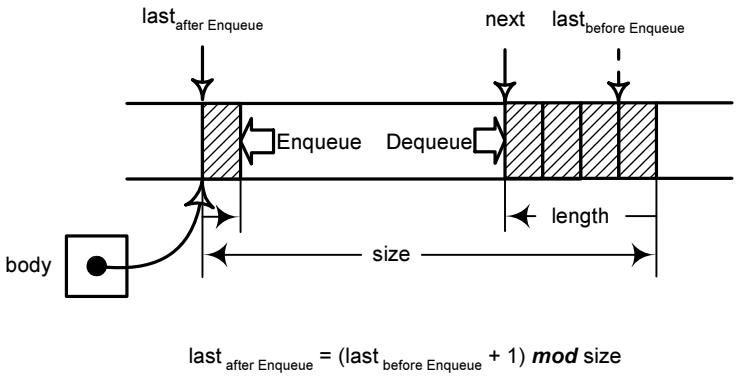


Рис. 3.3. Обработка индексов при работе с массивом `QueueInt::body`

Эта реализация вполне работоспособна, однако не свободна от ряда недостатков. Ниже перечислены наиболее существенные из них:

1. В ходе работы методов класса `QueueInt` происходит печать на консоль. Это ограничивает область применения класса только консольными приложениями.
2. Методы класса реализованы таким образом, что в случае возникновения ошибок производится печать информации об ошибке на консоль, что может не вписываться в организацию клиентского кода.
3. Встроенная обработка ошибок работы с очередью не позволяет пользователю класса реализовать свой обработчик ошибок с учетом инфраструктуры клиентского приложения.
4. В случае, если очередь пуста, функция `Dequeue()` возвращает фиктивное значение, которое вполне может совпадать с одним из элементов очереди. Это может быть источником ошибок при использовании очереди.

5. Спроектированный класс позволяет хранить в очереди только целые данные. Концептуально сам способ организации данных в виде очереди инвариантен к типу хранимых данных. Неплохо бы учесть это и разработать класс, реализующий модель очереди, легко адаптируемой к данным разных типов.

В дальнейших главах мы рассмотрим основные подходы к устранению этих недостатков.

3.4. Альтернативные способы инициализации объектов. Перегрузка конструкторов

Так же как и любые другие функции языка C++, конструктор можно перегружать. Поэтому класс может иметь несколько конструкторов с различными списками параметров. Это позволяет иметь несколько вариантов инициализации объекта. Рассмотрим пример фрагмента реализации класса для работы со строками (листинг 3.5).

Листинг 3.5. Фрагмент реализации класса `MString`

```
/*
 * MString.h
 * Model STRING
 *
 * Определение класса для работы со строками
 * Версия 1
 *
 */

#ifndef _MString_h_
#define _MString_h_

#include <cstring>
#include <cassert>

#include <iostream>
using namespace std;

const int DEFAULT_LEN = 128; // Длина строки по умолчанию

class MString {

    int size;      // Размер памяти, отводимой для хранения строки
    int length;    // Действительная длина строки
    char *value;   // ASCIIZ-строка
```

```
public:
    // Конструктор #1:
    //   Инициализация пустой строки заданием длины
    MString( int sz = DEFAULT_LEN ) {

        size = sz+1;
        value = new char[ size ];
        assert( value != 0 );

        length = 0;

        value[ 0 ] = '\0';
    }

    // Конструктор #2:
    //   Инициализация строки значением существующей
    //   ASCIIZ-строки
    MString( const char *pStr ) {

        length = strlen( pStr );
        size = length + 1;

        value = new char[ size ];
        assert( value != 0 );

        strcpy( value, pStr );
    }

    int GetSize() {
        return size;
    }

    int GetLength() {
        return length;
    }

    void PrintLn() {
        cout << "\"" << value << "\"" << endl;
    }
    // ...
};

#endif
```

Конструктор № 1 обеспечивает инициализацию строкового объекта определением длины строки, которую этот объект сможет хранить, и инициализацию пустой строкой. Конструктор № 2 позволяет создать строковый объект

на основе существующей строки. Таким образом, в распоряжении программиста есть два способа инициализации объекта класса `MString`. Первый способ предполагает использование конструктора № 1:

```
MString str1( 10 ); // Создается объект, содержащий пустую строку и
                    // способный вмещать строку длиной до 10 символов
```

Второй способ предполагает использование конструктора № 2:

```
MString str2( "abc" ); // Создается объект, содержащий строку "abc" и
                       // способный вмещать строку длиной до 3 символов
```

Явным недостатком рассмотренного примера является его зависимость от глобальной константы. Устранение этого недостатка достигается использованием статических членов класса, которые мы рассмотрим в *разд. 3.6 данной главы*.

Понятие о конструкторе по умолчанию

Конструктор, допускающий вызов без аргументов, называется конструктором по умолчанию. В рассмотренном выше примере класса `MString` первый из конструкторов является конструктором по умолчанию. Следующий фрагмент иллюстрирует создание объекта с использованием конструктора по умолчанию:

```
MString str3; // Создается объект, содержащий пустую строку и
              // способный вмещать строку длиной до 128 символов
              // (см. определение конструктора по умолчанию)
```

Если программист объявил несколько конструкторов, но среди них нет ни одного конструктора по умолчанию, транслятор генерирует сообщение об ошибке при попытке создания объекта без аргументов, например:

```
class X {
    int b;
public:
    // Конструктор есть, но он не является конструктором
    // по умолчанию
    X( int a ) {
        b = a;
    }
};

void SomeFunction() {

    X obj1( 1 ); // ПРАВИЛЬНО: вызывается конструктор X::X( 1 )
```

```
    X obj2;      // ОШИБКА: нет конструктора по умолчанию
    X ar[ 10 ];  // ОШИБКА: нет конструктора по умолчанию

}
```

Примечание

Заметим, что само по себе отсутствие конструктора по умолчанию ошибкой не является (ошибка генерируется только при попытке размещения объекта). Если явное задание конструктора отсутствует, компилятор попытается при необходимости сгенерировать конструктор по умолчанию, если это возможно.

Права доступа в связи с конструкторами

Как и любой другой член класса, конструктор может быть объявлен с разным спецификатором доступа. Помещение конструктора в защищенную секцию делает возможным обращение к нему только в инициализаторе производного класса *(см. гл. 4)*. Объявление закрытого конструктора запрещает инициализацию объекта определенным образом. В частности, довольно часто таким образом запрещают инициализацию копированием *(см. разд. 3.9)*.

Наконец, если класс не имеет ни одного открытого конструктора, объекты такого класса создавать нельзя. Это означает, что данный класс можно использовать только в качестве базового для других классов. В этом плане он подобен абстрактному классу *(см. разд. 5.2)*.

Конструкторы как неявные преобразования

По определению, конструктор, имеющий один аргумент, задает неявное преобразование типа, т. е. определяет правило преобразования типа аргумента конструктора в тип объекта. Например, конструкторы класса `MString` в листинге 3.5 задают неявные преобразования, поэтому наряду с определениями объектов

```
MString str4( "abc" );
MString str5( 10 );
```

можно писать:

```
MString str4 = "abc"; // Конструктор выполняет роль
                      // неявного преобразования
                      // из const char* в MString
MString str5 = 10;    // Конструктор выполняет роль
                      // неявного преобразования
                      // из int в MString
```

Семантически такая форма записи эквивалентна развернутой форме:

```
MString str4 = MString( "abc" );
MString str5 = MString( 10 );
```

Таким образом, в случае, если конструктор имеет один аргумент, он используется, когда ожидается значение объекта, и когда такое значение может быть создано конструктором из значения инициализатора. Во многих случаях такая возможность представляется довольно удобной. Одним из примеров может служить приведенная выше инициализация объекта класса `MString` значением строкового литерала. В качестве второго примера рассмотрим набросок определения класса, воплощающего концепцию комплексной величины (листинг 3.6).

Листинг 3.6. Конструктор с одним аргументом как неявное преобразование

```
class MyComplex {
    double re;
    double im;
public:
    MyComplex() {
        re = 0.0;
        im = 0.0;
    }
    MyComplex( double real, double image ) {
        re = real;
        im = image;
    }
    MyComplex( double real ) {
        re = real;
        im = 0.0;
    }
    void PrintLn() {
        cout << re << "+" << im << "i" << endl;
    }
    //...
};
```

Наличие первого конструктора позволяет инициализировать действительную и мнимую части нулями:

```
MyComplex z1; // z1 = { 0, 0 }
```

Второй конструктор реализует общий случай инициализации действительной и мнимой частей произвольными значениями:

```
MyComplex z2( 5, -1 ); // z2 = { 5, -1 }
```

Наконец, наличие третьего конструктора позволяет инициализировать комплексную переменную скалярным значением (ожидаемым эффектом такой инициализации должно быть заполнение действительной части задаваемым скалярным значением, а мнимой части — нулем):

```
MyComplex z3 = 5; // z3={ 5, 0 }
```

Последнее объявление эквивалентно следующей записи:

```
MyComplex z3( 5, 0 );
```

Заметим, что, воспользовавшись заданием аргументов по умолчанию, определение трех конструкторов класса `MyComplex` можно свести к одному:

```
class MyComplex {
    double re;
    double im;
public:
    MyComplex( double real = 0.0, double image = 0.0 ) {
        re = real;
        im = image;
    }
    //...
};
```

Если неявное преобразование типа, задаваемое конструктором, нежелательно, то его можно запретить, объявив соответствующий конструктор со спецификатором `explicit`. Рассмотрите в качестве примера модификацию класса `MString` (листинг 3.7).

Листинг 3.7. Конструктор, объявленный со спецификатором `explicit`

```
class MString {

    int size;      // Размер памяти, отводимой для хранения строки
    int length;    // Действительная длина строки
    char *value;   // ASCIIZ-строка

public:
    // Конструктор #1:
    //   Инициализация пустой строки заданием длины
    explicit MString( int sz = DEFAULT_LEN );
        // Спецификатор explicit запрещает неявное преобразование
    // Конструктор #2:
    //   Инициализация строки значением существующей
    //   ASCIIZ-строки
```

```
    MString( const char *pStr );
        // Неявное преобразование разрешено
    // ...
};
```

Инициализация объекта значением строкового литерала возможна:

```
MString s1 = "abc"; // ПРАВИЛЬНО
```

Попытка инициализации объекта неявным преобразованием из целого типа к `MString` приводит к ошибке компиляции:

```
MString s2 = 10; // ОШИБКА
```

Правильными вариантами являются теперь только явные формы:

```
MString s2( 10 ); // ПРАВИЛЬНО
MString s2 = MString( 10 ); // Тоже ПРАВИЛЬНО
```

3.5. Разрушение среды функционирования объектов. Деструкторы

Конструкторы отвечают за инициализацию объектов. Иногда создание объекта предполагает помимо прочего захват каких-либо ресурсов (например, резервирование динамической памяти).

Это означает, что для таких классов необходимо наличие функции, которая будет ГАРАНТИРОВАННО (как и в случае с конструктором) вызываться по окончании использования объекта. Окончание использования объекта может быть связано с выходом его из области видимости или с освобождением динамической памяти, занятой объектом, например:

```
void UsingMStrings() {

    //...
    {
        // Начало некоторого блока
        MString mStr1( "string1" ); // Инициализация объекта:
                                    // вызывается конструктор ==>
                                    // объект размещается в стеке

        // ... Какие-то действия с объектом mStr1

    } // Конец блока: объект mStr1 вышел из области видимости
      // НЕОБХОДИМО БЫЛО В КОНЦЕ БЛОКА ОСВОБОДИТЬ ЗАНЯТУЮ
      // ДИНАМИЧЕСКУЮ ПАМЯТЬ

    //...
```

```
   MString *mStr2 = new MString( "string2" );
                        // Объект размещен в динамической памяти:
                        // вызывается конструктор ==>
                        // резервируется динамическая
                        // память под value

    // ... Какие-то действия с объектом mStr2

    delete mStr2; // Освободили память, занятую объектом mStr2
                  // НЕОБХОДИМО БЫЛО ДО ОСВОБОЖДЕНИЯ ПАМЯТИ, ЗАНЯТОЙ
                  // ОБЪЕКТОМ, ОСВОБОДИТЬ ПАМЯТЬ, СВЯЗАННУЮ С value
}
```

Корректное разрушение среды, в которой функционирует объект, обеспечивается специальной функцией-членом класса, которая называется деструктором. Класс может иметь не более одного деструктора.

В подавляющем большинстве случаев (наиболее показательные из них иллюстрировал только что приведенный пример) деструкторы вызываются неявно. Лишь в редких случаях программисту необходимо явно вызывать деструктор (подробности см. в [Stroustrup, 2000, разд. 10.4.11]).

В C++ имя деструктора образуется присоединением знака "тильда" к имени класса.

Версия класса `MString`, дополненная деструктором, имеет следующий вид (листинг 3.8).

Листинг 3.8. Дополненный деструктором фрагмент определения класса `MString`

```
/*
 * MString.h
 *
 * Определение класса для работы со строками
 * Версия 1, дополненная деструктором
 *
 */

#ifndef _MString_h_
#define _MString_h_

#include <cstring>
#include <cassert>

const int DEFAULT_LEN = 128; // Длина строки по умолчанию
```

```
class MString {

    int size;      // Размер памяти, отводимой для хранения строки
    int length;    // Действительная длина строки
    char *value;   // ASCIIZ-строка

public:
    // Конструктор #1:
    //   Инициализация пустой строки заданием длины
    explicit MString( int sz = DEFAULT_LEN ) {

        size = sz+1;
        value = new char[ size ];
        assert( value != 0 );

        length = 0;

        value[ 0 ] = '\0';
    }

    // Конструктор #2:
    //   Инициализация строки значением существующей
    //   ASCIIZ-строки
    MString( const char *pStr ) {

        length = strlen( pStr );
        size = length + 1;

        value = new char[ size ];
        assert( value != 0 );

        strcpy( value, pStr );
    }

    int GetSize() {
        return size;
    }

    int GetLength() {
        return length;
    }

    void PrintLn() {
        cout << "\"" << value << "\"" << endl;
    }

    // Деструктор
    ~MString() {
```

```
        delete [] value;
    }

    // ...
};

#endif
```

Теперь при выходе объектной переменной из области видимости или при удалении из памяти объекта, размещенного в динамической памяти, вызывается деструктор:

```
void UsingMStrings() {

    // ...
    {
        // Начало некоторого блока
        MString mStr1( "string1" ); // Инициализация объекта:
                                    // вызывается конструктор ==>
                                    // объект размещается в стеке

        // ... Какие-то действия с объектом mStr1

    } // Конец блока: объект mStr1 вышел из области видимости
      // ВЫЗЫВАЕТСЯ ДЕСТРУКТОР

    //...

    MString *mStr2 = new MString( "string2" );
                        // Объект размещен в динамической памяти:
                        // вызывается конструктор ==>
                        // резервируется динамическая
                        // память под value

    // ... Какие-то действия с объектом mStr2

    delete mStr2; // Освободили память, занятую объектом mStr2
                  // ВЫЗЫВАЕТСЯ ДЕСТРУКТОР
}
```

Очевидно, что поскольку деструктор в большинстве случаев вызывается неявно, он не может иметь ни параметров, ни возвращаемого значения.

3.6. Основные возможности в связи с определением членов класса

В этом разделе мы рассмотрим основные разновидности членов класса: константы и константные функции, статические константы и переменные-члены

класса, статические методы класса, объектные и ссылочные переменные-члены класса, особенности их определения, инициализации и последующего использования. В данном разделе мы рассмотрим также вопрос о том, каким образом при вызове нестатической функции-члена класса ей передается информация об объекте, на котором вызвана эта функция.

Константные члены класса

Существует две разновидности константных членов класса: константные данные и константные функции.

Константные данные

В ряде случаев требуется обеспечить представление констант, действующих в области видимости класса и независимых для каждого объекта этого класса. В качестве примера определим класс `ImmutableString`, обеспечивающий представление строки с невозможностью ее дальнейшей модификации. В этом случае размер строки можно считать константным в течение всего времени существования объекта (и, как и всякую константу, это значение нельзя изменить). В листинге 3.9 особое внимание обратите на специальный синтаксис инициализации константы в заголовке конструктора (текст вслед за двоеточием после заголовка конструктора).

Листинг 3.9. Класс для работы с неизменяемой строкой

```
/*
 * ImString.h
 *
 * Определение класса для работы с неизменяемой строкой (фрагмент)
 */

#include <cstring>
#include <cassert>

#include <iostream>
using namespace std;

class ImmutableString {

    const int length; // Фактическая длина строки является константной
    char *value;

public:

    ImmutableString( const char *pStr = "" ) :
        length( strlen(pStr) ) // Инициализация константы
```

```
    {
        value = new char[ length + 1];
        assert( value != 0 );

        strcpy( value, pStr );
    }

    void PrintLn() const { // см. следующий раздел "Константные функции"
        cout << "\"" << value << "\"" << endl;
    }

    ~ImmutableString () {
        delete [] value;
    }
    // ...
};
```

Поскольку значение нестатической константы-члена класса может зависеть от переданных конструктору класса аргументов, технически невозможно записать ее инициализацию подобно инициализации обычной константы (в точке объявления с использованием знака присваивания). Внутри же конструктора нельзя поместить присваивание константе какого-либо значения: невозможность этого вытекает из определения константы, которой в принципе нельзя присвоить какое-либо значение. Поэтому для инициализации констант и ряда других требующих инициализации членов класса и используется специальный синтаксис.

Константные функции

Константные функции-члены класса не изменяют состояние объекта (отслеживается транслятором). Например, функция-член класса `MString`, печатающая строку на консоль, может быть объявлена константной:

```
void MString::PrintLn() const {
    cout << value;
}
```

Константные функции могут вызывать только другие константные функции-члены класса. Попытки вызвать функцию, относительно которой нельзя гарантировать неизменность объекта после ее работы, блокируются компилятором.

Статические константы, переменные и функции

Как уже было отмечено, явный недостаток приведенного в листинге 3.5 определения класса `MString` заключается в его зависимости от глобальной константы `DEFAULT_LEN`.

Таким образом, класс `MString` можно использовать только в том контексте, в котором константа `DEFAULT_LEN` определена. То есть имеется определенное ограничение на использование класса. Страуструп замечает по этому поводу: "Возможно, “всего лишь одна маленькая глобальная переменная” [в нашем случае это замечание применимо и к константе] не станет слишком неуправляемой, но подобный стиль приводит к коду, который непонятен для всех, кроме программиста, написавшего его. Этого следует избегать".

К счастью, существует возможность объявления в классе переменных-членов и констант-членов класса, которые, являясь элементами класса, не являются элементами объекта, или, иными словами, являются общими для всех объектов данного класса. Такие члены класса называются *статическими*. Статические члены класса позволяют, с одной стороны, обеспечить требуемую в подобных случаях независимость от состояния и поведения отдельного объекта, с другой стороны — обеспечить их существование в области видимости класса.

Классическим примером, иллюстрирующим использование статических данных, является реализация возможности "автономного" подсчета числа объектов одного типа, одновременно находящихся в памяти (см., например, [Рассохин, 1993]). Определение класса `MString`, содержащее статическую переменную для подсчета существующих объектов класса, представлено в листинге 3.10.

Листинг 3.10. Определение и реализация класса `MString`, содержащего статические члены

```
/*
 * MString.h
 *
 * Определение класса для работы со строками
 * Версия 2
 *
 * Используются статические члены класса
 *
 */

#ifndef _MString_h_
#define _MString_h_

#include <cstring>
#include <cassert>
```

```
class MString {

    static const int DEFAULT_LEN; // Длина строки по умолчанию
    static int counter;           // Число одновременно
                                  // находящихся в памяти
                                  // объектов типа MString

    int size;      // Размер памяти, отводимой для хранения строки
    int length;     // Действительная длина строки
    char *value;    // ASCIIZ-строка

public:
    // Конструктор #1:
    //   Инициализация пустой строки заданием длины
    explicit MString( int sz = DEFAULT_LEN );

    // Конструктор #2:
    //   Инициализация строки значением существующей
    //   ASCIIZ-строки
    MString( const char *pStr );

    // Деструктор
    ~MString();

    // Статическая функция, возвращающая значение счетчика объектов
    static int GetCounter();

    // ...
};

#endif

/*
 * MString.cpp
 *
 * Реализация методов класса для работы со строками
 */

#include <cstring>
#include <cassert>

#include "MString.h"

// Объявление и инициализация статической константы
const int MString::DEFAULT_LEN = 128;

// Объявление и инициализация статической переменной
int MString::counter = 0;
```

```
// Конструктор #1:
//   Инициализация пустой строки заданием длины
MString::MString( int sz ) {

    size = sz+1;
    value = new char[ size ];
    assert( value != 0 );

    length = 0;

    value[0] = '\0';

    counter++; // Увеличение счетчика объектов на 1
}

// Конструктор #2:
//   Инициализация строки значением существующей
//   ASCIIZ-строки
MString::MString( const char *pStr ) {

    length = strlen( pStr );
    size = length + 1;

    value = new char[ size ];
    assert( value != 0 );

    strcpy( value, pStr );

    counter++; // Увеличение счетчика объектов на 1
}

// Деструктор
MString::~MString() {

    counter--; // Уменьшение счетчика объектов на 1
    delete [] value;
}

// Функция, возвращающая значение счетчика объектов
// (статическая)
int MString::GetCounter() {
    return counter;
}

// ...
```

В данном примере статические члены объявлены как закрытые члены класса. Разумеется, они могут объявляться и с другими спецификаторами доступа.

К статическим членам класса можно обращаться без указания имени объекта[1]. В качестве квалификатора используется имя класса, который и образует область видимости статических данных.

В приведенном примере функция `GetCounter()` также объявлена статической (хотя она не обязана по правилам C++ таковой быть). Статический метод класса — это функция, которой требуется доступ к членам класса, но не требуется, чтобы она вызывалась для конкретного объекта класса. Поэтому статические функции не могут обращаться к нестатическим членам класса.

Поскольку функция `GetCounter()` работает только со статическими данными, она и не обязана существовать в контексте объекта. Кроме того, нам может потребоваться обратиться к ней даже в том случае, когда ни одного объекта типа `MString` еще не размещено в памяти, например (комментарии в теле функции содержат печатаемые значения счетчика объектов):

```
#include <iostream.h>
#include "MString.h" // (версия 2)

void TestFunction()
{
    cout << MString::GetCounter() << endl;      // 0
    {
        MString ms1( "abc" );
        cout << MString::GetCounter() << endl;  // 1
        cout << ms1.GetCounter() << endl;       // 1

        MString *pMs2 = new MString( 10 );
        cout << MString::GetCounter() << endl;  // 2
        cout << ms1.GetCounter() << endl;       // 2
        cout << pMs2->GetCounter() << endl;     // 2

        delete pMs2;
        cout << MString::GetCounter() << endl;  // 1
    }
    cout << MString::GetCounter() << endl;      // 0
}
```

Счетчики объектов не являются чисто теоретической идеей. Голуб приводит пример использования статического счетчика объектов для управления инициализацией внешних по отношению к проектируемому коду процессов [Holub, 1995]:

[1] Более того, в некоторых языках (например, в Java) к статическим данным вообще нельзя обращаться через объектную переменную, а только используя имя класса.

```
class window {
    static int num_windows;
public:
    window();
    ~window();
};

int window::num_windows = 0;

window::window() {
    if( ++num_windows == 1 ) {
         // Создается первое окно...
         initialize_video_system();
    }
}

window::~window() {
    if( --num_windows == 0 ) {
         // Уничтожается последнее окно...
         shut_down_video_system();
    }
}
```

Другой пример использования счетчиков *приводится в разд. 12.2* при рассмотрении подсчета ссылок на экземпляр компонента.

Замечание

В отличие от языка C++, в языках Java и C# принята более строгая модель, в соответствии с которой обращаться к статическим членам класса можно только из статических методов класса.

Члены класса, требующие обязательную инициализацию

Обязательную инициализацию требуют следующие члены класса:

- константы;
- ссылки;
- объекты, не имеющие конструктора по умолчанию.

Поскольку для таких членов класса нельзя написать операцию присваивания, их инициализация производится с использованием синтаксиса инициализаторов:

```
class X {
    int b;
public:
    // Конструктор есть, но он не является конструктором
    // по умолчанию
    X( int a ) {
        b = a;
    }
};

class T {
    const int a; // константа
    X mX;        // объект
                 // (класс X не имеет конструктора по умолчанию)
    X& mrefX;    // ссылка
public:
    T( int i,
       int j,
       X& refX ) :  a( i ),       // Инициализатор константы
                    mx( j ),      // Инициализатор объекта
                    mrefX( refX ) // Инициализатор ссылки
    { }
};
```

В свою очередь, если среди членов класса есть неинициализированные константы или ссылки (для которых инициализация обязательна), конструктор по умолчанию не может быть сгенерирован, поэтому отсутствие конструктора вызовет сообщение об ошибке:

```
class Y { // !!! определение класса содержит ошибку:
          // Нет конструктора по умолчанию,
          // при этом есть неинициализированная константа
    const int a;
public:
    // Конструктор есть, но он не является конструктором
    // по умолчанию
    Y( int initValue ) : a( initValue ) {}
};

class Z { // !!! правильное определение класса
          // Константа инициализирована
          // в конструкторе по умолчанию
    const int a;
public:
    Z( int initValue = 0 ) : a( initValue ) {}
};
```

Инициализация статических членов класса

Как уже было показано в примере листинга 3.10, определение и инициализация статических данных в C++ располагается за пределами объявления класса. Поскольку статические данные не являются частью объекта и существуют отдельно от объекта, а время их жизни (как и всех прочих статических данных) совпадает со временем жизни программы, объявление статической переменной или константы в области видимости класса не является определением. Определение статических данных производится на уровне модуля (обычно это модуль, содержащий реализации методов класса), поскольку статические данные участвуют в процессе связывания аналогично внешним данным, определяемым на глобальном уровне.

Если речь идет о статической переменной, то для задания ей начального значения можно написать специальный статический метод класса, например, определение класса `MString` в листинге 3.10 можно дополнить методом, задающим начальное значение `counter`:

```
class MString {
    //...
    static int counter;

public:
    // ...

    static void InitCounter() {
        counter = 0; // или любое другое значение, требующееся
                     // в контексте использования класса
    }
    // ...
};
```

Это не исключает необходимости помещения определения статической переменной в области видимости модуля:

```
int MString::counter;
```

Если явно не задано инициализирующее значение, то, как и все статические данные, `counter` инициализируется нулем. Таким образом, инициализация статических данных гарантирована. Наличие статического метода `InitCounter()` позволяет в случае необходимости в любой момент восстановить начальное значение:

```
void f() {
    //...
```

```
    MString::InitCounter();
    //...
}
```

Технически разделение объявления статических данных в классе и их определения на уровне модуля не очень удобно. Поэтому в других объектно-ориентированных языках определены иные формы синтаксиса инициализации статических членов. В Java статические члены инициализируются в точке объявления. В C# для инициализации статических членов класса используется специальный блок, называемый статическим конструктором. Статический конструктор вызывается автоматически до того, как будет создан первый объект класса или значение какого-либо статического данного будет использовано. Статический конструктор не имеет спецификатора доступа, не имеет параметров и не может быть вызван явно. На мой взгляд, статический конструктор — довольно удачная форма инициализации статических членов класса. Еще раз обращаю внимание читателя, что в C++ никаких статических конструкторов не предусмотрено.

Понятие о константном указателе *this*

Очень часто на практике возникает необходимость возвратить из функции-члена класса ссылку на объект-экземпляр такого класса.

Рассмотрим в качестве примера определение класса для представления комплексных чисел.

```
class MyComplex {
    double re;
    double im;
public:
    MyComplex( double real = 0.0, double image = 0.0 ) {
        re = real;
        im = image;
    }
    double GetRe() const { return re; }
    double GetIm() const { return im; }

    // ...
};
```

Предположим, что определение класса `MyComplex` должно содержать методы, реализующие некоторые арифметические действия, модифицирующие комплексные данные. В этом случае было бы неплохо, чтобы эти функции возвращали ссылки на измененный объект (листинг 3.11).

Листинг 3.11. Класс `MyComplex`: методы для арифметических вычислений

```
class MyComplex {
    double re;
    double im;
public:
    MyComplex( double real = 0.0, double image = 0.0 ) {
        re = real;
        im = image;
    }
    double GetRe() const { return re; }
    double GetIm() const { return im; }

    // Увеличить текущее значение на значение,
    // переданное в качестве аргумента
    MyComplex& Add( const MyComplex& arg2 );

    // Уменьшить текущее значение на значение,
    // переданное в качестве аргумента
    MyComplex& Sub( const MyComplex& arg2 );

    // Модифицировать текущее значение на значение,
    // умножив его на значение,
    // переданное в качестве аргумента
    MyComplex& Mul( const MyComplex& arg2 );

    // Модифицировать текущее значение на значение,
    // разделив его на значение,
    // переданное в качестве аргумента
    MyComplex& Div( const MyComplex& arg2 );

    // ...
};
```

В этом случае мы могли бы написать

```
MyComplex x, y, z, t;
x.Add( y ).Sub( z ).Mul( t );
```

вместо последовательных действий

```
x.Add( y );
x.Sub( z );
x.Mul( t );
```

Для доступа из метода класса к объекту класса, для которого вызван метод, в языке C++ существует служебное слово `this`, означающее объявлен-

ный неявно указатель на объект, для которого вызвана функция-член класса. Это почти то же самое, как если бы функция имела параметр `MyComplex *const this`.

Однако `this` не является обычной переменной: ей нельзя присвоить какое-либо значение или получить ее адрес.

При использовании указателя `this` реализация метода `Add()` может выглядеть следующим образом:

```
MyComplex& MyComplex::Add( const MyComplex& arg2 ) {
    re += arg2.re;
    im += arg2.im;

    return *this;
}
```

Обратите внимание на то, что закрытые переменные `re` и `im` объекта `arg2` доступны в функции `Add()`, поскольку `Add()` является членом класса `MyComplex`, а права доступа сформулированы на уровне классов, а не на уровне объектов.

Теперь рассмотрим более сложный пример (листинг 3.12), дополнив реализацию класса `MString` функцией конкатенации строк. Аргументом функции будет ссылка на другую строку типа `MString`. Функция возвращает ссылку на измененный объект типа `MString`.

Листинг 3.12. Конкатенация строк `MString`

```
MString& MString::Concat( const MString& ws ) {

    // Вычислим размер памяти, требуемой для хранения результирующей
    // строки
    int requiredSize = length + ws.length + 1;
    if( size >= requiredSize ) {
        // Здесь перераспределения памяти не требуется
        strcat( value, ws.value );
        length = requiredSize - 1;
        return *this;
    }

    // Здесь осуществляем перераспределение памяти
    char *copy = new char[ length + 1 ];
    strcpy( copy, value );

    delete [] value;
```

```
    size = requiredSize;
    length = requiredSize - 1;
    value = new char[ size ];

    strcpy( value, copy );
    strcat( value, ws.value );

    delete [] copy;

    return *this;
}
```

Дополним определение класса `MString` функцией, печатающей полную информацию о состоянии объекта:

```
void MString::Print() const {
    cout << "Size:" << size;
    cout << " Length:" << length;
    cout << " Value:\"" << value << "\"" << endl;
}
```

Использование метода `Concat()` иллюстрируют следующие функции:

```
void TestFunction1()
{
    // Иллюстрирует случай, когда для успешной конкатенации
    // требуется перераспределение памяти
    MString ms1 = "";
    cout << "Before catenation:" << endl;
    ms1.Print();

    MString ms2 = "abc";
    MString ms3 = "defgh";

    ms1.Concat( ms2 ).Concat( ms3 );

    cout << "After catenation:" << endl;
    ms1.Print();
}

void TestFunction2()
{
    // Иллюстрирует случай, когда для успешной конкатенации
    // НЕ требуется перераспределение памяти
    MString ms1( 20 );
    cout << "Before catenation:" << endl;
    ms1.Print();
```

```
    MString ms2 = "abc";
    MString ms3 = "defgh";

    ms1.Concat( ms2 ).Concat( ms3 );

    cout << "After catenation:" << endl;
    ms1.Print();
}
```

Рассмотрим тестирующую программу, содержащую обращения к функциям `TestFunction1()` **и** `TestFunction2()`:

```
void main() {
    TestFunction1();
    TestFunction2();
}
```

В результате исполнения этой программы получим следующие сообщения:

```
Before catenation:
Size:1 Length:0 Value:""
After catenation:
Size:9 Length:8 Value: "abcdefgh"
Before catenation:
Size:21 Length:0 Value:""
After catenation:
Size:21 Length:8 Value: "abcdefgh"
```

Вывод программы подтверждает, что во втором случае результирующая строка сохранена в памяти, связанной с объектом ранее, и перераспределения памяти не производится.

Представление о константном указателе `this` позволяет в новом свете взглянуть на константные и статические методы класса. Обычная (нестатическая и неконстантная) функция-член класса имеет доступ к `this`, получая его в качестве неявного аргумента. Статическая функция-член класса не получает доступа к `this`. Константная функция-член класса получает доступ к `this` только для чтения данных, доступных через этот указатель, и поэтому не может модифицировать члены класса.

Иными словами, для экземплярного метода можно полагать действующим объявление:

```
SomeClass *const this;
```

Для константного метода можно полагать действующим объявление:

```
const SomeClass *const this;
```

В статическом методе указатель `this` не определен.

Встроенные функции-члены класса

Напомним, что обычная функция C++ может быть объявлена встроенной, или подставляемой, с помощью спецификатора `inline`, помещаемого в начале заголовка функции. Спецификатор `inline` является указанием компилятору на то, что в месте вызова этой функции следует попытаться (если это возможно) встроить код функции, а не обращение к ней. В некоторых случаях компиляторы могут сгенерировать не просто код функции, а даже результат ее работы (если аргументы функции константны, а никаких побочных эффектов функция не производит).

Для объявления функции-члена класса в качестве встроенной имеются два способа.

Первый заключается в определении функции в пределах определения класса. Этот способ подходит для компактных и часто используемых функций, реализующих простые действия. Использование спецификатора `inline` в данном случае излишне (хотя и не запрещено). Так же как и определение класса, определение встроенной функции-члена класса в этом случае может повторяться в различных единицах трансляции (в результате использования директивы `#include`).

Второй способ заключается в объявлении функции с явным спецификатором `inline` с последующей реализацией функции за пределами определения класса. Это может облегчить чтение программы (особенно в том случае, когда встроенная функция использует значения переменных-членов класса, объявленных в конце определения класса), например:

```
class ContainsInline {
public:
    //...
    bool SetValue( int );
    bool IsCorrect( int );
    int GetValue() { return x }; // Встроенная функция

private:
    int x;
};

// Тоже встроенная функция
inline ContainsInline::SetValue( int value ) {
    if( IsCorrect( value)) {
        x = value;
        return true;
    }
    return false;
}
```

3.7. Перегруженные функции и операции-члены класса

В данном разделе мы рассмотрим варианты перегрузки методов класса, а также возможности перегрузки операций языка таким образом, чтобы их можно было использовать применительно к объектам проектируемого типа.

Перегрузка методов класса

Механизм перегрузки методов класса мало отличается от механизма перегрузки обычных функций C++. Напомним, что перегруженные имена функций удобны в тех случаях, когда функции выполняют схожие по смыслу действий задачи с данными различных типов. Идея перегружаемых имен вполне согласуется даже с естественными языками, где подобные перегружаемые имена используются нами довольно часто. Вот наглядный пример из английского языка (рис. 3.4).

to wash	+	a car	=	**помыть**	машину
to wash	+	a dress	=	**выстирать**	платье
to wash	+	one's face	=	**умыть**	лицо

Рис. 3.4. "Перегруженные имена" в естественных языках

Глагол "to wash" используется в английском языке для обозначения таких разных по организации действий, как "мыть" (например, машину), "стирать" (например, платье), "умывать" (например, лицо). Если бы английский язык, так же как и язык C, не допускал перегрузку имен, мы должны были бы употреблять уникальные идентификаторы для выявления разницы между этими действиями (например, конструируя что-нибудь вроде carwash a car, dresswash a dress, facewash one's face и т. д.). Вместо этого значение слова определяется контекстом его употребления [Eckel, 2000-2].

Примерно в этом и заключается смысл механизма перегрузки имен, который поддерживается большинством современных языков программирования. Ответственность за семантическую согласованность действий, обозначаемых одинаковыми именами, лежит на программисте: транслятор не имеет возможности проверить, действительно ли одноименные методы связаны с концептуально схожими задачами. Компилятор лишь обеспечивает автоматическую генерацию уникальных имен для всех перегруженных функций.

Один из вариантов перегрузки методов класса был рассмотрен ранее применительно к перегрузке конструкторов. Аналогичным образом можно перегружать и прочие методы класса.

Так же как и при перегрузке обычных функций, не являющихся членами класса, выбор подходящего метода осуществляется на основании последовательной проверки следующих критериев:

1. Точное соответствие типов или соответствие, достигаемое тривиальным преобразованием. Примеры таких тривиальных преобразований: преобразование имени массива в указатель, преобразование имени функции в указатель на функцию, преобразование типа в тот же тип с модификатором `const` (но не обратно!).
2. Соответствие, достигаемое неявными сохраняющими значение преобразованиями интегральных типов, или продвижениями (`char`, `short` в `int` или `unsigned int`; `wchar_t` в `int`, `unsigned int`, `long` или `unsigned long`; битовые поля в `int` или `unsigned int`; `bool` в `int`). Заметим, что основная задача продвижений — это создание значений типа `int` из "более коротких" целых типов, поэтому преобразование `int` в `long` не считается продвижением. Заметим, что продвижения обеспечивают сохранение значения и поэтому относятся к категории безопасных преобразований.
3. Соответствие, достигаемое путем стандартных преобразований (например, `int` в `long`, `long` в `double`, `int` в `double`, `int` в `char` и т. п.) или преобразований указателей или ссылок на производные типы в указатели или ссылки на базовые типы, а также указателей на произвольные типы в `void*`. В отличие от продвижений, стандартные преобразования могут быть потенциально опасными и приводить к потере значащих разрядов (например, `int` в `char`, `double` в `int` и т. п.).
4. Соответствия, достигаемые при помощи преобразований, определяемых пользователем посредством перегрузки операции преобразования типа.
5. Соответствие, достигаемое за счет использования переменного числа параметров функции (известными любому программисту на C примерами функций с переменным числом параметров являются функции из семейства `scanf` и `printf`). Следует отметить, что в программировании на C++ использование функций с переменным числом параметров весьма ограничено по сравнению с программированием на C именно ввиду наличия механизма перегрузки функций, который позволяет программисту явно определить возможные варианты наборов параметров функции или метода класса.

Если соответствие может быть достигнуто двумя или более способами на одном и том же уровне критериев, такая ситуация считается противоречивой, и компилятор выдаст сообщение об ошибке. Тип возвращаемого значения не

участвует в разрешении перегрузки из-за того, что, если бы функции различались только типом возвращаемого значения, в некоторых случаях было бы невозможно определить, какую версию функции вызывать (например, в случае, когда возвращаемое значение при вызове не используется).

Перегрузка арифметических операций

Перегрузка арифметических операций полезна тогда, когда хочется обеспечить привычную форму записи выражений при манипуляции с данными пользовательских типов. Запись выражений, оперирующих данными пользовательских типов в форме, привычной для выражений, оперирующих со встроенными типами, может в ряде случае обеспечивать более наглядный и естественный код.

Перегружаемые операции-члены класса

Определение класса `MyComplex` можно дополнить перегруженными версиями стандартных арифметических операций (листинг 3.13).

Листинг 3.13. Класс `MyComplex`: методы и операции для арифметических вычислений

```
class MyComplex {
    double re;
    double im;
public:
    MyComplex( double real = 0.0, double image = 0.0) {
        re = real;
        im = image;
    }
    double GetRe() const { return re; }
    double GetIm() const { return im; }

    MyComplex& Add( const MyComplex& arg2);
    MyComplex& Sub( const MyComplex& arg2);
    MyComplex& Mul( const MyComplex& arg2);
    MyComplex& Div( const MyComplex& arg2);

    MyComplex operator+( const MyComplex& arg2);
    MyComplex operator-( const MyComplex& arg2);
    MyComplex operator*( const MyComplex& arg2);
    MyComplex operator/( const MyComplex& arg2);

    // ...
};
```

Отличие функций `Add()`, `Sub()`, `Mul()`, `Div()` от соответственных операций заключается в том, что функции возвращают ссылку на измененный объект, а операции — значение нового объекта, например:

```
MyComplex MyComplex::operator+( const MyComplex& arg2) {

    return MyComplex( re + arg2.re, im + arg2.im);
}
```

Аналогично могут быть определены и другие перегруженные операции.

Используя перегруженные операции, программист имеет возможность записывать выражения над объектами класса `MyComplex` аналогично арифметическим выражениям с данными встроенных типов, например:

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

Следует учитывать, что хотя операции при перегрузке могут приобретать новый смысл, их приоритет и порядок выполнения не зависят от типов операндов и регулируются правилами языка. Так, если в приведенном выражении убрать скобки, первой будет выполняться операция умножения.

При перегрузке операций следует учитывать общепринятую семантику вычислений. Это касается типов операндов и типов возвращаемых значений. Операции, перегружаемые для объектов пользовательских типов, должны работать подобно операциям над данными встроенных типов.

Исходя их этого требования, можно заключить, что не следует перегружать операции `&&`, `||` и операцию "запятая", хотя формально это и не запрещено [Meyers, 1996]. Необходимость такого ограничения становится понятной, если проанализировать правила вычисления логических выражений в C++. Напомним, что вычисление логического выражения продолжается до тех пор, пока нет ясности относительно окончательного результата. Рассмотрим следующий фрагмент кода:

```
int f( int);
int x;
//...
if( x && f( x)) {
    // ...
}
```

При вычислении выражения `x && f(x)`, в том случае, когда `x` имеет значение 0, обработка выражения прекращается, а это означает, что функция `f()` не будет вызвана. Теперь рассмотрим реализацию с перегруженной операцией `&&`:

```
int operator &&( const int&, const int&);
int f( int);
int x;
// ...
if( x && f( x)) {
    // ...
}
```

Здесь семантика вычисления логического выражения "вытесняется" семантикой вызова функции (с "именем" `operator&&`). Поэтому прежде чем управление будет передано функции, должны быть вычислены значения всех аргументов, а это означает, что функция `f()` будет вызвана. Таким образом, последовательность вычислений во втором случае расходится с тем, что может предполагать программист, исходящий из принятой в C/C++ стандартной семантики вычисления логических выражений. Это же замечание относится и к операции "запятая" [Meyers, 1996, Item 7].

Перегружаемые операции, не являющиеся членами класса

В примере листинга 3.13 предполагалось, что программист имеет доступ к представлению класса и может определить перегруженную операцию-член класса. Если доступа к представлению класса нет, можно определить перегруженную операцию за пределами объявления класса. Разумеется, в этом случае доступ имеется только к открытым членам класса. Эквивалентное определение перегруженной операции, не требующее модификации класса `MyComplex`, выглядит следующим образом:

```
MyComplex operator+( const MyComplex& arg1,
                     const MyComplex& arg2) {
    double tempRe = arg1.GetRe() + arg2.GetRe();
    double tempIm = arg1.GetIm() + arg2.GetIm();

    return MyComplex( tempRe, tempIm);
}
```

При перегрузке для обеспечения корректной интерпретации выражений, содержащих операции в различных сочетаниях, следует помнить о том, что обычно приходится перегружать целое множество разнообразных операций. Например, перегрузка операции сложения `+` и операции присваивания `=`, разумеется, не обеспечивает корректную (или вообще какую-либо) семантику операции `+=`. Ее следует перегружать отдельно. *См. также информацию об определении перегружаемых операций в связи с иерархиями классов в разд. 5.2.*

Перегружаемые операции-друзья класса

В отличие от перегрузки операции +, перегрузка операции += требует доступа к представлению класса, поскольку модифицирует значения закрытых членов класса. В этом случае у разработчика имеется две альтернативы. Первая состоит в том, чтобы определить эту функцию аналогично методу `Add()` как открытый член класса (необходимые изменения в спецификацию класса внесите самостоятельно):

```
MyComplex& MyComplex::operator+=( const MyComplex& arg2) {
    re += arg2.re;
    im += arg2.im;

    return *this;
}
```

Вторая возможность заключается в предоставлении так называемого дружественного доступа к закрытым членам класса. Для того чтобы предоставить перегружаемой операции, не являющейся членом класса, дружественный доступ к закрытой части объявления в определении класса должно быть помещено объявление перегружаемой операции со спецификатором `friend`:

```
class MyComplex {
    //...

    friend MyComplex& operator+=( MyComplex&, const MyComplex&);
};
```

Реализация операции может быть тогда помещена за пределами класса:

```
MyComplex& operator+=(       MyComplex& arg1,
                       const MyComplex& arg2) {
    arg1.re += arg2.re;
    agr1.im += arg2.im;

    return arg1;
}
```

Перегрузка операций ввода-вывода

Особым и очень часто применяемым на практике вариантом перегруженных операций является перегрузка операций ввода-вывода данных. *Этот случай подробно обсуждается в материалах приложения 2 "Средства потокового ввода-вывода в C++" (разд. П2.1).*

Перегрузка других операций

Перегружать можно большинство операций C++, включая унарные операции, операции `new`, `delete`, `()`, `[]`, `->`, `->*`, а также операцию "запятая". Нельзя перегружать следующие операции:

- операцию выбора члена класса или элемента структуры (точка);
- условную (тернарную) операцию;
- операцию `sizeof`;
- операцию разрешения области видимости (двойное двоеточие).

Рассмотрим принцип перегрузки некоторых из перечисленных операций на примере. Вернемся к примеру с очередью целых чисел и рассмотрим еще раз механизм работы с переменными `last`, `next`, а также с переменной `ix` (в теле метода `Print()`) (см. листинги 3.3, 3.4). Как уже было отмечено, эти индексы модифицируются в соответствии с правилами вычислений по модулю, где второй аргумент операции `mod` принимает значение `size` (размер массива `body`) — см. рис. 3.2. Можно вынести специфику обработки таких данных за пределы класса `QueueInt`, определив класс для представления категории данных, обрабатываемых подобным образом (фрагмент определения класса представлен в листинге 3.14). В представленном фрагменте перегружены только те операции, которые необходимы для корректной работы с переменными такого рода в классе `QueueInt`. Полную версию читателю предлагается разработать самостоятельно *(см. упражнение 18 разд. 3.10)*.

Листинг 3.14. Определение класса для представления чисел по модулю `modArg`

```
/*
 * ModInteger.h
 *
 * Определение класса для представления и обработки данных,
 * обрабатываемых по правилам вычислений по модулю
 *
 */
#ifndef _ModInteger_h_
#define _ModInteger_h_

// Класс для представления целочисленных данных,
// обрабатываемых по правилам вычислений по модулю
class ModInteger {
    int value;         // Внутреннее представление числа
    static int modArg; // Второй аргумент операции mod
                       // (общий для всех объектов,
                       //  участвующих в вычислениях)
```

```
public:
    // Инициализация второго аргумента операции mod
    static InitModArgument( int _modArg) {
        modArg = _modArg;
    }

    // Перегрузка операции присваивания в случае,
    // если вторым аргументом присваивания является
    // целое число
    ModInteger& operator=( int _value) {
        this->value = _value % modArg;
        return *this;
    }

    // Перегрузка операции преобразования типа
    // из ModInteger в int
    operator int() const { return value; }

    // Перегрузка префиксной формы операции
    // инкремента (возвращает ссылку на объект!)
    ModInteger& operator++() {
        value = (value+1) % modArg;
        return *this;
    }

    // Перегрузка постфиксной формы операции
    // инкремента (возвращает значение объекта!)
    // NB! Искусственный аргумент int используется в C++
    //     для различения префиксной и постфиксной формы
    ModInteger operator++( int) {
        ModInteger temp = *this; // Значение до изменения
        value = (value+1) % modArg;
        return temp;
    }
    // ...
};
#endif
```

Обратите внимание, что следует не забыть поместить в область видимости модуля объявление и инициализацию статической переменной `modArg`. Явная инициализация `modArg` заменяет необходимость вызова статического метода `InitModArgument()`. В листинге 3.15 приведены определение и реализация класса `QueueInt` с учетом использования функциональности, предоставляемой классом `ModInteger`. В реализации (листинг 3.15) существенные изменения относительно предыдущей версии выделены полужирным начертанием. Под-

черкиванием выделены те фрагменты текста, которые не изменились, однако будут обрабатываться с учетом функциональности, предоставляемой классом ModInteger.

Листинг 3.15. Определение и реализация класса QueueInt, использующего определение ModInteger

```
/*
 * QueueInt.h
 *
 * Определение класса для работы с очередью
 * Версия 3.1 (с использованием класса ModInteger)
 *
 */

#include "ModInteger.h"

class QueueInt {  // Стандартная очередь из целых чисел:
                  // "первым пришел - первым обслужен"
private:
    //
    // Закрытые члены класса
    //(доступ только в пределах класса)
    //

    enum {
        EMPTY = -1,      // Признак пустоты очереди
        MAXSIZE = 100    // Предельный размер очереди
    };

    int body[ MAXSIZE ]; // Массив для хранения данных
    int size;            // Предельный размер
                         // инициализированной очереди

    int length;          // Текущая длина очереди

    ModInteger last;     // Индекс последнего занесенного
                         // элемента очереди (по модулю size)
    ModInteger next;     // Индекс последнего извлеченного
                         // элемента очереди (по модулю size)

public:
    //
    // Открытые члены класса
    //
```

```
    QueueInt ( int customSize = MAXSIZE);
                                    // Конструктор:
                                    // обеспечивает гарантированную
                                    // однократную инициализацию

    void Enqueue( int);  // Занесение элемента
    int  Dequeue();      // Извлечение элемента

    void Print();        // Печать текущего содержимого
};

/*
 * QueueInt.cpp
 * QUEUE of INTegers: implementation
 *
 * Реализация методов класса QueueInt
 *
 */

#include <iostream>
using namespace std;

#include "QueueInt.h" // (в версии 3.1)
#include "ModInteger.h"

int ModInteger::modArg; // Определения модуля

// Конструктор
QueueInt::QueueInt( int customSize) {

    if( customSize < 1 || customSize > MAXSIZE) {

        cout << "Incorrect size. ";
        cout << "Using default value.\n";

        size = MAXSIZE;
    }
    else size = customSize;

    ModInteger::InitModArgument( size);

    last = EMPTY;    // Преобразование типа из ModInteger
    next = 0;        // Преобразование типа из ModInteger

    length = 0;
}
```

```
// Добавление элемента в очередь
void QueueInt::Enqueue( int item) {

    if( length == size) {
        cout << "Enqueue error. ";
        cout << "Queue is full." << endl;

        return;
    }

    length++;

    body[ ++last ] = item;   // Префиксная операция из
                             // ModInteger + преобразование типа
                             // ModInteger -> int
}

// Извлечение элемента из очереди
int QueueInt::Dequeue() {

    if( length == 0) {
        cout << "Dequeue error. ";
        cout << "Queue is empty." << endl;

        return EMPTY;
    }
    // Переменная copy теперь не нужна:
    // работает постфиксная операция (две строки ниже)

    length--;

    return body[ next++ ];  // Постфиксная операция из
                            // ModInteger + преобразование типа
                            // ModInteger -> int

}

// Печать содержимого очереди
void QueueInt::Print() {

    if( length == 0) {
        cout << "Queue is empty." << endl;
        return;
    }

    int count;
    ModInteger ix;
    for( ix=next, count=1;
```

```
        count<=length;
        ix++, count++) {

      cout << body[ ix ] << " "; // Преобразование типа
                                 // ModInteger -> int
    }

    cout << endl;
}
```

Проанализируем работу методов класса `QueueInt` в редакции, представленной в листинге 3.15.

1. *Конструктор* `QueueInt`. Для обеспечения правильной работы с объектами класса `ModInteger` явно вызывается метод `InitModArgument()` (второй аргумент операции `mod` в нашем случае всегда `size`).
2. *Метод* `Enqueue()`. Правильное изменение индекса `last` теперь обеспечивается префиксной операцией `++`, перегруженной в классе `ModInteger`. Результатом операции является значение объекта класса `ModInteger`, а не целое число, подразумеваемое в операции взятия индекса `[]`. Однако в классе `ModInteger` явно определено преобразование объекта `ModInteger` в значение типа `int` в тех случаях, когда это преобразование требуется. Наличие данного преобразования и позволяет определить целочисленный индекс для обращения к элементу массива `body`.
3. *Метод* `Dequeue()`. Правильное изменение индекса `next` обеспечивается постфиксной операцией `++`, перегруженной в классе `ModInteger`. Результатом операции является ссылка на объект класса `ModInteger`, а не целое число, подразумеваемое в операции взятия индекса `[]`, поэтому здесь также работает явно определенное преобразование объекта `ModInteger` в целое число.
4. *Метод* `Print()`. Переменная `ix`, изменяющаяся аналогично `next`, также определена теперь как объект класса `ModInteger`, поэтому при вычислении индекса для обращения к элементу массива `body` здесь также работает явно определенное преобразование объекта `ModInteger` в целое число.

Несмотря на определенное изящество этого примера, при дальнейшей модификации класса `QueueInt` (а нам еще предстоит над ним поработать!) мы будем ориентироваться на реализацию в версии 3 (см. листинг 3.3), поскольку она существенно более проста в восприятии. Кроме того, затраты на реализацию класса `ModInteger` не обеспечивают значительного выигрыша в реализации кода для класса `QueueInt`.

Однако рассмотренный пример не бесполезен. Во-первых, проиллюстрирован целый комплект перегруженных операций, включая преобразование типа и

традиционно вызывающую затруднения перегрузку одноместных постфиксных и префиксных операций. Во-вторых, полноценная (а не фрагментарная) реализация класса `ModInteger` могла бы иметь и практическую пользу. Если, скажем, нам нужно разработать целую библиотеку классов, обеспечивающих представление очередей разных типов, то подобная версия с использованием класса `ModInteger` может оказаться полезной (затраты на реализацию класса `ModInteger` окупятся многократным его использованием).

Несколько замечаний по поводу перегрузки операций *new* и *delete*

Необходимость перегрузки операций управления ресурсами памяти может быть обусловлена разными обстоятельствами, в частности, соображениями повышения эффективности кода или реализацией механизма, отличного от стандартного, например, при реализации работы с иными формами организации памяти аналогично работе с динамической памятью (см. [Stroustrup, 2000, гл. 19]).

Говоря о перегрузке операций `new` и `delete`, следует разделять задачи собственно резервирования памяти и инициализации размещаемых в памяти объектов. Рассмотрим этот вопрос на примере операции `new`.

Вообще говоря, применительно к `new`, можно говорить о двух смыслах употребления термина "операция `new`". Во-первых, это встроенная операция языка (*new* operator), основное назначение которой состоит в выполнении следующей последовательности действий:

1. Выделение памяти под объект.
2. Вызов конструктора класса, обеспечивающего инициализацию объекта.

Например, в следующем фрагменте программы

```
class X;
X *obj = new X;
```

сначала выделяется память под объект класса `X`, а затем вызывается конструктор класса `X`. Эта семантика встроенной операции зафиксирована, и ее нельзя изменить. Для выделения памяти под объект вызывается специальная функция, которая называется "операция-new" (`operator new`). Вот эту операцию и можно перегрузить применительно к данным пользовательского типа. То есть единственное, что вы можете изменить, — это процедура распределения памяти (рис. 3.5).

Подробности в связи с перегрузкой операций управления ресурсами памяти см., например, в [Meyers, 1996, Item 8].

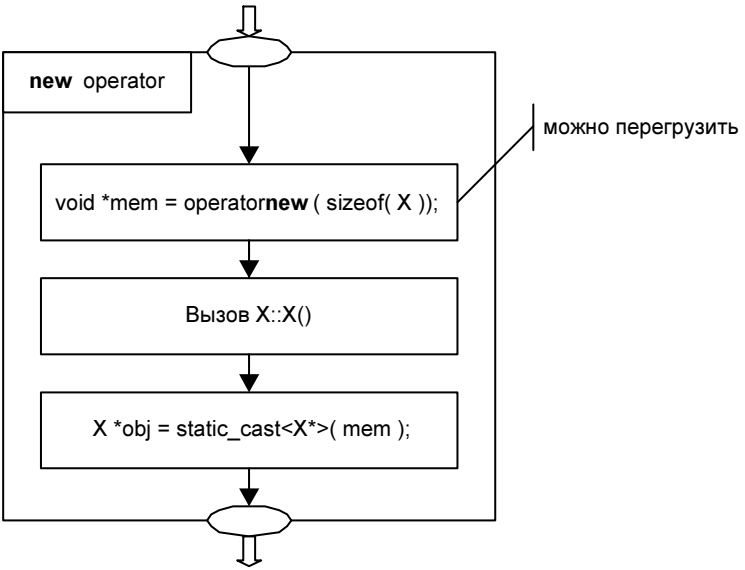


Рис. 3.5. Перегрузка операции `operator new`

3.8. Дружественный доступ. Локальные и вложенные классы

Выше приведены примеры предоставления дружественного доступа к закрытой части определения класса перегружаемым операциям. Дружественный доступ можно предоставить любой функции C++, которая является или не является членом какого-либо класса. При предоставлении дружественного доступа функции-члену другого класса, при ее объявлении в классе, предоставляющем доступ к своей закрытой части, необходимо указать полное имя функции (операции), включающее имя класса, членом которого она является:

```
class AllowingFriendAccess {
    //...

    friend void UsingFriendAccess::f();
};

class UsingFriendAccess {
    //...

    void f() {
        // Здесь есть доступ к закрытой части
        // класса AllowingFriendAccess
```

```
        //...
    }
};
```

Функция, получающая дружественный доступ, должна быть объявлена (но не обязательно определена) в той же области видимости, что и класс, который предоставляет дружественный доступ. Это объявление не обязательно должно предшествовать определению класса, предоставляющего дружественный доступ.

Если все функции одного класса являются друзьями другого класса, можно использовать более короткую форму записи, объявляя дружественный доступ для всех членов класса.

```
class UsingFriendAccess; // Обязательное предварительное
                         // объявление

class AllowingFriendAccess {
    //...

    friend class UsingFriendAccess;
};

class UsingFriendAccess {

    // Все функции этого класса имеют
    // доступ к закрытой части
    // класса AllowingFriendAccess
    //...
};
```

Отметим, что предоставление дружественного доступа лишь дает возможность обратиться к закрытой части класса из пределов дружественной функции (операции) или дружественного класса, но не помещает дружественный программный объект в область видимости класса.

Дружественный доступ не наследуется ("никто не обязан дружить с друзьями своих родителей") и не транзитивен ("друг моего друга не обязательно мой друг").

Несколько дополнительных замечаний о дружественном доступе

Применение спецификатора `friend` позволяет реализовать достаточно гибкую модель доступа. Однако ряд разработчиков языков программирования считают модель дружественного доступа недостаточно прозрачной (или, наоборот, слишком прозрачной ☺).

В связи с этим, например, в языке C# область применения концепции дружественного доступа сужена. Спецификатор `internal` предоставляет доступ к закрытым членам класса, входящим в одну сборку с объявляемым классом. В рамках вычислительной платформы Microsoft. NET сборка является автономной единицей распространения программного кода и может объединять один или несколько файлов *(см. разд. 12.5)*. Дополнительную гибкость управления данными в языке C# обеспечивает класс доступа `protected internal`, означающий, что член класса доступен как для классов, получающих доступ на уровне *protected*, так и для классов, получающих доступ на уровне *internal*.

В языке Java реализован механизм доступа по умолчанию, похожий на класс доступа *internal* в C#. Областью видимости при этом является пакет, определяющий пространство имен классов (пакет, однако, всегда представляет собой отдельный файл). *Подробно реализация модульности в языках Java и C# рассматривается в разд. 8.3 и 8.4.*

Локальные и вложенные классы

Определение класса может размещаться внутри определения функции или блока. В этом случае говорят о локальном классе. Областью видимости локального класса является тело функции или блок, в котором содержится объявление класса.

Для локальных классов существует два ограничения:

1. Функции-члены класса должны быть встроенными и обязательно помещаться в пределах определения класса.
2. Нельзя использовать статические члены класса, т. к. имя класса не видно за пределами блока, и определение статических членов класса становится невозможным.

Определение класса может размещаться непосредственно внутри определения другого класса. Такой класс называется *вложенным*. Обычно вложенные классы применяют в том случае, когда хотят подчеркнуть осмысленность их использования только в пределах пространства имен основного класса. Часто применение вложенного класса является альтернативой использованию дружественного доступа. Полная квалификация имени члена вложенного класса включает имя основного класса и имя вложенного класса, например:

```
class Process {
public:
    class ErrorInProcess {
        static int errorsCountered;
```

```
    public:
        ErrorInProcess() { errorsCountered++; }
        int isError();
    };
    // ...
};

int Process::ErrorInProcess::errorsCountered = 0;

bool Process::ErrorInProcess::isError() {
    return errorsCountered;
}
```

Следует обратить внимание на то, что объявление вложенного класса является только объявлением типа и не влечет создания объектов. Поэтому в пределах вложенного класса можно непосредственно использовать из вмещающего класса только имена типов, статических членов и перечислений. Для обращения к другим членам вмещающего класса, равно как и к членам прочих классов, следует использовать указатели, ссылки или объектные переменные:

```
template <class T>
class Outer {
public:
    typedef T  value_type;
    value_type var;
    static value_type stat;

    class Inner { // Определение вложенного класса
    public:
        void SetValues( const value_type& x) {
                    // Верно: value_type - имя типа

            var = x; // Ошибка: var - экземплярная переменная
            stat= x; // Верно: stat - статическая переменная
        }
        void SetVar( Outer *p, const value_type& x) {
            p->var = x; // Верно: Outer::var - открытый член
        }
    }
};
```

При этом следует иметь в виду, что члены вложенного класса не имеют особого доступа к членам вмещающего класса. Однако иногда бывает удобно предоставить вложенному классу доступ к закрытой части вмещающего класса. Для этого следует объявить вложенный класс другом вмещающего класса:

```
class Outer {
    int internal;
public:
    class Inner; // Предварительное объявление
    friend class Inner; // Предоставление дружественного
                        // доступа
    class Inner { // Определение вложенного класса
    public:
        void ResetInternal( Outer& out) {
            out.internal = 0; // internal доступна
        }
    };
};
```

Практический пример подобной организации класса см. в разд. 9.4 (определение контейнерного класса в стиле STL).

3.9. Копирование объектов класса

Внимательный читатель должен был заметить, что в некоторых рассмотренных ранее примерах встречалось присваивание объектов (см., например, присваивание объектов класса `ModInteger` в листинге 3.15) или использование объектов, возвращаемых методами классов по инструкции `return` (см., например, листинг 3.13). В подобных случаях возникает копирование объектов. Несмотря на то, что в указанных примерах копирование не порождает никаких проблем, в общем случае копирование объектов не всегда безопасно.

Проблемы копирования объектов и понятие поверхностного копирования

Копирование объекта может встретиться, например, при инициализации объекта в точке объявления:

```
MyComplex a( 5, 3);
MyComplex b = a; // Инициализация копированием
```

Аналогично, объекты могут копироваться при помощи присваивания:

```
void CopyComplex( MyComplex& a) {
    MyComplex b;
    b = a; // Копирование присваиванием
    //...
}
```

В обоих случаях подразумевается почленное копирование объекта, иначе называемое поверхностным копированием.

В том случае, когда у объекта есть члены, являющиеся указателями, такая интерпретация копирования может вовсе не совпадать с пожеланиями программиста. В качестве примера рассмотрим копирование объектов класса `MString` (определение класса см. в листинге 3.8):

```
MString mStr1( "String1");
MString mStr2( 10);

mStr2 = mStr1;
```

В результате подобного присваивания значение указателя `value` объекта `mStr2` будет указывать на ту же память, что и указатель `value` объекта `mStr1`, т. е. результатом поверхностного копирования стало копирование значения указателя (рис. 3.6). Это означает, что любое последующее изменение состояния объекта `mStr1`, например вызов метода `Concat()`, будет воздействовать и на объект `mStr2`.

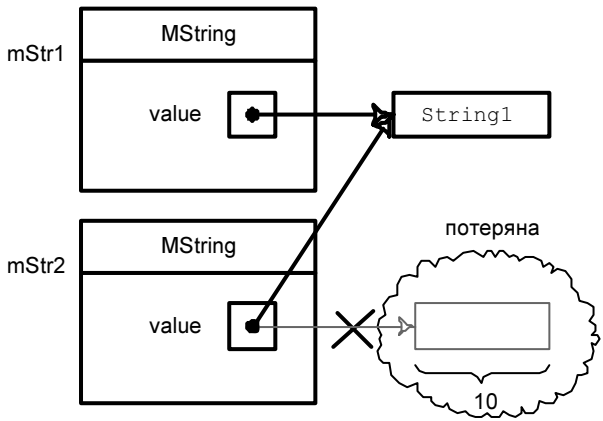


Рис. 3.6. Поверхностное копирование объектов присваиванием

Использование объектом ресурсов, управляемых конструктором и деструктором (применительно к `MString` речь идет о массиве символов, размещаемом в динамической памяти), может привести к серьезным ошибкам при нарушении семантики копирования объектов:

```
void f() {
    MString mStr1( "String1");
    MString mStr2 = mStr1; // Инициализация копированием: ПРОБЛЕМА
```

```
    MString mStr3( "Lost_string");
    mStr3 = mStr2; // Копирующее присваивание: ПРОБЛЕМА
}
```

В данном примере конструктор `MString` будет вызван дважды — для `mStr1` и для `mStr3`. Для `mStr2` конструктор не вызывается, т. к. данный объект инициализируется при помощи копирования (рис. 3.7). Указатель на память, содержащую ASCIIZ-строку `"Lost_string"`, будет перезаписан в результате копирующего присваивания, поэтому эта память при отсутствии автоматической сборки мусора будет навсегда потеряна для программы *(об автоматической сборке мусора см. разд. 10.3)*.

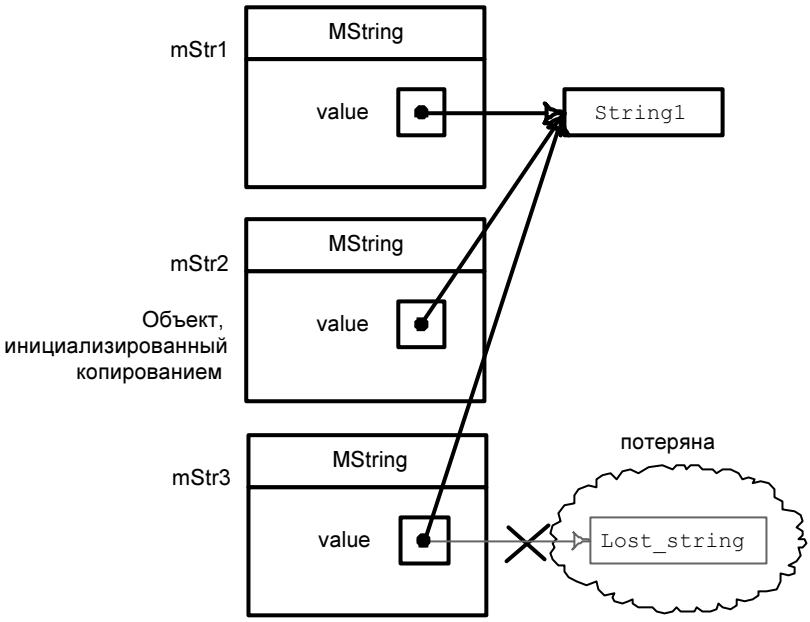


Рис. 3.7. Проблемы поверхностного копирования (инициализация и присваивание объектов)

По завершении работы функции `f()` (при выходе всех объектов из области видимости) деструктор будет вызываться для всех трех экземпляров класса!

Поскольку объекты разрушаются в порядке, обратном порядку создания объектов, первым автоматически (поскольку объекты размещены в стеке) будет вызван деструктор объекта `mStr3`, что приведет к освобождению памяти, занятой строкой, содержащей текст `String1`, а затем к освобождению памяти, занятой самим объектом (рис. 3.8). В результате этого оказывается, что переменная `value` для объектов `mStr1` и `mStr2` хранит значение адреса несущест-

вующей в памяти строки, что вызовет ошибку при попытке освобождения этой памяти в ходе работы деструктора объекта `mStr2`.

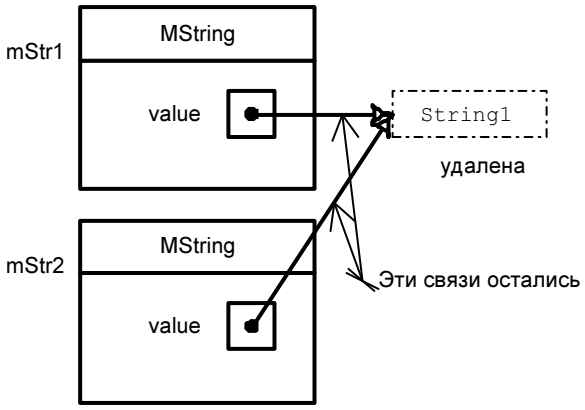


Рис. 3.8. Ситуация после вызова деструктора объекта `mStr3`

Таким образом, результатом реализации семантики поверхностного копирования стало некорректное управление памятью, занимаемой, используемой или адресуемой объектами.

Реализация корректной семантики копирования объектов

Вероятно, желательным поведением программы в приведенных примерах было бы копирование содержимого памяти, связанного с указателем `value`, а не копирование самого указателя.

Для исправления семантики копирования по умолчанию программист должен явно определить подходящий смысл копирующей инициализации и копирующего присваивания применительно к конкретной ситуации. Достигается это явным определением специального конструктора (называемого копирующим конструктором) и перегрузки операции присваивания для объектов данного класса.

Копирующий конструктор и перегрузка операции присваивания

Применительно к `MString`, решение может выглядеть следующим образом:

```
class MString {
    // ...
    MString( const MString&);
```

```
    MString& operator=( const MString&);
    // ...
};

// Копирующий конструктор
MString::MString( const MString& ms) {
    size = ms.size;
    length = ms.length;
    value = new char[ size ];
    strcpy( value, ms.value);
}

// Перегруженная операция присваивания
MString& MString::operator=( const MString& ms) {
    // Копируем, если не ссылается сам на себя
    if( this != &ms)
    {
        delete [] value;
        size = ms.size;
        length = ms.length;
        value = new char[ size ];
        strcpy( value, ms.value);
    }
    return *this;
}
```

Подобный механизм реализации копирования объектов иногда называют глубинным копированием, или рекурсивным копированием (имея в виду, что в некоторых случаях объекты могут указывать сами на себя, следовательно, явная реализация копирования может оказаться рекурсивной по своей природе).

Рассмотрите подробную реализацию класса `MString`, учитывающую наличие конструктора копирования и перегруженной операции присваивания, и программу тестирования, представленные в листинге 3.16.

> ... не удаляйте тест-процедуру; ...утром вы удалите эту тестовую функцию, а уже к обеду она вам понадобится снова, даже если вы не пользовались ей последние два года.
>
> *Ален Голуб*

Листинг 3.16. Реализация класса `MString`, содержащего копирующий конструктор и перегруженную операцию присваивания, и работа с объектами класса `MString`

```
/*
 * MString.h
 *
```

```
 * Определение класса для работы со строками
 * Версия 4 - дополнена копирующим конструктором и
 *            перегруженной операцией присваивания
 */

#ifndef _MString_h_
#define _MString_h_

#include <cstring>
#include <cassert>

class MString {

    static const int DEFAULT_LEN; // Длина строки по умолчанию

    int size;      // Размер памяти, отводимой для хранения строки
    int length;     // Действительная длина строки
    char *value;    // ASCIIZ-строка

public:
    // Конструктор #1:
    //   Инициализация пустой строки заданием длины
    explicit MString( int sz = DEFAULT_LEN);

    // Конструктор #2:
    //   Инициализация строки значением существующей
    //   ASCIIZ-строки
    MString( const char *pStr);

    // Копирующий конструктор
    MString( const MString&);

    // Перегруженная операция присваивания
    MString& operator=( const MString&);

    // Деструктор
    ~MString();

    // Функция конкатенации строк
    MString& Concat( const MString& ws);

    // Функция печати текущего содержимого объекта
    void Print() const;
    // ...
};

#endif
```

```
/*
 * MString.cpp
 *
 * Реализация методов класса для работы со строками (в версии 4)
 */

#include <cstring>
#include <cassert>

#include "MString.h"

// Объявление и инициализация статической константы
const int MString::DEFAULT_LEN =128;

// Конструктор #1:
//   Инициализация пустой строки заданием длины
MString::MString( int sz) {

    size = sz+1;
    value = new char[ size ];
    assert( value != 0);

    length = 0;

    value[0] = '\0';
}

// Конструктор #2:
//   Инициализация строки значением существующей
//   ASCIIZ-строки
MString::MString( const char *pStr) {

    length = strlen( pStr);
    size = length + 1;

    value = new char[ size ];
    assert( value != 0);

    strcpy( value, pStr);
}

// Копирующий конструктор
MString::MString( const MString& ms) {
    size = ms.size;
    length = ms.length;
    value = new char[ size ];

    strcpy( value, ms.value);
}
```

```
// Деструктор
MString::~MString() {

    delete [] value;
}

// Перегруженная операция присваивания
MString& MString::operator=( const MString& ms) {
    // Копируем, если не ссылается сам на себя
    if( this != &ms)
    {
        delete [] value;

        size = ms.size;
        length = ms.length;
        value = new char[ size ];

        strcpy( value, ms.value);
    }
    return *this;
}

// Функция конкатенации строк
MString& MString::Concat( const MString& ws) {

    // Вычислим размер памяти, требуемой для хранения результирующей
    // строки
    int requiredSize = length + ws.length + 1;
    if( size >= requiredSize) {
        // Здесь перераспределения памяти не требуется
        strcat( value, ws.value);
        length = requiredSize - 1;
        return *this;
    }

    // Здесь осуществляем перераспределение памяти
    char *copy = new char[ length + 1 ];
    strcpy( copy, value);

    delete [] value;

    size = requiredSize;
    length = requiredSize - 1;
    value = new char[ size ];

    strcpy( value, copy);
    strcat( value, ws.value);
```

```
    delete [] copy;

    return *this;
}

// Функция печати текущего содержимого объекта
void MString::Print() const {
    cout << "Size:" << size;
    cout << " Length:" << length;
    cout << " Value:\"" << value << "\"" << endl;
}

// ...

/*
 * MStringApp.cpp
 *
 * Программа тестирования класса MString(в версии 4)
 */

#include <iostream>
using namespace std;

#include "MString.h"

void main()
{
    // Создаем объект ms1, инициализируемый значением строки "abc"
    cout << "Creating ms1:" << endl;
    MString ms1( "abc");
    ms1.Print();

    // Создаем объект ms2 копированием содержимого объекта ms1
    cout << "Creating ms2 by copy contructor:" << endl;
    MString ms2 = ms1;
    ms2.Print();
    // Изменяем состояние объекта ms2
    // NB! Если семантика инициализации копированием реализована верно,
    //     объект ms1 не должен быть затронут
    cout << "Modifying ms2:" << endl;
    ms2.Concat( "def");
    ms2.Print();

   // Создаем объект ms3, инициализируемый заданием допустимой длины
    cout << "Creating ms3:" << endl;
    MString ms3( 20);
```

```
    ms3.Print();
    // Изменяем состояние объекта ms3:
    // теперь объект хранит строку "def"
    cout << "Modifying ms3:" << endl;
    ms3.Concat( "def");
    ms3.Print();

    // Создаем объект ms4, инициализируемый значением строки "gh"
    cout << "Creating ms4:" << endl;
    MString ms4( "gh");
    ms4.Print();

    // Изменяем состояние объекта ms4 присваиванием значения объекта ms3
    cout << "Modifying ms4 by assignment of ms3 value:" << endl;
    ms4 = ms3;
    ms4.Print();

    // Изменяем состояние объекта ms3
    // NB! Если семантика присваивания реализована верно,
    //     объект ms4 не должен быть затронут
    cout << "Modifying ms3:" << endl;
    ms3.Concat( "qwerty");
    ms3.Print();

    // Убеждаемся в корректности состояний объектов
    cout << "Printing current object states:" << endl;
    cout << "ms1 => "; ms1.Print(); // должен хранить "abc"
    cout << "ms2 => "; ms2.Print(); // должен хранить "abcdef"
    cout << "ms3 => "; ms3.Print(); // должен хранить "defqwerty"
    cout << "ms4 => "; ms4.Print(); // должен хранить "def"
}
```

Анализируя сообщения, печатаемые в ходе выполнения программы, можно убедиться в корректной реализации семантики копирования в процессе инициализации объектов и копирования присваиванием:

```
Creating ms1:
Size:4 Length:3 Value:"abc"
Creating ms2 by copy contructor:
Copy constructor called
Size:4 Length:3 Value:"abc"
Modifying ms2:
Size:7 Length:6 Value:"abcdef"
Creating ms3:
Size:21 Length:0 Value:""
```

```
Modifying ms3:
Size:21 Length:3 Value:"def"
Creating ms4:
Size:3 Length:2 Value:"gh"
Modifying ms4 by assignment of ms3 value:
Overloaded assignment used
Size:21 Length:3 Value:"def"
Modifying ms3:
Size:21 Length:9 Value:"defqwerty"
Printing current object states:
ms1 => Size:4 Length:3 Value:"abc"
ms2 => Size:7 Length:6 Value:"abcdef"
ms3 => Size:21 Length:9 Value:"defqwerty"
ms4 => Size:21 Length:3 Value:"def"
```

Запрет копирования объектов данного класса

Несмотря на то, что задачи, которые решает копирующий конструктор, вообще говоря, отличаются от задач, решаемых перегрузкой операции присваивания, в большинстве случаев одновременно определяют и копирующий конструктор, и перегруженную операцию присваивания. Однако иногда встречаются ситуации, когда, например, объекты разрешается инициализировать копированием, но в дальнейшем запрещается изменять посредством присваивания. В качестве примера приведем копирование объектов класса `ImmutableString` (предварительное определение класса см. в листинге 3.9). Инициализация неизменяемой строки копированием, конечно, возможна. Поэтому конструктор копирования следует определить. При этом нужно не забыть о том, что один из членов класса `ImmutableString` является константой, для корректной инициализации которой следует применить синтаксис инициализации констант. Что касается присваивания, то семантика неизменяемой строки предполагает невозможность изменения значений членов класса в ходе работы с созданным объектом, поэтому операция присваивания объектов класса `ImmutableSting` должна быть запрещена. Это достигается помещением объявления перегруженной операции присваивания в закрытую часть определения класса.

Определение класса `ImmutableSting`, дополненное конструктором копирования и запрещенной операцией присваивания, представлено в листинге 3.17.

Листинг 3.17. Класс для работы с неизменяемой строкой, дополненный конструктором копирования

```
/*
 * ImString.h
 *
```

```
 * Определение класса для работы с неизменяемой строкой,
 * (версия 2 - с конструктором копирования)
 */

#include <cstring>
#include <cassert>

#include <iostream>
using namespace std;

class ImmutableString {

    const int length; // Фактическая длина строки является константной
    char *value;

public:
    ImmutableString( const char *pStr = "") :
        length( strlen(pStr)) // Инициализация константы
    {
        value = new char[ length + 1];
        assert( value != 0);

        strcpy( value, pStr);
    }

    // Конструктор копирования
    ImmutableString( const ImmutableString& ref) :
        length( ref.length) // Инициализация константы
    {
        value = new char[ length + 1];
        assert( value != 0);

        strcpy( value, ref.value);
    }

    void PrintLn() const {
        cout << "\"" << value << "\"" << endl;
    }
    ~ImmutableString () {
        delete [] value;
    }
    // ...
private:
    // Присваивание объектов ImmutableString запрещено:
    ImmutableString& operator=( const ImmutableString& ref);
};
```

Нижеследующий пример программы иллюстрирует работу конструкторов класса `ImmutableString`.

```
void main() {
    ImmutableString is1( "abc"); // Работает основной конструктор
    is1.PrintLn(); // Печатается строка "abc"

    ImmutableString is2; // Работает основной конструктор
    is2.PrintLn(); // Печатается строка ""

    ImmutableString is3 = is1; // Работает конструктор копирования
    is3.PrintLn();// Печатается строка "abc"

    // ! Если снять знак комментария со следующей инструкции,
    // компилятор выдаст сообщение об ошибке:
    // "can not access private member"
    // is2 = is3; // NB! Эта строка содержит ошибку!
}
```

Если правильное поведение объекта вообще исключает инициализацию копированием и использование копирующего присваивания, то достаточно сделать закрытыми и копирующий конструктор, и операцию присваивания:

```
class X {
private:
    X( const X&);
    X& operator= ( const X&);
    // ...
public:
    // ...
}
```

3.10. Вопросы и упражнения

1. Сформулируйте основные недостатки спецификации очереди целых чисел, определяющей реализацию, рассмотренную в *разд. 3.1.*
2. Спецификатор `private` закрывает доступ к члену класса извне. Обеспечивает ли такое ограничение доступа полную защиту от преднамеренного некорректного использования класса? Постарайтесь обосновать свой ответ.
3. Рассмотрите определение класса:

```
class Test {
private:
    int x;
```

```
public:
    void InitX( const Test& arg) {
        x = arg.x;
    }
    //...
};
```

Переменная `x` является закрытым членом класса. Как Вы считаете, правомерно ли использовать запись `arg.x` в функции `InitX()`. Обоснуйте свой ответ.

4. Почему для инициализации объекта плохо подходит обычная функция-член класса?
5. Какую основную задачу решает конструктор класса?
6. Когда вызывается конструктор? Почему конструктор не может иметь возвращаемого значения?
7. Может ли конструктор класса быть объявлен со спецификатором доступа `private`? Если да, то поясните смысл такого объявления. Если нет, объясните, почему это невозможно.
8. Можно ли определить в классе более одного деструктора? Если да, то объясните, когда может оказаться необходимым использовать несколько вариантов деструктора. Если нет, объясните, почему это невозможно.
9. Может ли деструктор вызываться явно? Если да, то объясните, в каком контексте может понадобиться такой вызов. Если нет, объясните, почему это невозможно.
10. Рассмотрите определение класса и объясните, почему при трансляции данного текста будет выдано сообщение об ошибке. Как исправить ошибку.

```
class ContainsConst {
private:
    const int x;
public:
    ContainsConst() {
        x = -1;
    }
    // ...
};
```

11. Рассмотрите текст программы и объясните, какие строчки функции `main()` вызовут сообщения компилятора об ошибках.

```
struct Type0 {
    int x;
};
class Type1 {
    int x;
};
class Type2 {
    int x;
    Type2(){}
};
class Type3 {
    int x;
public:
    Type3( int t){ x = t; }
};
class Type4 {
    int x;
public:
    Type4( int t=0){ x = t; }
};
class Type5 {
    const int a;
public:
    Type5( int initValue = 0) : a( initValue){}
};
class Type6 {
    int a;
public:
    Type6( int initValue) { a = initValue; }
};
void main() {
    Type0 obj0;
    Type1 obj1;
    Type1 *array1 = new Type1[ 10 ];
    Type2 obj2;
    Type2 *array2 = new Type2[ 10 ];
    Type2 *pObj2;
    Type3 *pObj3 = new Type3( 10);
    Type3 *pObj31;
    Type3 obj3;
    Type4 obj4;
    Type5 *pObj5 = new Type5;
    Type6 *pObj6 = new Type6;
    //...
}
```

12. Приведите пример функции-члена класса, в которой нельзя использовать `this`. Объясните, почему нельзя.

13. Для приведенного ниже фрагмента программы запишите в местах, указанных комментариями, примеры обращений к соответствующим функциям.

```
void f( int x) {
    // ...
}

class A {
public:
    void f( int x) {
        // ...
    }
    static void g() {
        // ...
    }
};

void main() {
    A a;
    // Здесь запишите возможные варианты обращения к функции
    // static void g() - члену класса A
    // ...

    // Здесь запишите возможные варианты обращения к функции
    // void f( int) - члену класса A
    // ...

    // Здесь запишите возможные варианты обращения к функции
    // void f( int), определенной в начале файла
    // ...
}
```

14. Исходя из определения класса `MString`, представленного в листинге 3.6, рассмотрите текст следующей функции и объясните: сколько раз в ходе исполнения данной функции будет вызван конструктор класса `MString`? А деструктор? Почему строка, хранящаяся в `ms3`, будет потеряна?

```
void f() {
    MString ms1( "string1");
    MString ms2 = ms1;
    MString ms3( "String_that_will_be_lost");

    ms3 = ms2;
};
```

15. В каких из перечисленных случаев возникает необходимость копирования объектов (ответы обоснуйте и проиллюстрируйте примерами):
    - при инициализации объекта в точке объявления значением другого объекта;
    - при передаче объекта в функцию по значению;
    - при передаче объекта в функцию по ссылке;
    - при размещении объекта в динамической памяти;
    - при формировании функцией возвращаемого значения ссылки на объект;
    - при присваивании объектов.
16. Перегрузите операцию сложения для конкатенации строк, хранящихся в объектах класса `ImmutableString`. Не забудьте, что эти строки неизменяемы, поэтому результатом конкатенации должно стать создание нового объекта класса `ImmutableString`. Поясните отличие Вашей реализации от реализации метода `Concat()` класса `MString` в листинге 3.10.
17. Напишите пример программы, иллюстрирующей приводимые в *разд. 3.7* рассуждения об опасности перегрузки операций `&&`, `||` и операции "запятая". Пример должен демонстрировать, что последовательность вычислений, осуществляемых с использованием указанных перегруженных операций, отличается от последовательности вычислений, определяемой стандартными правилами языка.
18. Доработайте определение класса `MyComplex` таким образом, чтобы, по-вашему, он наиболее удачно выражал концепцию комплексного данного и предоставлял удобное множество операций для манипулирования комплексными данными. Напишите консольное приложение, использующее спроектированный класс.
19. Разработайте определение класса `Rational`, воплощающего абстракцию рациональной дроби. Перегрузите операции языка C++ таким образом, чтобы была возможность записывать арифметические выражения над объектами класса `Rational`. Какие методы класса Вы считаете нужным реализовать? Напишите тестирующую программу, манипулирующую данными типа `Rational`.
20. Доработайте и протестируйте определение класса `ModInteger` для вычислений по модулю.
21. Определите и реализуйте класс `IntSet` для множества целых чисел. Реализуйте операции объединения, пересечения и симметрической разности двух множеств [Stroustrup, 2000].

22. Определите и реализуйте класс для выполнения целочисленных арифметических вычислений с произвольным числом значащих цифр. Протестируйте его, вычислив значения 2^{4096} и 1 000!

23. С использованием класса, спроектированного для решения упражнения 22, решите известную задачу о ханойских башнях для произвольного числа дисков. Напомним формулировку задачи. Имеется три колышка, на один из которых нанизано заданное число дисков разного диаметра (например, 1 000), образующих пирамиду (внизу лежит самый большой диск, на нем — самый большой из оставшихся, и т. д.; сверху — самый маленький диск). Требуется переставить пирамиду на один из двух свободных колышков. Перестановка пирамиды представляет собой набор перемещений дисков. За одно перемещение диск переносится с одного колышка на другой, причем нельзя класть диск большего диаметра на диск меньшего диаметра. Требуется выяснить минимальное число перемещений, требуемых для переноса всей пирамиды с одного колышка на другой и напечатать это значение в виде целого десятичного числа.

24. Переработайте решение *упражнения 4 разд. 2.6* с учетом изученного материала в связи с конструированием и инициализацией объектов. В частности, замените методы, устанавливающие начальные состояния объектов, соответствующими конструкторами.

25. Перепишите программу из *упражнения 4 разд. 1.4* в объектно-ориентированном стиле. Сравните полученные реализации, укажите достоинства и недостатки каждой из них.

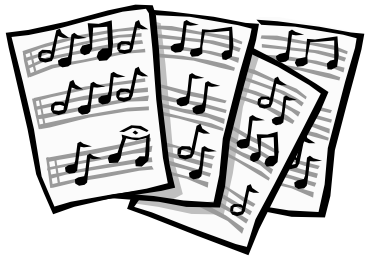
26. Напишите и протестируйте реализацию методов класса `StackInt`, ориентируясь на приводимое ниже определение класса. Напоминаем, что стек — это очередь типа LIFO (Last Input — First Output), т. е. извлекается всегда тот элемент, который был занесен в стек позднее других.

```
class StackInt {
public:
    enum {              // Предопределенные константы:
        EMPTY = -1,     //   Стек пуст
        MAXSIZE = 100   //   Предельный максимальный размер стека
    };

private:
    int *body;          // Указатель на начало массива, хранящего
                        // целочисленные данные, записываемые в стек
    int size;           // Действительный размер стека
    int top;            // Вершина стека (EMPTY, если стек пуст)
```

```
public:
    Stack( int actualSize = MAXSIZE);
                        // Конструктор обеспечивает размещение в
                        // динамической памяти массива, адрес начала
                        // которого записывается в переменную body.
                        // Размер массива определяется переданным
                        // аргументом, но не должен превосходить
                        // MAXSIZE
    ~Stack();           // Деструктор освобождает занятую память

    void Push( int);  // Занесение данного в стек
    int  Pop();         // Извлечение данного из стека

    void Print();       // Печать содержимого стека
};
```

27. Подумайте, нужно ли дополнить определение класса из *упражнения 26* копирующим конструктором. Обоснуйте свой ответ и обсудите его с Вашим преподавателем или коллегой.

ГЛАВА 4



Наследование и иерархии классов

В *гл. 2* обзорно представлены основные задачи, решаемые наследованием, и виды иерархий наследования в объектно-ориентированной программе. В данной главе рассматривается наследование как основной механизм построения иерархий классов и основные средства языка, обеспечивающие реализацию наследования.

4.1. Наследование как основная форма отношения обобщения

Как было указано в *гл. 2*, наследование является самой распространенной в объектно-ориентированных программах формой отношения обобщения. При наследовании базовый класс реализует наиболее общую модель сущности, а производный класс уточняет и развивает определение базового класса.

Можно ли обойтись без наследования

Можно ли вообще обойтись без наследования и какие преимущества получает программист, выстраивая иерархию классов на базе отношения наследования?

Для ответа на эти вопросы рассмотрим пример фрагментов определений типов в программе, обрабатывающей информацию о студентах некоторого учебного заведения (несмотря на то, что подобные примеры обычно ненавистны студентам, их достоинство заключается в том, что основное содержание представляемых данных и действий интуитивно понятно и не требует дополнительных пояснений):

```
struct Date { // Структура для представления календарной даты
    int day;
    int month;
    int year;
};

class Student { // Сведения о студенте вуза
public:
    string name;          // Имя
    string surname;       // Фамилия
    string displayName;   // Имя, используемое при печати
    Date birth;           // Дата рождения
    // ...
};
```

Далее нам может потребоваться определить, скажем, аспирантов, которые во многом подобны студентам, но для которых необходимо обеспечить представление дополнительных сведений (тема диссертации, номер специальности и т. п.).

Задачу объявления такого типа можно было бы разрешить, используя отношение агрегирования (рис. 4.1).

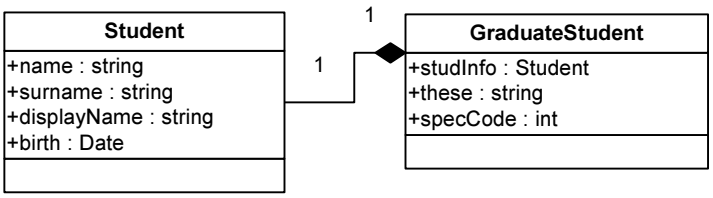


Рис. 4.1. Конструирование класса `GraduateStudent` на основе отношения композиции

Соответствующий фрагмент определения на C++ в этом случае будет выглядеть так:

```
class GraduateStudent { // Сведения об аспиранте
public:
    Student studInfo;  // Аспирант как студент

    // Только для аспиранта
    string these;      // Тема диссертации
    int specCode;      // Код специальности
    // ...
};
```

Итак, аспирант в таком контексте также является студентом: соответствующие данные объявлены в классе `GraduateStudent`.

Основные недостатки такого представления заключаются в следующем. Тип `GraduateStudent` является независимым, поэтому нельзя трактовать его как разновидность типа `Student`. А это означает, что нельзя будет использовать один объект там, где требуется другой, даже если специфика обработки объектов в некотором контексте сходна, в частности, нельзя будет создать общий массив со сведениями обо всех студентах и аспирантах.

Какие преимущества получает программист, используя отношение наследования классов

Наиболее правильным и естественным решением в этом случае будет явное указание на то, что объект типа `GraduateStudent` является разновидностью объекта типа `Student` и, кроме того, характеризуется некоторой дополнительной информацией. Дополнив классы некоторыми методами, которые будут полезны для изучения особенностей поведения объектов классов, получим структуру, представленную на рис. 4.2.

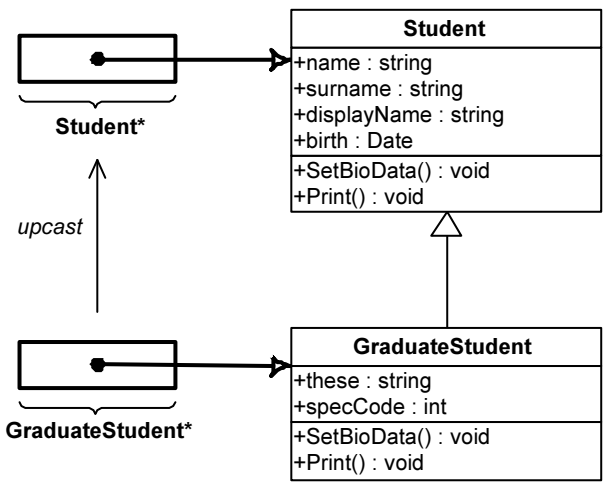


Рис. 4.2. Класс `GraduateStudent` наследует тип `Student` (повышающее приведение типов)

Пока оставим за скобками вопрос о правах доступа к членам класса при их наследовании, объявив все члены класса со спецификатором доступа `public`. Учитывая, что предложенные определения имеют модельный характер, все определяемые функции снабдим выводом информационных сообщений. Это

позволит нам проследить, какие функции и в каком порядке вызываются. С учетом этих оговорок определения классов на C++ выглядят примерно так (листинг 4.1).

Листинг 4.1. Определения классов `Student` и `GraduateStudent` (версия 1)

```
/*
 * Students.h
 *
 * Определение классов в связи с представлением
 * информации о студентах и аспирантах
 * (версия 1)
 */

#ifndef _Students_h_
#define _Students_h_

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

struct Date {
    int day;
    int month;
    int year;
};

class Student {
public:
    string name;
    string surname;
    string displayName;
    Date birth;

    void SetBioData( string _name, string _surname,
                     int day, int month, int year ) {

        name = _name;
        surname = _surname;
        birth.day = day;
        birth.month = month;
        birth.year = year;

        displayName = name + " " + surname;
    }
```

```
    void Print(){
        cout << "From Student::Print() => Name: ";
        cout << displayName << endl;
    }

    // ...
};

class GraduateStudent : public Student {
public:
    string these;
    int specCode;
    // ...

    void SetBioData( string _name, string _surname,
                     int day, int month, int year,
                     int _specCode, string _these ) {

        Student::SetBioData( _name, _surname,
                             day, month, year );

        specCode = _specCode;
        these = _these;
    }

    void Print() {
        Student::Print();
        cout << "\tFrom GraduateStudent::Print()";
        cout << " => SpecCode:";
        cout << specCode << endl;
        cout << "\tThese: " << these << endl;
    }
};

#endif
```

Поскольку объект типа `GraduateStudent` является разновидностью объекта типа `Student`, функция, требующая указатель на `Student`, может получать и указатель на `GraduateStudent`, иными словами, тип `GraduateStudent*` можно использовать в качестве типа `Student*`, например:

```
void PrintBirthDate( const Student* stud ) {
    cout << "From PrintBirthDate() => BirthDate: ";
    cout << (stud->birth).day << ".";
    cout << (stud->birth).month << ".";
    cout << (stud->birth).year << endl;
```

```
    // ...
}

void InformGraduate( const GraduateStudent* grad ) {
    // Получает указатель на GraduateStudent...

    // ...
    // .. и передает его как указатель на Student:
    PrintBirthDate( grad );
}
```

Нижеследующий фрагмент иллюстрирует возможность обработки объектов производных классов функциями, спроектированными для базовых классов, при условии передачи этим функциям указателей на объекты:

```
#include "Students.h" // (в версии 1)
void main() {

    // Создание и использование объекта типа Student
    Student *ps = new Student;
    ps->SetBioData( "John","First", 1, 4, 1980 );
    PrintBirthDate( ps );
    delete ps;

    // Создание и использование объекта типа GraduateStudent
    GraduateStudent *pgs = new GraduateStudent;
    pgs->SetBioData( "William", "Second", 12, 6, 1976,
        2204, "Software testing methods" );
    InformGraduate( pgs );
    delete pgs;
}
```

Вывод тестирующей программы будет выглядеть следующим образом:

```
From PrintBirthDate() => BirthDate: 1.4.1980
From PrintBirthDate() => BirthDate: 12.6.1976
```

Таким образом, в той части представления `GraduateStudent`, которая унаследована от `Student`, обработка осуществлена функциями, которые могут "ничего не знать" о существовании класса `GraduateStudent`.

Аналогично дело обстоит и в случае использования ссылочных типов:

```
void PrintBirthDateRef( const Student& refStud ) {

    cout << "From PrintBirthDateRef() => BirthDate: ";
    cout << refStud.birth.day << ".";
```

```
    cout << refStud.birth.month << ".";
    cout << refStud.birth.year << endl;
    // ...
}

void InformGraduateRef( const GraduateStudent& refGrad ) {
    // Получает ссылку на объект типа GraduateStudent...

    // ...
    // .. и передает ее в качестве ссылки на Student:
    PrintBirthDateRef( refGrad );
}
```

Таким образом, в рамках той функциональности, которая спроектирована в связи с классом `Student`, могут обрабатываться и объекты производного класса `GraduateStudent`. Неявное преобразование типа `GraduateStudent*` к типу `Student*` (или типа `GraduateStudent&` к типу `Student&`) называют *повышающим приведением типа* (upcast, см. рис. 4.2). Причина появления такого термина связана с тем, что при изображении иерархии классов базовый класс традиционно изображается вверху.

Используя отношение наследования, программист получает следующие преимущества:

- возможность повторного использования кода (это достигается и агрегированием);
- возможность отразить взаимоотношения объектов, свойственные предметной области;
- возможность рассматривать производный класс как подкласс базового, упрощая реализацию кода для производного класса;
- возможность обработки производных классов методами, разработанными при проектировании базового класса;
- возможность обновления или изменения содержания этих методов применительно к объектам производного класса, если такая необходимость имеется.

Теперь рассмотрим, как происходит обращение к методам класса `Student` в том случае, когда одноименные методы определены в производном классе. Рассмотрим простую тестирующую программу (листинг 4.2), иллюстрирующую обращение к методу `Print()` классов `Student` и `GraduateStudent` (внимательно изучите комментарии в функции `main()`).

Примечание

Все имена и даты в предлагаемом примере являются вымышленными, а возможные совпадения могут быть объяснены только чистой случайностью.

Листинг 4.2. Повышающие приведения типов (upcasts) при наследовании

```
/*
 * StudentTest.cpp
 *
 * Функции для тестирования особенностей обработки
 * объектов классов Student и GraduateStudent (в версии 1)
 */

#include "Students.h" // (в версии 1)

typedef Student *PStudent; // Тип: указатель на Student

void CallPrint( Student *stud ) {
    stud->Print();
}

// Основная программа
void main() {

    // Создание и использование объекта типа Student
    Student s1;
    s1.SetBioData( "John", "Smith", 1, 1, 1975 );
    s1.Print(); // Вызывается Student::Print()

    // Создание и использование объекта типа GraduateStudent
    GraduateStudent gs1;
    gs1.SetBioData( "Jeffrey", "Writer", 14, 7, 1963,
        2201, "Applied .Net Framework Programming" );
    gs1.Print(); // Вызывается GraduateStudent::Print()

    // Указатель на объект типа Student
    // получает адрес объекта типа GraduateStudent
    Student *ptrGs1 = &gs1;
    ptrGs1->Print(); // NB! Вызывается Student::Print(),
                     // поскольку для транслятора ptrGs1
                     // имеет тип Student*, а не
                     // GraduateStudent*

    // Создание еще одного объекта типа GraduateStudent
    GraduateStudent gs2;
    gs2.SetBioData( "Bjarne", "Strawberry", 25, 12, 1950,
        2204, "The C++ Programming Language" );

    // Создание массива указателей на объекты
    PStudent *all = new PStudent[ 3 ];
```

```
                    // Связывание...
    all[ 0 ] = &s1;  //  с объектом "John Smith"
    all[ 1 ] = &gs1; //  с объектом "Jeffrey Writer"
    all[ 2 ] = &gs2; //  с объектом "Bjarne Strawberry"

    // Печать информации обо всех студентах
    for( int ix = 0; ix < 3; ix++ ) {
        CallPrint( all[ ix ] ); // NB! Вызывается
                                // Student::Print()!
    }

    delete [] all;
}
```

Обратите внимание, что при вызове метода `Print()` из функции `CallPrint()` все три раза будет вызываться версия, объявленная в классе `Student`, хотя последние два указателя в действительности указывают на объекты класса `GraduateStudent`. Вывод тестирующей программы будет выглядеть следующим образом:

```
From Student::Print() => Name: John Smith
From Student::Print() => Name: Jeffrey Writer
    From GraduateStudent::Print() => SpecCode: 2201
    These: Applied .Net Framework Programming
From Student::Print() => Name: Jeffrey Writer
From Student::Print() => Name: John Smith
From Student::Print() => Name: Jeffrey Writer
From Student::Print() => Name: Bjarne Strawberry
```

В чем же дело? Почему для объектов класса `GraduateStudent` функция `CallPrint()` не обеспечивает вызов "правильной" версии функции `Print()`? Внимательный читатель должен вспомнить пример с геометрическими фигурами из *разд. 1.3* и обнаружить причину: функция `Print()` объявлена в базовом классе `Student` без спецификатора `virtual`, требующегося по правилам C++ для разрешения замещения функций при наследовании. Однако данное наблюдение не позволяет объяснить действительное устройство механизма разрешения вызовов при наследовании.

Точный ответ на поставленный вопрос невозможно дать без анализа одной из важнейших концепций объектно-ориентированного программирования — концепции позднего, или динамического, связывания, чем мы и *займемся в гл. 5*. Однако прежде рассмотрим некоторые важные особенности определения методов производного класса в отношении к методам базового класса и правила доступа к членам базового класса из производных классов, а также при вызове методов базового класса на объектах производных классов.

4.2. Методы производного класса в отношении к методам базового класса

Основное назначение наследования заключается в развитии и уточнении абстракций, определяемых базовым классом. В связи с этим необходимо разобраться в управлении доступом к членам базового класса в производном классе и при работе с объектами производного класса.

Переопределение функций-членов базового класса в производном классе

Разрабатывая производный класс, программист может переопределить унаследованную функцию-член базового класса, если содержание метода базового класса его не устраивает. Пример такого переопределения иллюстрирует объявление в классе `GraduateStudent` функции `Print()` (см. листинг 4.1):

```
class Student {
    // ...
public:
    void Print() {
        // ...
    }
};

class GraduateStudent : public Student {
    // ...
public:
    void Print() { // Переопределенная (redefined) функция
        // ...
    }
};
```

При этом сохраняется возможность обратиться к функции-члену базового класса с использованием явного разрешения области видимости, например:

```
class GraduateStudent : public Student {
    // ...
public:
    void Print() {
        Student::Print();  // Сначала вызвать функцию-член
                           // базового класса...
        // ... затем реализовать необходимую дополнительную
        // функциональность
    }
};
```

Про переопределяющую функцию говорят в этом случае, что она *скрывает* (hides) доступ к функции-члену базового класса, причем переопределяющая функция скрывает одноименный метод базового класса *независимо от того, совпадают ли сигнатуры методов*. Таким образом, если базовый класс содержит несколько перегруженных имен методов, то объявление одноименного метода в производном классе скрывает все унаследованные методы базового класса (листинг 4.3).

Листинг 4.3. Сокрытие имен при наследовании

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

class Base {
public:
    void g( int x ) {
        cout << "void Base::g( int ) => x = " << x << endl;
    }

    void g( char *s ){
        cout << "void Base::g( char* ) => s = " << s << endl;
    }
};

class Derived : public Base {
public:
    void g( int x, int y ) {
         cout << "void Derived::g( int, int ) => ";
         cout << "x = " << x << " y = " << y << endl;
    }
};

void main() {
    Derived d;

    d.g( 4, 5 );   // Верно: вызывается Derived::g( int, int )

    // Если в следующем обращении к g() снять ведущий знак комментария,
    // компилятор выдает сообщение об ошибке (нет такого метода)
    // d.g( "test" ); // Ошибка (выявляется при компиляции):
                      // нет нужной перегруженной версии
                      // (все методы g() базового класса скрыты)

    d.Base::g( "test2" ); // Правильно:
                          // вызывается Base::g( char* )
}
```

Поскольку перегруженные функции с именем `g` в базовом классе определены со спецификатором `public`, обратиться к ним можно при условии явного указания области видимости класса:

```
d.Base::g( "test2" ); // Вызов открытой функции-члена базового класса
                      // с использованием явного разрешения области
                      // видимости
```

Диаграмма на рис. 4.3 наглядно иллюстрирует механизм действия сокрытия имен в C++.

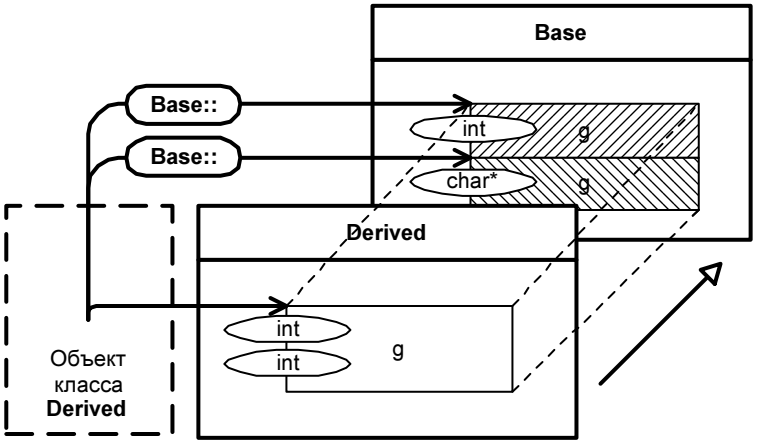


Рис. 4.3. Сокрытие имен при наследовании

Инициализация объектов производных классов

Если базовый класс имеет конструктор, он должен быть вызван. Конструкторы по умолчанию могут быть вызваны неявно. Если базовый класс не имеет конструктора по умолчанию, следует обеспечить его явный вызов. Достигается это использованием синтаксиса инициализации членов (листинг 4.4).

Листинг 4.4. Явная инициализация подобъекта базового класса

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

class Base {
public:
    Base( int x ) { // Конструктор базового класса
                    // с обязательным аргументом: должен вызываться явно
```

```
        cout << "Base initialized with value: " << x << endl;
        //...
    }
    // ...
};

class Derived : public Base {
public:
    Derived( int a, int b ) : Base( a ) {
        cout << "Derived initialized with values: " << a << " " << b;
        cout << endl;
        // ...
    }
    // ...
};

void main() {
    Derived drv( 5, 10 );
}
```

В результате исполнения этой программы будет напечатано:

```
Base initialized with value: 5
Derived initialized with values: 5 10
```

Конструктор производного класса может содержать инициализаторы как для инициализации собственных членов, так и для инициализации членов базового класса, однако члены базового класса могут инициализироваться только путем вызова конструктора базового класса. Нельзя из производного класса непосредственно инициализировать члены базового класса, даже если они объявлены со спецификатором `public` (см. рис. 4.4). Пусть имеется, например, следующее определение базового класса:

```
class Base {
public:
    const int a;
    Base1( int x ) : a( x ) {
        // ...
    }
    // ...
};
```

Тогда такая инициализация объекта производного класса корректна:

```
class Derived : public Base {
public:
    const int b;
```

```
    Derived( int x, int y ) : Base( x ), b( y ) {
        // ...
    }
    // ...
};
```

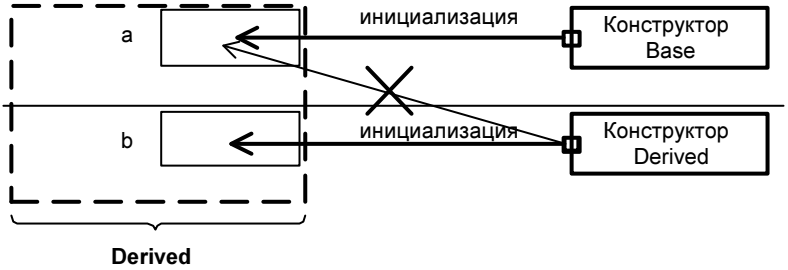


Рис. 4.4. Инициализация членов базового класса при создании объекта производного класса

Напрямую (минуя вызов конструктора базового класса) инициализировать член базового класса нельзя:

```
class Derived : public Base {
public:
    const int b;
    Derived( int x, int y ) : a( x ), b( y ) { // ОШИБКА!
        // ...
    }
    // ...
};
```

В соответствии с порядком наследования раньше вызываются конструкторы тех классов, которые находятся выше в иерархии наследования.

Наследование конструкторов и операций

С наследованием конструкторов дело обстоит предельно просто: конструкторы никогда не наследуются!!! Это относится и к копирующим конструкторам.

Если операция копирующего присваивания не определена, соответствующий код будет сгенерирован компилятором автоматически (т. е. будет реализована семантика поверхностного копирования). Отсюда следует, что операции присваивания также не наследуются.

Однако при этом сохраняется возможность использовать конструкторы копирования и перегруженные операции присваивания, определенные в базовом

классе, если функции обработки объекта принимают указатель или ссылку на объект базового класса (которые, как уже было показано ранее, в действительности могут ссылаться на объекты производного класса), например:

```
class Student {
    // ...
public:
    Student( const Student& ); // Конструктор копирования
    Student& operator=( const Student& ); // Перегруженная операция
                                          // присваивания
    // Далее как в листинге 4.1...
};

class GraduateStudent : public Student {
    // См. листинг 4.1...
};

void f( const GraduateStudent& grad ) {
     Student s = grad;      // Создание s из части grad
     s = grad;              // Копирование присваиванием части grad в s
}
```

В данном примере следует обратить внимание на то, что объект класса `Student` "ничего не знает" об объекте класса `GraduateStudent`, поэтому будет скопирована только часть объекта класса `GraduateStudent`: подобный эффект часто называют срезкой. Иногда это является источником ошибок.

Порядок вызова конструкторов и деструкторов в связи с иерархией классов

Как было отмечено ранее, при создании объектов производных классов в первую очередь вызываются конструкторы базовых классов, сначала — корневого класса иерархии, затем — производного от него класса, и т. д.

Вызов деструкторов (при наличии таковых) осуществляется в обратном порядке. Сначала вызывается деструктор класса, определяющего тип объектной переменной, затем — деструктор его базового класса, и т. д. В последнюю очередь вызывается деструктор корневого класса в иерархии классов. Ниже-следующий пример иллюстрирует особенности вызова конструкторов и деструкторов (листинг 4.5).

Листинг 4.5. Порядок вызова конструкторов и деструкторов

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

```
class Root {
public:
    Root() { cout << "Root constructed" << endl; }
    ~Root() { cout << "Root destroyed" << endl; }
};

class Base : public Root {
public:
    Base() { cout << "Base constructed" << endl; }
    ~Base() { cout << "Base destroyed" << endl; }
};

class Derived : public Base {
public:
    Derived() { cout << "Derived constructed" << endl; }
    ~Derived() { cout << "Derived destroyed" << endl; }
};

class MostDerived : public Derived {
public:
    MostDerived() { cout << "MostDerived constructed" << endl; }
    ~MostDerived() { cout << "MostDerived destroyed" << endl; }
};

void main() {
    Base *drv = new Derived;
    {
        MostDerived mostDrv;
        // ...
    } // Здесь разрушается объект mostDrv
    delete drv; // Здесь разрушается объект, управляемый указателем
                // drv
}
```

Данная программа порождает следующий вывод (сообщения пронумерованы для наглядности):

```
(1)Root constructed
(2)Base constructed
(3)Derived constructed
(4)Root constructed
(5)Base constructed
(6)Derived constructed
(7)MostDerived constructed
(8)MostDerived destroyed
```

```
 (9)Derived destroyed
(10)Base destroyed
(11)Root destroyed
(12)Base destroyed
(13)Root destroyed
```

Первые три сообщения выдаются в процессе создания экземпляра класса `Derived`, связанного с указателем `drv`. Сначала инициализируется подобъект класса `Root` в составе подобъекта класса `Base` (работает конструктор `Root`). Затем инициализируется подобъект класса `Base` (работает конструктор `Base`). Наконец, инициализируется сам объект класса `Derived` (работает конструктор `Derived`). Сообщения 4—7 обусловлены аналогичной последовательностью вызова четырех конструкторов при создании в стеке объекта `mostDrv` класса `MostDerived`. При выходе переменной `mostDrv` из области видимости в обратном порядке вызываются деструкторы всех классов (сообщения 8—11). Наконец, сообщения 12 и 13 свидетельствуют о вызове деструкторов классов `Base` и `Root`. Деструктор класса `Derived` не вызывается, несмотря на то, что указатель `drv` указывает на объект класса `Derived`. Это связано с тем, что переменная `drv` объявлена как указатель на тип `Base`, поэтому "с точки зрения компилятора" с `drv` связан объект, точный тип которого может быть неизвестен во время компиляции, но, по крайней мере, он совместим с `Base`.

Примечание

Забегая немного вперед, укажем, что обеспечить реализацию управления объектами таким образом, чтобы при разрушении объектов учитывался действительный тип объекта в памяти, а не тип времени компиляции, возможно. Собственно, именно это обстоятельство и приведет к определенной ревизии данного раздела в связи с изучением полиморфного поведения объектов в *гл. 5*.

4.3. Управление доступом к членам класса в связи с наследованием

Определение правил доступа к членам класса обеспечивается применением спецификаторов доступа при объявлении членов класса и при объявлении самого производного класса.

Спецификаторы доступа, указываемые при объявлении членов класса, управляют доступом к членам класса. Напомним, что члены класса, объявленные со спецификатором `public`, доступны как в функциях-членах класса, так и за их пределами, т. е. могут использоваться при работе с объектами класса, а также в функциях-членах производных классов.

Члены класса, объявленные со спецификатором `private`, доступны только в функциях-членах класса.

Наконец, если необходимо предоставить доступ к членам класса для функций-членов производного класса, но при этом закрыть доступ извне (т. е. для функций-членов других классов, а также для всех объектов), используется спецификатор доступа `protected`.

Производный класс может в свою очередь оказаться базовым классом для других классов. Права доступа к элементам базового класса из производного класса автоматически вытекают из определения базового класса.

Однако если планируется развивать иерархию наследования далее, то может потребоваться ограничить права доступа к членам базового класса из производных классов нижних уровней иерархии. Для этого используется спецификатор доступа, указываемый при объявлении производного класса. Это делается в том случае, когда специализация базового, обеспечиваемая на некотором уровне иерархии, является в определенной части окончательной и не допускает изменений в производных классах нижних уровней иерархии. Поясним сказанное на примерах.

Спецификатор `public`, указанный при наследовании, сохраняет все права доступа к членам, унаследованным от старшего базового класса, такими, какими они определены в старшем базовом классе:

```
class Base {
public:
    void fPubl();
protected:
    void fProt();
private:
    void fPrvt();
};

class Derived1 : public Base {
public:
    Derived1() {
        fPubl(); // доступна
        fProt(); // доступна

        // fPrvt() недоступна
    }
};

class DerivedDerived1 : public Derived1 {
public:
    DerivedDerived1() {
        fPubl(); // доступна
        fProt(); // доступна
```

```
        // fPrvt() недоступна
    }
};

void main() {
    Base b;
    b.fPubl(); // доступна

    // b.fProt() недоступна
    // b.fPrvt() недоступна

    Derived1 d1;
    d1.fPubl(); // доступна

    // d1.fProt() недоступна
    // d1.fPrvt() недоступна

    DerivedDerived1 dd1;
    dd1.fPubl(); // доступна

    // dd1.fProt() недоступна
    // dd1.fPrvt() недоступна
}
```

Класс `Derived1` наследует классу `Base` со спецификатором `public`. Это означает, что в классе `Derived1` сохраняются все соглашения по правам доступа к унаследованным из `Base` членам. Следовательно, объектам класса `Derived1` доступны `public`-члены класса `Base`, а в классе `DerivedDerived1` доступны те же члены класса `Base`, что и в классе `Derived1` с теми же спецификаторами. Поскольку `public`-члены класса `Base` "транзитивно" унаследованы классом `DerivedDerived1` со спецификатором `public`, то к этим членам можно обратиться и на *объектах* класса `DerivedDerived1`.

В следующем примере спецификатор `protected` ограничивает доступ к унаследованным членам таким образом, что их теперь возможно использовать только в пределах производных классов, и нельзя использовать извне:

```
class Base {
public:
    void fPubl();
protected:
    void fProt();
private:
    void fPrvt();
};
```

```
class Derived2 : protected Base {
public:
    Derived2() {
        fPubl(); // доступна
        fProt(); // доступна

        // fPrvt() недоступна
    }
};

class DerivedDerived2 : public Derived2 {
public:
    DerivedDerived2() {
        fPubl(); // доступна
        fProt(); // доступна

        // fPrvt() недоступна
    }
};

void main() {
    Base b;
    b.fPubl(); // доступна

    // b.fProt() недоступна
    // b.fPrvt() недоступна

    Derived2 d2;

    // d2.fPubl() недоступна
    // d2.fProt() недоступна
    // d2.fPrvt() недоступна

    DerivedDerived2 dd2;

    // dd2.fPubl() недоступна
    // dd2.fProt() недоступна
    // dd2.fPrvt() недоступна
}
```

Класс `Derived2` наследует классу `Base` со спецификатором `protected`. Это означает, что права доступа для унаследованных членов `Base` ограничиваются до уровня доступа `protected`. Следовательно, *объектам* класса `Derived2` недоступны члены класса `Base`, а из класса `DerivedDerived2` доступны те же члены класса `Base`, что и в классе `Derived2`, но со спецификаторами `protected`. Разумеется, *объектам* класса `DerivedDerived2` никакие члены класса `Base` не будут доступны.

Теперь рассмотрим наследование со спецификатором `private`. Спецификатор `private` закрывает все унаследованные члены от использования при дальнейшем наследовании, а также в объектах самого производного класса.

```
class Base {
public:
    void fPubl();
protected:
    void fProt();
private:
    void fPrvt();
};

class Derived3 : private Base {
public:
    Derived3() {
        fPubl(); // доступна
        fProt(); // доступна

        // fPrvt() недоступна
    }
};

class DerivedDerived3 : public Derived3 {
public:
    DerivedDerived3() {

        // fPubl() недоступна
        // fProt() недоступна
        // fPrvt() недоступна
    }
};

void main() {
    Base b;
    b.fPubl(); // доступна

    // b.fProt() недоступна
    // b.fPrvt() недоступна

    Derived3 d1;

    // d3.fPubl() недоступна
    // d3.fProt() недоступна
    // d3.fPrvt() недоступна
```

```
    DerivedDerived3 dd3;

    // dd3.fPubl() недоступна
    // dd3.fProt() недоступна
    // dd3.fPrvt() недоступна
}
```

В последнем примере класс `Derived3` наследует классу `Base` со спецификатором `private`. Это означает, члены, унаследованные из `Base`, будучи доступны в классе `Derived3` (со спецификаторами `private`), недоступны ни объектам класса `Derived3` (равно как и никаким другим объектам), ни в классе `DerivedDerived3`.

Обратите внимание, что если спецификатор явно не указан, действует спецификатор наследования доступа `private`, т. е. запись

```
class Derived3 : private Base {
    // ...
}
```

эквивалентна записи

```
class Derived3 : Base {
    // ...
}
```

Наследование как многократное использование интерфейса

По замечанию Эккеля, композиция применяется в тех случаях, когда необходимо использовать характеристики и возможности уже существующего класса, но не его интерфейс. При наследовании создается особая версия уже существующего класса, в основе которой — интерфейс, определяемый базовым классом [Eckel, 2000-2].

В этом смысле методы базового класса, объявленные со спецификатором `public`, могут трактоваться как элементы механизма доступа к интерфейсу. Поэтому не следует при определении базового класса стремиться охватить все варианты возможного использования класса в иерархии классов при наследовании. Удачный базовый класс определяет наиболее общие свойства, присущие большинству объектов предметной области, а тонкости абстрагирования отдельных групп объектов учитываются при конструировании производных классов. *Более подробный анализ наследования как механизма многократного использования интерфейса представлен в гл. 7.*

4.4. Реализация наследования: модели, отличные от C++

Поскольку заявленное во введении назначение книги заключается в анализе концепций и механизмов ООП как таковых, а не только в свете их реализации в языке C++, рассмотрим наиболее распространенные отличительные особенности реализации наследования, которые могут проявляться в других объектно-ориентированных языках.

Однокоренная иерархия классов

Если в C++ класс, не наследующий явно другому классу, является совершенно обособленной сущностью, то многие языки, относящиеся к разновидности "чистых" объектно-ориентированных языков, построены с использованием принципа однокоренной иерархии классов. Однокоренная иерархия предполагает, что все классы неявно унаследованы (прямо или опосредованно) от некоторого суперкласса (который чаще всего имеет имя `Object`). Класс `Object` определяет набор операций, общих для всех объектов. Обычно предполагается, что методы класса `Object` могут (а часто — и должны) замещаться в производных классах.

Рассмотрим пример на Java, представленный листингом 4.6.

Листинг 4.6. Все классы унаследованы от `Object` (пример на языке Java)

```
class X { // Неявно наследует классу Object
}

public class Test {
    public static void main( String[] args ) {
        X x = new X();
        String info = x.toString();
        System.out.println( info );
    }
}
```

Класс `X` не содержит собственных членов и явно не наследует никакому другому классу. Однако класс `X`, как и все классы Java, неявно наследует корневому классу общей иерархии `Object`. Это иллюстрирует обращение к методу `toString()`, определенному в классе `Object`, на объекте `x`. В результате исполнения программы будет напечатано что-нибудь вроде:

```
X@372a1a
```

Строка до знака @ является именем класса объектной переменной. После знака @ следует hash-код объекта, который может изменяться от запуска к запуску. Hash-коды объектов используются для хранения и поиска объектов в Hash-таблицах (перемешанных таблицах). *Подробнее о перемешанных таблицах см. в литературе по алгоритмам и в документации JDK.*

Читателю, возможно, будет небезынтересно сравнить код листинга 4.6 с эквивалентной по своей организации программой на языке C#, представленной листингом 4.7.

Листинг 4.7. Все классы унаследованы от `Object` (пример на языке C#)

```
using System;
class X { // Неявно наследует классу System.Object
}

public class Test {
    public static void Main( String[] args ) {
        X x = new X();
        String info = x.ToString();
        Console.WriteLine ( info );
    }
}
```

В результате исполнения программы будет напечатано имя типа объекта, на котором вызван метод `ToString()`:

```
X
```

Сокрытие имен

Как следует из примера, представленного в листинге 4.3, в C++ методы производного класса, одноименные методам базового класса, скрывают методы базового класса. Такая реализация механизма сокрытия имен не является обязательным требованием объектно-ориентированного подхода. Так, в языке Java, если в производном классе определен метод с именем, совпадающим с одним из перегруженных имен в базовом классе, то это имя не скрывает ни одну из версий в классах верхних уровней (листинг 4.8).

Листинг 4.8. В Java нет сокрытия имен при наследовании

```
class Base {
    public void g( int x ) {
        System.out.println( "void Base::g( int ) => x = " + x );
    }
```

```
    public void g( String s ){
        System.out.println( "void Base::g( String ) => s = " + s );
    }
}

class Derived extends Base {
    public void g( int x, int y ) {
         System.out.print( "void Derived::g( int, int ) => " );
         System.out.println( "x = " + x + " y = " + y );
    }
}

class NoHiding {
    public static void main( String[] args ) {
        Derived d = new Derived();

        d.g( 4, 5 );   // Верно: вызывается Derived.g( int, int )

        d.g( "test" ); // Верно: вызывается Base.g( String )

        d.g( 10 );     // Верно: вызывается Base.g( int )
    }
}
```

Сравните схему возможных вызовов, представленную на рис. 4.5, с иллюстрацией механизма сокрытия имен в C++, представленную на рис. 4.3.

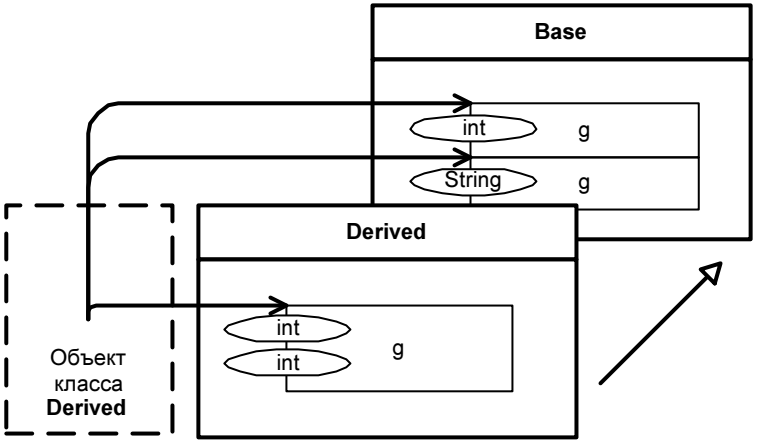


Рис. 4.5. В Java нет сокрытия имен при наследовании

Механизм разрешения вызовов, используемый языком C#, рассматривается в гл. 11.

Неизменные методы и классы

В процессе разработки иногда возникают ситуации, когда для дальнейшей предсказуемой работы с объектами некоторого класса нежелательно изменение реализации того или иного метода класса. В C++ по умолчанию методы класса не допускают замещения (для разрешения замещения требуется использовать спецификатор `virtual`). Определение невиртуального метода производного класса с именем, совпадающим с перегруженными именами невиртуальных методов базового класса, скрывает методы базового класса (см. пример в листинге 4.3). Это означает, что в обоих случаях речь идет о невозможности изменения поведения объекта базового класса.

Однако модели реализации механизма наследования могут отличаться в других языках программирования. Так, в Java все методы по умолчанию полиморфны, т. е. могут быть замещены в производных классах, а также нет сокрытия имен. Это означает, что в случае, когда разрешенное замещение или перегрузка противоречат намерениям разработчика базового класса, этот разработчик должен иметь возможность обеспечить неизменность метода. В языке Java для объявления неизменного метода используется специальный спецификатор `final`. Неизменный метод рассматривается компилятором как кандидат на реализацию в качестве встроенной функции, т. е. функцией, вызов которой может быть заменен в точке вызова копией тела функции.

В C# для реализации эквивалентной семантики объявления метода используется служебное слово `sealed`.

Аналогично неизменным методам разработчик может захотеть предотвратить наследование от классов, структура и реализация которых такова, что, по мнению разработчика, дальнейшее наследование от этих классов нежелательно. В этом случае модификатором `final` (в случае Java) или `sealed` (в случае C#) снабжается определение класса. Компилятор контролирует соблюдение неизменности класса (рис. 4.6). При этом многие разработчики отмечают, что следует быть осторожным при создании неизменных классов (особенно включаемых в некоторую библиотеку). Разработчик может не предвидеть всех возможных вариантов использования класса и, объявляя его неизменным, затрудняет его повторное использование в клиентском коде. По замечанию Страуструпа, "решение, каким образом класс должен использоваться, должно быть сознательным (осознанным и проектировщиком, и пользователем [разрядка моя — Е. П.])" [Stroustrup, 2000].

Вообще говоря, классы, плохо пригодные для последующего наследования, обычно относятся к категории типов, которые Страуструп называл *конкретными* типами. Примером конкретного типа может служить класс `ModInteger` (см. листинг 3.14). Конкретные типы характеризуются тем, что их можно по-

нять в отрыве от других классов. Связи конкретных типов с другими классами минимальны. Конкретный тип точно соответствует моделируемой сущности предметной области, и потому он обычно не слишком полезен как основа для конструирования новых типов. Язык C++, о котором справедливо сказать, что он предъявляет довольно серьезные требования к квалификации программиста, оставляет на усмотрение программиста решение вопроса, можно ли и нужно ли использовать некоторый тип в качестве базового: нет специальных средств языка, ограничивающих возможности программиста. Философия C++ предполагает среди прочего, что программист должен понимать назначение и стратегии реализации кода, оперирующего с пользовательскими типами, и самостоятельно принимать осмысленные решения о том, является ли тип конкретным или воплощает более общую абстракцию (рис. 4.6).

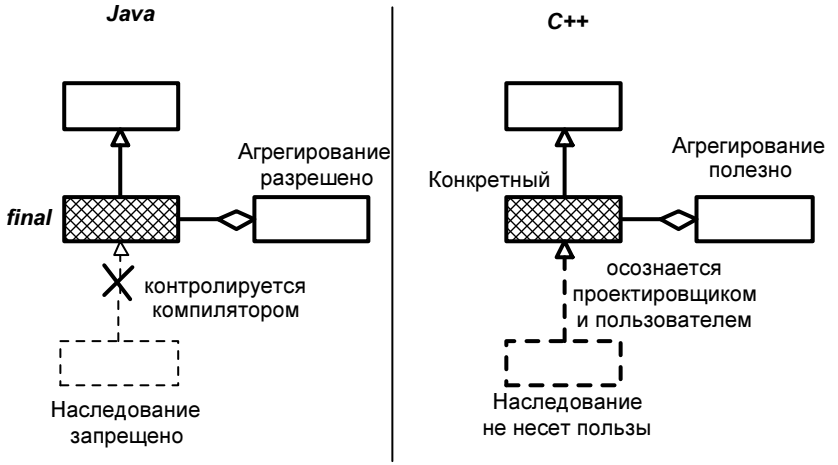


Рис. 4.6. Неизменные классы и конкретные типы

С другой стороны, существуют типы, предназначенные исключительно для наследования, а не для создания объектов. *Подробности проектирования таких типов см. в гл. 5.*

4.5. Вопросы и упражнения

1. Рассмотрите определения классов:

```
class A {
protected:
    int ax;
```

```
public:
    void af();
};

class B : A {
    int bx;
protected:
    int by;
public:
    int bf();
};

class C : public B {
};
```

Какие члены классов A и B будут доступны *объектам* классов B и C?

2. Какие ошибки будут выявлены при компиляции данной программы? Как их исправить?

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

class Base {
    int x;
public:
    Base( int arg ) {
        x = arg;
    }
    void f() {
        cout << "Base::f() => x = " << x << endl;
    }
};

class Derived : public Base {
public:
    void f( int x ) {
        cout << "Derived::f() => x = " << x << endl;
    }
};

void main() {
    Derived dObj;
    dObj.f();
    dObj.f( 4 );
}
```

3. Для приведенного ниже фрагмента программы запишите в местах, указанных комментариями, примеры обращений к соответствующим функциям. Проверьте свое решение на компьютере и обсудите его с преподавателем.

```
class A {
public:
    void f( int x ) {
        // ...
    }
};

class B : A {
    void f( int x ) {
        // Здесь запишите обращение к функции void f( int ) -
        // члену класса A или объясните, почему этого нельзя сделать
        // ...
    }
public:
    void f() {
        // ...
    }
};

class C : public B {
public:
    void f( int x ) {
        // Здесь запишите обращение к функции void f( int ) -
        // члену класса B или объясните, почему этого нельзя сделать
        // ...
    }
};

void main() {
    A *c1 = new C;
    // Используя указатель c1, запишите обращение к функции
    // void f( int ) — члену класса A или объясните,
    // почему этого нельзя сделать
    // ...
    // Используя указатель c1, запишите обращение к функции
    // void f( int ) — члену класса B или объясните,
    // почему этого нельзя сделать
    // ...
    delete c1;
```

```
    C c2;
    // На объекте c2 запишите обращение к функции
    // void f( int ) — члену класса A или объясните,
    // почему этого нельзя сделать
    // ...
    // На объекте c2 запишите обращение к функции
    // void f() — члену класса B или объясните,
    // почему этого нельзя сделать
    // ...
}
```

4. Что будет напечатано в результате исполнения следующей программы?

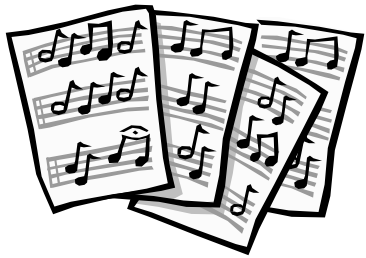
```
#include <iostream>
using namespace std;

class Base {
    int x;
public:
    Base( int arg = 10 ) {
        x = arg;
        cout << "Base constructed, arg = " << arg;
        cout << endl;
    }
    ~Base() {
        cout << "Base destroyed" << endl;
    }
};

class Derived : private Base {
public:
    Derived() {
        cout << "Derived constructed" << endl;
    }
    ~Derived() {
        cout << "Derived destroyed" << endl;
    }
};

Base global( 0 );

void main() {
    Derived dObj;
}
```

5. Проанализируйте приведенную ниже программу и, если она корректна, напишите, что будет напечатано в результате ее исполнения. Если, по Вашему мнению, в результате трансляции будут обнаружены ошибки, укажите какие.

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

class A {
public:
    void print( int x ) {
        cout << " From A::print(): x=" << x << endl;
    }
};

class B : public A {
public:
    void print( int x ) {
        cout << "From B::print(): x=" << x << endl;
    }
};

void main() {
    A a;
    a.print( 5 );

    A *b = new B;
    b->print( 5 );
}
```

6. Разработайте реалистичную версию программы для обработки и отображения сведений о студентах и аспирантах. Предложите структуру информации, которая могла бы обрабатываться подобной программой. Реализуйте в Вашей программе модуль сбора и обработки какой-либо статистической информации, вычисляемой на основе содержащихся в базе студентов сведений (например, средний балл группы студентов, процент успевающих студентов и т. п.).

ГЛАВА 5



Объекты классов и полиморфизм

В данной главе подробно рассматриваются особенности управления объектами классов, входящих в иерархии наследования. Особое внимание уделяется вариантам наследования, при котором определение производного класса видоизменяет реализацию одноименных методов базового класса, обеспечивая полиморфное поведение объектов.

5.1. Понятие о статическом и динамическом связывании

Понятие динамического связывания может поначалу вызывать трудности у тех студентов, которые начинают изучение программирования с освоения процедурной парадигмы и структурного программирования. Связыванием обычно называют привязку тела функции к месту ее вызова. В процедурных языках задача связывания решается на этапе компиляции (если определение функции находится в том же модуле, что и обращение к ней) или, самое позднее, на этапе компоновки (если определение функции находится в другом модуле). Поэтому этап компоновки часто называют редактированием связей. Соответственно, на момент исполнения программы каждый вызов функции в точности "привязан" к ее определению. Собственно, это и есть единственная модель связывания, используемая в процедурных программах (см. рис. 1.13). В терминах, используемых в данной главе, такое связывание называют ранним, или статическим, связыванием.

Иначе обстоит дело в связи с управлением объектами. Как было проиллюстрировано в ряде примеров, рассмотренных в *гл. 2* и *4*, переменная-указатель на объект базового типа может в действительности быть связана с объектом производного типа, что делает в общем случае невозможным установление действительного экземпляра метода, который нужно вызвать, в момент

трансляции или компоновки программы. Рассмотрим эту проблему более детально.

Статическое связывание. Установление типов объектов во время компиляции

Для начала вернемся к примеру программы, представленному в *гл. 4*, листинг 4.2. Рассмотрим функцию `CallPrint()`:

```
void CallPrint( Student *stud ) {
    stud->Print();
}
```

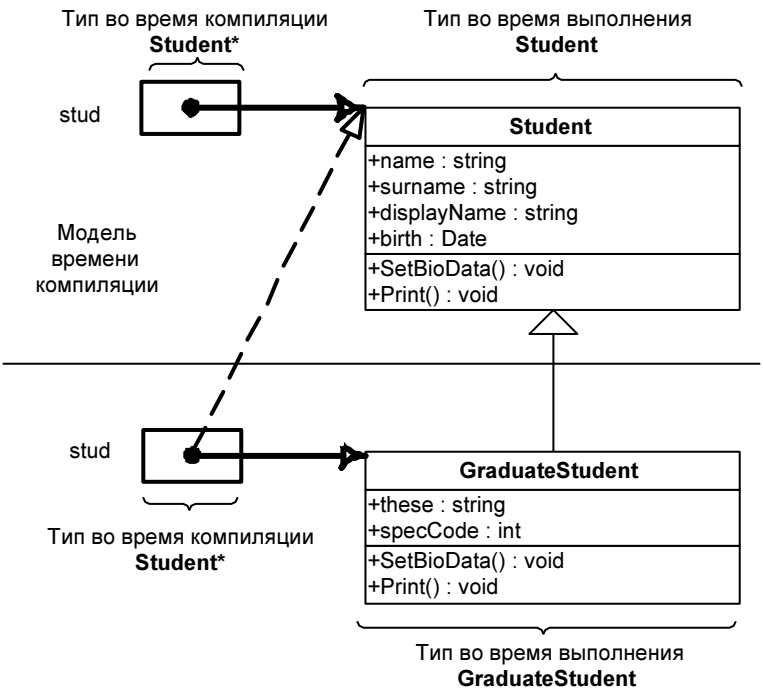


Рис. 5.1. Статическое (раннее) связывание

На рис. 5.1 анализируемая ситуация представлена графически. Во время компиляции невозможно установить, на объект какого типа будет указывать указатель `stud` во время выполнения: на объект типа `Student` или на объект производного типа (например, `GraduateStudent`). Единственное, что гарантируется — так это то, что указатель, передаваемый функции `CallPrint()` в качестве аргумента, может быть безопасно и корректно преобразован к указателю

на Student. Таким образом, обработка объектов типа GraduateStudent возможна, но только в рамках той функциональности, которую обеспечивает класс Student (это показано на рисунке перечеркнутыми членами класса GraduateStudent). Модель времени компиляции в этом случае является моделью времени выполнения. Как было сказано выше, такой вариант связывания называют ранним (или статическим) связыванием, т. е. связыванием, оперирующим типами, известными во время компиляции. Такие типы иногда называют статическими — отсюда и название модели связывания кода и данных. Языки, не являющиеся объектно-ориентированными, обычно предоставляют только такой механизм связывания.

Подведем промежуточный итог. Даже если в действительности указатель stud связан с объектом типа GraduateStudent, в соответствии с механизмом статического связывания вызываться всегда будет версия функции Print(), объявленная в классе Student (что и демонстрировала приведенная в листинге 4.2 программа).

Конечно, можно попытаться "уговорить" компилятор, воспользовавшись явным преобразованием типа static_cast, например:

```
void CallPrint( Student *stud ) {
    static_cast <GraduateStudent*>( stud )->Print();
}
```

Однако такой вариант будет работоспособен только в том случае, когда на входе функции CallPrint() действительно окажется указатель на GraduateStudent. В остальных случаях результат вывода функции GraduateStudent::Print() будет непредсказуем. Такого рода преобразования называют небезопасными (рис. 5.2), т. к. корректность этих преобразований невозможно установить во время трансляции.

Более того, ошибки, являющиеся следствием таких преобразований, не всегда очевидно проявляются при выполнении программы.

В качестве примера рассмотрим тестирующую программу следующего содержания (листинг 5.1).

Листинг 5.1. Небезопасное статическое преобразование типов

```
#include "Students.h"

void CallPrint( Student *stud ) {
    static_cast <GraduateStudent*>( stud )->Print();
}

void main() {
    Student s1;
```

```
    s1.SetBioData( "John", "Smith", 1, 1, 1975 );
    s1.Print();  // Student::Print()

    GraduateStudent gs1;
    gs1.SetBioData( "Jeffrey", "Writer", 14, 7, 1963, 2201, "Software" );
    gs1.Print(); // GraduateStudent::Print()

    CallPrint( &gs1 ); // Вызывается GraduateStudent::Print()
                       // за счет преобразования,
                       // достигнутого грубой силой:
                       // в данном случае результат корректен

    CallPrint( &s1 );  // Вызывается GraduateStudent::Print()
                       // за счет преобразования,
                       // достигнутого грубой силой:
                       // непредсказуемый вывод
}
```

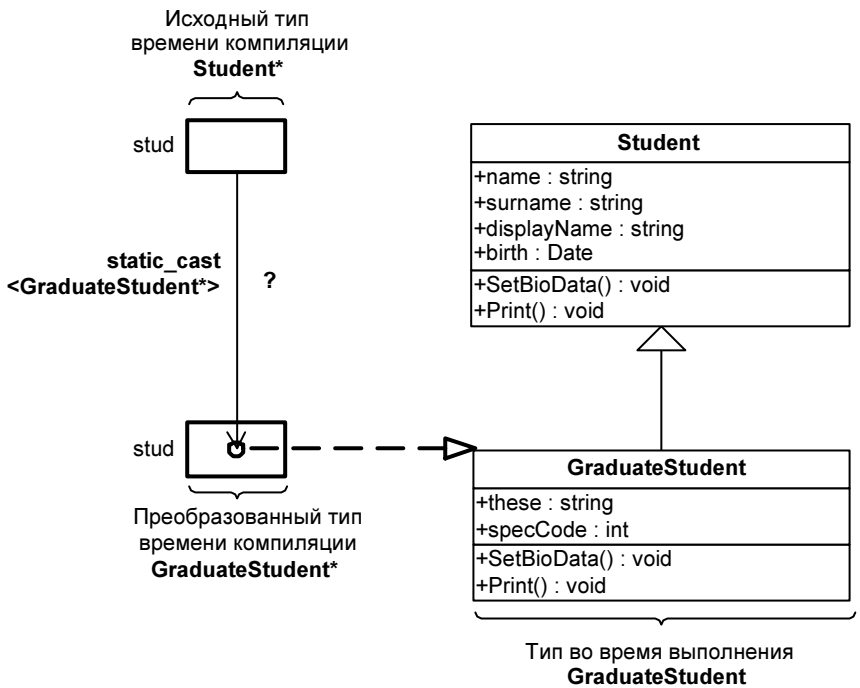


Рис. 5.2. Небезопасное статическое преобразование типа

В первом случае при вызове функции `CallPrint()` указатель `stud` действительно связан с объектом типа `GraduateStudent`, и результат соответствует

нашим ожиданиям, во втором случае указатель `stud` связан с объектом типа `Student`, но за счет статического преобразования также происходит обращение к функции `GraduateStudent::Print()`, которая (вызывая функцию базового класса) корректно печатает содержимое поля `displayName`, а последующий вывод непредсказуем, т. к. объект типа `Student` не содержит полей `specCode` и `these`.

Динамическое связывание. Установление типов объектов во время выполнения

Основная проблема статического связывания заключается в том, что функция, аргументом которой является указатель или ссылка на объект базового класса, способна принять в качестве аргумента указатель или ссылку на объект производного класса, но это не позволяет обратиться к членам производного класса. Выясняется, что модель связывания, при которой все решения относительно типов объектов принимаются на стадии компиляции, недостаточна для реализации идей, заложенных в концепцию наследования.

Решение проблемы заключается в так называемом динамическом, или позднем, связывании, когда определение конкретного экземпляра функции-члена производного класса может происходить не во время компиляции, а во время выполнения. Динамическое связывание часто называют среди трех основных понятий объектно-ориентированного программирования наряду с абстрагированием и инкапсуляцией.

Если язык поддерживает позднее связывание, он должен предоставлять некоторый механизм, позволяющий определить тип объекта во время выполнения и вызвать подходящий метод в зависимости от типа объекта. В языке C++ подобный механизм основывается на использовании виртуальных функций. Объекты, определения классов которых допускают динамическое связывание, называют объектами, допускающими полиморфное поведение, или полиморфными объектами.

5.2. Виртуальные функции — механизм реализации полиморфизма в C++

Итак, механизм реализации динамического связывания основан на использовании виртуальных (перезаписываемых, замещающих) функций. Для того чтобы объявить для некоторого метода необходимость реализовать применительно к этому методу модель позднего связывания, в C++ используется ключевое слово `virtual`, указываемое при объявлении функции. Если функция объявлена виртуальной, то компилятор и загрузчик гарантируют уста-

новление правильного соответствия между объектами и функциями, вызываемыми на этих объектах.

Общие сведения

Представленные в этом разделе примеры иллюстрируют основные правила использования виртуальных функций.

Пример со студентами и аспирантами

Внесем в определение класса `Student`, представленное в листинге 4.1, всего лишь одно изменение, объявив функцию `Print()` виртуальной (листинг 5.2). Теперь `Print()` выступает в качестве интерфейса как к функции `Print()`, определенной в базовом классе, так и к функции `Print()`, определенной в производном классе.

Листинг 5.2. Определения классов `Student` и `GraduateStudent`, содержащих виртуальную функцию `Print()` (версия 2), и тестирующая программа

```
/*
 * Students.h
 *
 * Определение классов в связи с представлением
 * информации о студентах и аспирантах
 * (версия 2: функция Print() объявлена виртуальной)
 */

#ifndef _Students_h_
#define _Students_h_

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

struct Date {
    int day;
    int month;
    int year;
};

class Student {
public:

    // см. листинг 4.1...
```

```
    virtual void Print(){
        cout << "From Student::Print() => Name: ";
        cout << displayName << endl;
    }

    // ...
};

class GraduateStudent : public Student {
public:
    // ...

    void Print() { // virtual подразумевается
        Student::Print();
        cout << "\tFrom GraduateStudent::Print()";
        cout << " => SpecCode:";
        cout << specCode << endl;
        cout << "\tThese: " << these << endl;
    }
};

#endif

/*
 * Students.cpp
 *
 * Функции для тестирования особенностей обработки
 * объектов классов Student и GraduateStudent (в версии 2)
 */

#include "Students.h"

typedef Student *PStudent; // Тип: указатель на Student

void CallPrint( Student *stud ) {
    stud->Print();
}

// Основная программа
void main() {

    // Создание и использование объекта типа Student
    Student s1;
    s1.SetBioData( "John", "Smith", 1, 1, 1975 );
    s1.Print();  // Вызывается Student::Print()

    // Создание и использование объекта типа GraduateStudent
    GraduateStudent gs1;
```

```
    gs1.SetBioData( "Jeffrey", "Writer", 14, 7, 1963,
        2201, "Applied .Net Framework Programming" );
    gs1.Print(); // Вызывается GraduateStudent::Print()

    // Указатель на объект типа Student
    // получает адрес объекта типа GraduateStudent
    Student *ptrGs1 = &gs1;
    ptrGs1->Print(); // Вызывается GraduateStudent::Print(),
                     // поскольку Print() — виртуальная:
                     // работает позднее связывание

    // Создание еще одного объекта типа GraduateStudent
    GraduateStudent gs2;
    gs2.SetBioData( "Bjarne", "Strawberry", 25, 12, 1950,
        2204, "The C++ Programming Language" );

    // Создание массива указателей на объекты
    PStudent *all = new PStudent[ 3 ];

                     // Связывание...
    all[ 0 ] = &s1;  //  с объектом "John Smith"
    all[ 1 ] = &gs1; //  с объектом "Jeffrey Writer"
    all[ 2 ] = &gs2; //  с объектом "Bjarne Strawberry"

    // Печать информации обо всех студентах
    for( int ix = 0; ix < 3; ix++ ) {
        CallPrint( all[ ix ] );
            // ix = 0: Вызывается Student::Print()
            // ix = 1: Вызывается GraduateStudent::Print()
            // ix = 2: Вызывается GraduateStudent::Print()
    }

    delete [] all;
}
```

Функция `Print()`, объявленная в классе `GraduateStudent`, также считается виртуальной (независимо от того, содержится ли в ее объявлении ключевое слово `virtual`). При этом функция, объявленная в производном классе (с тем же набором параметров), замещает (overrides) виртуальную функцию-член базового класса.

Как видно из листинга 5.2, тестирующая программа отличается от программы, представленной в листинге 4.2, только содержанием комментариев. Комментарии в тексте программы позволяют разобраться в том, почему на этот раз вывод будет иметь следующий вид:

```
From Student::Print() => Name: John Smith
From Student::Print() => Name: Jeffrey Writer
    From GraduateStudent::Print() => SpecCode: 2201
    These: Applied .Net Framework Programming
From Student::Print() => Name: Jeffrey Writer
    From GraduateStudent::Print() => SpecCode: 2201
    These: Applied .Net Framework Programming
From Student::Print() => Name: John Smith
From Student::Print() => Name: Jeffrey Writer
    From GraduateStudent::Print() => SpecCode: 2201
    These: Applied .Net Framework Programming
From Student::Print() => Name: Bjarne Strawberry
    From GraduateStudent::Print() => SpecCode: 2204
    These: The C++ Programming Language
```

Полужирным начертанием здесь выделены те фрагменты выходного текста, которые отличаются от вывода программы, рассмотренной в *гл. 4* (см. листинг 4.2).

Пример с геометрическими фигурами

Иногда студенты, рассмотрев пример, подобный предыдущему, задают вопрос о том, так ли уж невозможно установить тип объектов на этапе компиляции. Нельзя ли при разработке транслятора и компоновщика как-нибудь "исхитриться", применив более продвинутые методы анализа исходного текста и, может быть, объектного кода. Ведь в только что рассмотренном примере (листинг 5.2) при желании можно было бы установить, с объектом какого типа связываются указатели `ptrGs1`, `all[1]` и `all[2]`, в процессе трансляции.

Здесь мы рассмотрим другой пример, показывающий, что даже если все определения находятся в одном модуле, существуют случаи, когда определенно нет никакой возможности установить типы всех объектов до исполнения программы. Листинг 5.3 содержит модельную реализацию классов, соответствующую фрагменту иерархии классов на рис. 1.17. Под модельной реализацией здесь понимается то, что предметное содержание функций `Draw()` и `Erase()` нас сейчас не очень интересует, а разобраться с деталями реализации полиморфизма мы сможем, просто анализируя информацию, выводимую этими функциями на консоль.

Листинг 5.3. Виртуальные функции и полиморфизм

```
/*
 * Shapes.cpp
 *
```

```
 * Пример с геометрическими фигурами для
 * анализа реализации концепции полиморфизма в C++
 *
 */
#include <cstdlib>     // для srand() и rand()
#include <ctime>       // для time()

#include <iostream>
using namespace std;

class Shape {
public:
    virtual void Draw() {
        cout << "Shape::Draw()" << endl;;
    }
    virtual void Erase() {
        cout << "Shape::Erase()" << endl;
    }
};

typedef Shape *PShape;

class Circle : public Shape {
public:
    void Draw() {
      cout << "Circle::Draw()" << endl;
    }
    void Erase() {
      cout << "Circle::Erase()" << endl;
    }
};

class Square : public Shape {
public:
    void Draw() {
        cout << "Square::Draw()" << endl;
    }
    void Erase() {
        cout << "Square::Erase()" << endl;
    }
};

class Triangle : public Shape {
public:
    void Draw() {
        cout << "Triangle::Draw()" << endl;
    }
```

```
    void Erase() {
        cout << "Triangle::Erase()" << endl;
    }

    virtual void Rotate90() {
        cout << "Triangle::Rotate90()" << endl;
    }
};

// Класс-генератор объектов-геометрических фигур
class Shapes {
public:
    static ShapesInit() {
        // Инициализация стандартного генератора случайных
        // чисел
        srand( ( unsigned )time( NULL ) );
    }

    // Создание одной из трех фигур случайным образом
    static void RandomShape( PShape& s ) {
        switch( (rand() % 3) ) {
            default:
            case 0 : s = new Circle;
                     break;
            case 1 : s = new Square;
                     break;
            case 2 : s = new Triangle;
                     break;
        }
    }
};

// Тестирующая программа
void main() {

    PShape s[ 5 ]; // Массив указателей на геом. фигуры
    int i;         // Счетчик при переборе объектов из s

    // Заполнение массива указателей на геометрические фигуры
    Shapes::ShapesInit();
    for( i = 0; i < 5; i++ ) {

        // Фигура генерируется случайным образом!
        Shapes::RandomShape( s[ i ] );
    }
```

```
    for( i = 0; i < 5; i++ ) {

        // Вызов виртуальных методов
        s[ i ]->Draw();
        s[ i ]->Erase();
    }

    // Освобождение памяти, занятой объектами
    for( i = 0; i < 5; i++ ) {
        delete s[ i ];
    }
}
```

Как явствует из определения функции `RandomShape()`, объекты создаются во время выполнения случайным образом, так что никакой практической возможности установить тип созданных объектов-геометрических фигур на более ранних стадиях нет.

Вывод этой программы будет отличаться от запуска к запуску и может быть, например, таким:

```
Circle::Draw()
Circle::Erase()
Square::Draw()
Square::Erase()
Square::Draw()
Square::Erase()
Triangle::Draw()
Triangle::Erase()
Circle::Draw()
Circle::Erase()
```

Главное, что иллюстрирует этот вывод — это то, что благодаря позднему связыванию виртуальных функций, вызываются функции `Draw()` и `Erase()`, объявленные в соответствующих классах `Circle`, `Square` и `Triangle`, а не "абстрактные" версии из класса `Shape` (и это несмотря на то, что все указатели в массиве `s` имеют тип `Shape*`).

Требования к сигнатуре виртуальной функции

Предшествующие примеры показывают: для того чтобы объявление виртуальной функции в базовом классе могло быть использовано в качестве интерфейса к функциям, определенным в производных классах, виртуальная функция-член производного класса должна иметь ту же сигнатуру, что и функция, объявленная в базовом классе.

Единственное "послабление", разрешенное правилами языка C++, заключается в том, что виртуальная функция-член производного класса может возвращать указатель на производный класс, в то время как версия, объявленная в базовом классе, возвращает указатель на базовый класс. То же справедливо для ссылок. Это связано с тем, что в ряде случаев нужно иметь возможность создания объекта, точный тип которого во время трансляции неизвестен, т. е. написать функцию, являющуюся своего рода "виртуальным конструктором" (см. [Stroustrup, 2000, разд. 15.6.2]).

Таблица виртуальных функций — основной элемент механизма реализации позднего связывания

> Тем, кто внимает нашему рассказу единственно из желания узнать, чем он завершится... эту главку позволительно вовсе пропустить. Никакого ущерба для стройности повествования от этого не произойдет.
>
> *Борис Акунин. "Пелагия и белый бульдог"*

Для изложения подробностей реализации механизма позднего связывания продолжим рассмотрение примера с геометрическими фигурами.

Таблица виртуальных функций

Стандартное решение для обеспечения динамического связывания, используемое разработчиками компиляторов, заключается в следующем. Для каждого класса, допускающего полиморфное поведение своих объектов (т. е. содержащего хотя бы одну виртуальную функцию), создается специальная таблица, содержащая адреса виртуальных функций, объявленных в этом классе (картина для классов `Shape` и `Triangle` представлена на рис. 5.3). В определение класса добавляется скрытый член класса — указатель на таблицу виртуальных функций `VTable`. Таким образом, размер памяти, занимаемой полиморфным объектом, увеличится ровно на один указатель по отношению к аналогичному объекту класса, не содержащего виртуальные функции.

Адреса одноименных виртуальных функций разных классов, находящихся в отношении наследования, помещаются в ячейки таблицы `VTable` с одинаковыми индексами.

Если какая-либо виртуальная функция не перезаписывается в производном классе (это означает, что для объектов этого класса должна использоваться версия базового класса), в таблицу виртуальных функций помещается адрес соответствующей функции базового класса.

В случае одиночного наследования накладные расходы, связанные с реализацией полиморфизма, составляют один указатель `vptr` для каждого объекта

класса, содержащего виртуальные функции, и одна таблица виртуальных функций для каждого такого класса.

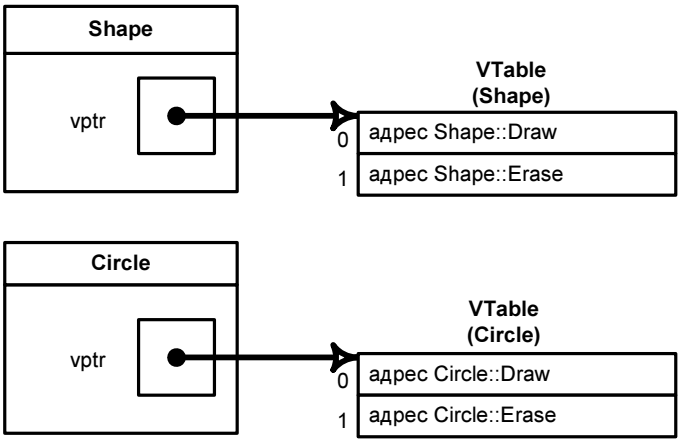


Рис. 5.3. Таблицы виртуальных функций

Выбор корректной версии виртуальной функции

Процесс выбора корректной версии виртуальной функции представлен на рис. 5.4.

Рассмотрим пример генерации кода для вызова виртуальной функции `Erase()`.

Имея в качестве `s[i]` указатель на объект типа `Shape` (потому что все элементы массива являются указателями на `Shape`), компилятор обращается к скрытому элементу `vptr`, который существует у любого класса, содержащего виртуальные функции (обычно этот элемент размещается в начале области памяти, занимаемой объектом). Указатель `vptr` содержит адрес начала таблицы виртуальных функций `VTable`. Далее имя вызываемой функции преобразуется в индекс таблицы. Поскольку адреса виртуальных функций базового и производного класса хранятся в таблице в одинаковом порядке, то элемент таблицы с индексом 0 всегда содержит адрес функции `Draw()`, элемент таблицы с индексом 1 всегда содержит адрес функции `Erase()` и т. д. Таким образом, независимо от того, какого типа будет в действительности объект, связанный с указателем `s[i]`, для вызова функции `Erase()` нужно извлечь ее адрес из элемента с индексом 1 таблицы `VTable`, т. е. адрес вызываемой функции извлекается из ячейки памяти с адресом `vptr+1`.

Итак, если конфигурация объектов соответствует картинке на рис. 5.4, в процессе выполнения программы для объекта, адресуемого указателем `s[0]`,

будет вызвана функция по адресу `*(s[0].vptr+1)` (т. е. функция `Circle::Erase()`); для объекта, адресуемого указателем `s[1]`, будет вызвана функция по адресу `*(s[1].vptr+1)` (т. е. функция `Square::Erase()`); для объекта, адресуемого указателем `s[3]`, будет вызвана функция по адресу `*(s[3].vptr+1)` (т. е. функция `Triangle::Erase()`).

Процесс вычисления адреса функции, таким образом, осуществляется во время выполнения программы — это и называется поздним связыванием.

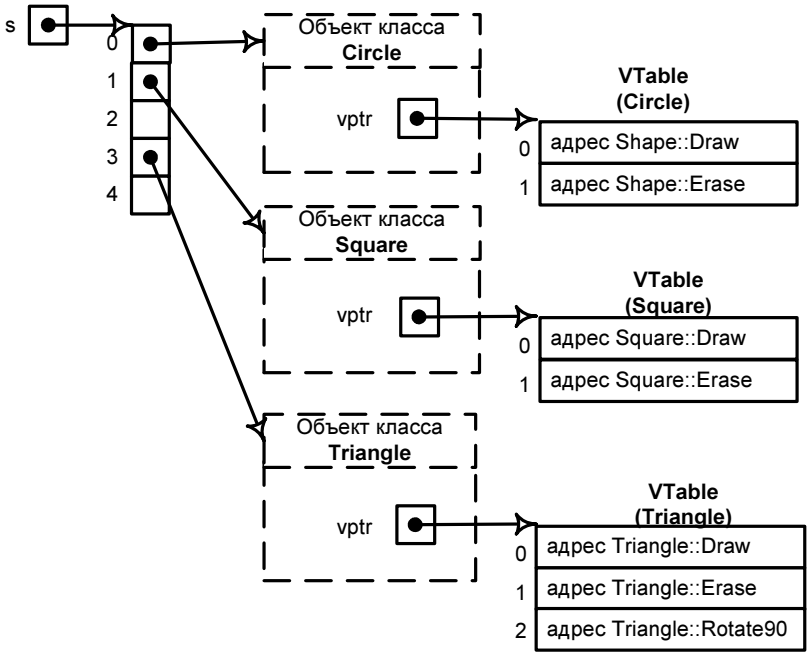


Рис. 5.4. Выбор правильной виртуальной функции

Инициализация указателя на таблицу виртуальных функций

Для того чтобы механизм, описанный в предыдущем разделе, работал корректно, нужно обеспечить инициализацию указателя `vptr` таким образом, чтобы он действительно указывал на корректную версию `VTable`. Как известно, задачу инициализации объекта класса решает конструктор. Когда вызывается конструктор, тип объекта известен, поэтому имеется возможность ориентировать указатель `vptr` на соответствующую типу объекта таблицу виртуальных функций.

Однако ни в одном классе из иерархии классов для представления геометрических фигур нет явного конструктора. Напомним, что если конструктор по умолчанию отсутствует, и не объявлено ни одного явного конструктора, конструктор по умолчанию генерируется автоматически *(см. разд. 3.4)*.

Теперь смысл автоматической генерации конструктора по умолчанию становится очевидным — без этого, в частности, невозможно было бы гарантировать правильную инициализацию `vptr`.

Дополнительно, следует отметить, что при вызове виртуальной функции из конструктора всегда будет вызываться локальная версия функции (т. е. версия, объявленная в данном классе). Это означает, что механизм полиморфных вызовов не работает в пределах конструктора. Это связано с тем, что полиморфный вызов обычно предполагает вызов версии функции, объявленной в производном классе, инициализация которого, возможно, еще не состоялась, а значит — и не инициализирована его таблица виртуальных функций. Кроме того, окончательное значение указатель `vprt` принимает после того, как закончит работу конструктор самого нижнего класса в иерархии классов. Это означает, что нет возможности полагаться на текущее значение указателя `vptr` [Eckel, 2000-1].

Простой пример для иллюстрации затрат памяти на реализацию полиморфизма

Для того чтобы проиллюстрировать, каким образом наличие виртуальных функций влияет на размер памяти, занимаемой объектом, рассмотрим пару несложных примеров. Сначала определим класс, в котором нет полиморфных функций (листинг 5.4).

Листинг 5.4. Объект класса, не содержащего виртуальные функции

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

class Class1 {
    int x;

public:

    Class1( int _x = 0 ) {
        if( _x < 0 ) {
            cout << "Incorrect initial value " << _x;
            cout << ".  Initialized by zero." << endl;
            x = 0;
            return;
        }
```

```
        x = _x;
    }

    void SetX( int _x = 0 ) {
        if( _x < 0 ) {
            cout << "Incorrect set value " << _x;
            cout << ". Zero restored." << endl;
            x = 0;
            return;
        }
        x = _x;
    }

    int GetX() { return x; }

    void PrintX() {
        cout << "x = " << x << endl;
    }
};

void main() {
    Class1 obj1( -10 );
    cout << "Size of obj1  : " << sizeof( obj1 ) << endl;

    obj1.PrintX();
    obj1.SetX( -15 );
    obj1.PrintX();
    obj1.SetX( 15 );
    obj1.PrintX();
    cout << endl;
}
```

Этот простой класс выражает концепцию неотрицательного целого числа, контролируемого на корректность инициализации. Используя методы, составляющие открытый интерфейс класса, невозможно сохранить в объекте класса отрицательные значения.

Исполнение приведенной в листинге 5.4 программы приведет к выводу на консоль следующих сообщений:

```
Incorrect initial value -10. Initialized by zero.
Size of obj1: 4
x = 0
Incorrect set value -15. Zero restored.
x = 0
x = 15
```

Вторая строка содержит информацию о размере объекта, полученную благодаря использованию в тексте программы операции `sizeof`. Действительно, определение класса `Class1` содержит всего одну переменную-член класса. Эта переменная имеет тип `int`, экземпляры которого занимают в оперативной памяти 4 байта (программа исполнялась на ЭВМ 32-разрядной архитектуры).

Теперь изменим определение класса для представления контролируемых неотрицательных чисел таким образом, чтобы он допускал полиморфное поведение своих объектов (например, так, как это показано в листинге 5.5).

Листинг 5.5. Объекты классов, содержащих виртуальные функции

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

class Class2 {
    int x;

public:

    Class2( int _x = 0 ) {
        if( _x < 0 ) {
            cout << "Incorrect initial value " << _x;
            cout << ". Initialized by zero." << endl;
            x = 0;
            return;
        }
        x = _x;
    }

    virtual void SetX( int _x = 0 ) {
        if(_x < 0) {
            cout << "Incorrect set value " << _x;
            cout << ".  Zero restored." << endl;
            x = 0;
            return;
        }
        x = _x;
    }

    int GetX() { return x; }

    virtual void PrintX() {
        cout << "x = " << x << endl;
    }
};
```

```
class Derived2 : public Class2 {
public:
    Derived2( int _x = 0 ) : Class2( _x ) {};

    void SetX( int _x = 0 ) {
        if( _x < 0 ) {
            cout << "Incorrect set value " << _x;
            cout << ". Old value restored." << endl;
            return;
        }

        Class2::SetX( _x );
    }

    void PrintX() {
        cout << "x = "<< hex << GetX() << endl;
        cout << dec;
    }
};

void main() {
    Class2 *pObj1 = new Class2( 10 );
    cout << "Size of obj1  : " << sizeof( *pObj1 ) << endl;

    pObj1->PrintX();
    pObj1->SetX( -15 );
    pObj1->PrintX();
    pObj1->SetX( 15 );
    pObj1->PrintX();
    cout << endl;

    Derived2 *pObj2 = new Derived2( 100 );
    cout << "Size of obj2  : " << sizeof( *pObj2 ) << endl;
    pObj2->PrintX();
    pObj2->SetX( -10 );
    pObj2->PrintX();
    pObj2->SetX( 10 );
    pObj2->PrintX();
    cout << endl;
}
```

Определение класса `Class2` в листинге 5.5 идентично объявлению класса `Class1` в листинге 5.4 за исключением того, что два метода (`SetX()` и `PrintX()`) объявлены виртуальными, т. е. допускается их замещение при определении производных от `Class2` классов. Действительно, реализации этих методов отличаются в классе `Derived2`, сконструированном на базе класса `Class2`.

Так, при передаче некорректного аргумента методу `Derived2::SetX()` сохраняется прежнее значение `x` (в отличие от версии базового класса, в которой переменная `x` обнуляется). Печать значения переменной `x` осуществляется в шестнадцатеричном формате (в отличие от печати в формате по умолчанию в версии базового класса).

Функция `main()` содержит простой тест двух этих классов. Вывод программы, представленной в листинге 5.5, выглядит следующим образом:

```
Size of obj1: 8
x = 10
Incorrect set value -15. Zero restored.
x = 0
x = 15

Size of obj2: 8
x = 64
Incorrect set value -10. Old value restored.
x = 64
x = a
```

С особенностями обработки данных читатель вполне может разобраться самостоятельно. Мы же остановимся на двух моментах, связанных с хранением объектов в памяти:

1. Размер объекта, адресуемого указателем `pObj1`, 8 байтов, т. е. на 4 байта больше объекта `obj1` из примера в листинге 5.4. Эти "лишние" 4 байта и соответствуют скрытому указателю на таблицу виртуальных функций (единственное отличие класса `Class2` от класса `Class1` заключается в том, что методы класса `Class2` являются виртуальными).
2. Размер объекта, адресуемого указателем `pObj2`, тоже 8 байтов. Они складываются из четырех байтов, занимаемых закрытым членом `x`, унаследованным от класса `Class2` (и недоступным объектам класса `Derived2`!), и четырех байтов, занимаемых указателем на таблицу виртуальных функций класса `Derived2`. Итак, объект производного класса содержит в себе в качестве подобъекта все переменные-члены базового класса (в данном случае, одну переменную `x` типа `int`).

Чисто виртуальные функции и абстрактные классы

В ряде случаев уровень абстрагирования, представляемый базовым классом, не подразумевает какую-либо практическую реализацию некоторых (или всех) методов класса. В этом случае базовый класс рассматривается как вы-

разитель некоторой абстрактной концепции, предоставляя не реализацию методов, а общий интерфейс, подходящий для работы с элементами такого типа.

Так, вряд ли можно придумать какую-нибудь конкретную реализацию методов `Draw()` и `Erase()` в классе `Shape`, хотя сам по себе этот класс весьма полезен: он позволяет рассматривать все производные классы как его разновидности. При этом нет никакой необходимости (и, вероятно, смысла) в создании объектов класса `Shape` (трудно изобразить абстрактную фигуру).

В этом случае можно определить так называемую чисто виртуальную функцию (pure virtual function), что достигается при помощи инициализатора "`= 0`":

```
class Shape {
public:
    virtual void Draw() = 0;
    virtual void Erase() = 0;
};
```

Класс, содержащий хотя бы одну чисто виртуальную функцию, называется абстрактным (abstract) классом. Объекты такого класса создавать нельзя (но определять переменную-указатель на такой тип можно). Замена в программе (см. листинг 5.3) класса `Shape` на абстрактный класс позволит более точно выразить смысл класса `Shape` как общей абстрактной модели геометрической фигуры (хотя на результатах выполнения программы эта замена никак не отразится).

Дополняя обсуждение отношения реализации в *разд. 2.4*, отметим, что абстрактные классы, содержащие только открытые чисто виртуальные функции, являются инструментом реализации концепции интерфейса в языке C++ (таким образом, интерфейс `IColorable` может быть представлен на языке C++ абстрактным классом).

Виртуальные функции: некоторые подробности

После изучения основных идей организации управления полиморфными объектами, рассмотрим ряд важных подробностей в связи с определением и использованием виртуальных функций.

Разрушение среды функционирования полиморфных объектов. Виртуальные деструкторы

При разрушении объекта класса, являющегося элементом иерархии классов, вызываются деструкторы всех классов иерархии в порядке, обратном вызову конструкторов: сначала вызывается деструктор данного класса, затем дест-

руктор его базового класса, далее деструктор следующего вверх по иерархии класса и т. д. Соответственно, из деструктора можно вызывать методы и обращаться к открытым данным базовых классов.

В отличие от конструкторов, деструкторы могут и даже иногда должны быть виртуальными. Необходимость в виртуальных деструкторах связана с обстоятельствами, которые иллюстрирует небольшой пример (листинг 5.6).

Листинг 5.6. Зачем нужны виртуальные деструкторы

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

/*
 * Функции некоторой внешней библиотеки
 */
// Функция моделирует захват некоторого внешнего ресурса
void CatchResource( int number ) {
    cout << "External resource No." << number << " caught" << endl;
    // ...
}
// Функция моделирует освобождение некоторого внешнего ресурса
void ReleaseResource( int number ) {
    cout << "External resource No." << number << " released" << endl;
    // ...
}

/*
 * Классы, использующие внешние ресурсы
 */
class ResourceManager {
public:
    // Конструктор: захватывает некоторый внешний ресурс
    ResourceManager() {
        CatchResource( 1 );
        // Другие необходимые действия...
    }
    // Деструктор: освобождает захваченный внешний ресурс
    ~ResourceManager() {
        ReleaseResource( 1 );
        // Другие необходимые действия...
    }
};
```

```
class SpecificManager : public ResourceManager {
public:
    // Конструктор: захватывает другой внешний ресурс
    SpecificManager() {
        CatchResource( 2 );
        // Другие необходимые действия...
    }
    // Деструктор: освобождает захваченный внешний ресурс
    ~SpecificManager() {
        ReleaseResource( 2 );
        // Другие необходимые действия...
    }
};

void main () {
    // Создаем объект класса SpecificManager
    ResourceManager *mgr = new SpecificManager;

    // Используем его...

    // Удаляем объект
    delete mgr;
}
```

Предположим, что с точки зрения целей проектирования объект класса `SpecificManager` управляется посредством переменной-указателя на класс `ResourceManager`. При создании объекта класса `SpecificManager` последовательно вызываются конструкторы базового и производного классов, каждый из которых обеспечивает захват некоторого внешнего ресурса. Однако в соответствии с концепцией статического связывания будет вызван только деструктор класса `ResourceManager`, что и подтверждает вывод программы:

```
External resource No.1 caught
External resource No.2 caught
External resource No.1 released
```

Таким образом, при удалении из памяти объекта, адресуемого указателем `mgr`, с использованием операции `delete` вызывается деструктор класса `ResourceManager` (ведь компилятор полагает, что с указателем `mgr` связывается объект класса `ResourceManager`), а значит — не произошло корректное освобождение ресурса, занятого в процессе работы конструктора класса `SpecificManager`.

А как же с деструктором класса `SpecificManager`? Знакомая картина, не правда ли? К счастью, механизм виртуальных функций работает и для деструкто-

ров. Для исправления ситуации достаточно объявить деструктор базового класса виртуальным (листинг 5.7).

Листинг 5.7. Использование виртуальных деструкторов

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

/*
 * Функции некоторой внешней библиотеки
 */
// Функция моделирует захват некоторого внешнего ресурса
void CatchResource( int number ) {
    cout << "External resource No." << number << " caught" << endl;
    // ...
}
// Функция моделирует освобождение некоторого внешнего ресурса
void ReleaseResource( int number ) {
    cout << "External resource No." << number << " released" << endl;
    // ...
}

/*
 * Классы, использующие внешние ресурсы
 */
class ResourceManager {
public:
    // Конструктор: захватывает некоторый внешний ресурс
    ResourceManager() {
        CatchResource( 1 );
        // Другие необходимые действия...
    }
    // Деструктор: освобождает захваченный внешний ресурс
    virtual ~ResourceManager() {
        ReleaseResource( 1 );
        // Другие необходимые действия...
    }
};

class SpecificManager : public ResourceManager {
    // см. листинг 5.6

void main () {
    // Создаем объект класса SpecificManager
    ResourceManager *mgr = new SpecificManager;
```

```
    // Используем его...

    // Удаляем объект
    delete mgr;
}
```

В результате проделанной модификации при удалении объекта, управляемого указателем `mgr`, сначала будет вызван деструктор класса `SpecificManager` и только потом — деструктор класса `ResourceManager`. При этом вывод программы имеет следующий вид:

```
External resource No.1 caught
External resource No.2 caught
External resource No.2 released
External resource No.1 released
```

Здесь отметим отличие виртуальных деструкторов от обычных виртуальных функций. Виртуальная функция производного класса вызывается "вместо" виртуальной функции базового класса, а деструктор производного класса вызывается "перед" виртуальным деструктором базового класса. Адреса виртуальных деструкторов помещаются в таблицы виртуальных функций аналогично другим виртуальным функциям.

Вызов виртуальной функции из функции-члена класса

Предшествующие примеры иллюстрировали обращение к виртуальным функциям извне. Не реже на практике встречается обращение к виртуальным функциям из пределов базового класса (который при этом может быть даже абстрактным).

Рассмотрим игрушечный пример, наглядно демонстрирующий обращение к виртуальным функциям, включая виртуальные деструкторы (листинг 5.8).

Листинг 5.8. "Пример с поеданием апельсина" (абстрактные классы и виртуальные функции)

```
#include <cassert>

#include <iostream>
using namespace std;

class Fruit {
public:
    virtual void Eat() {
        cout << "Inside Fruit::Eat()" << endl;
```

```
        while( !IsEaten() ) {
            TakePiece();
        }
        FinishEating();
    }
    virtual bool IsEaten() = 0;
    virtual void TakePiece() = 0;

    virtual void FinishEating() {
        // No Actions by default
    }

    virtual ~Fruit() {
        cout << "Fruit destructed" << endl;
    }
};

class Orange : public Fruit {
    int pieces;
public:
    Orange( int _pieces = 8 ) {

        pieces = _pieces;
        cout << "Got an orange of " << pieces << " pieces!";
        cout << endl;
    }

    bool IsEaten() {
        cout << "Inside Orange::IsEaten()" << endl;

        return pieces > 0 ? false : true;
    }

    void TakePiece() {
        cout << "Inside Orange::TakePiece(): ";
        pieces--;

        cout << pieces << " piece(s) left." << endl;
    }

    void FinishEating() {
        cout << "Whole orange has been eaten" << endl;
    }

    ~Orange() {
        cout << "Orange destructed" << endl;
    }
};
```

```
void main() {

    Fruit *pFruit = new Orange( 4 );
    assert( pFruit != 0 );

    pFruit->Eat();
    delete pFruit;
}
```

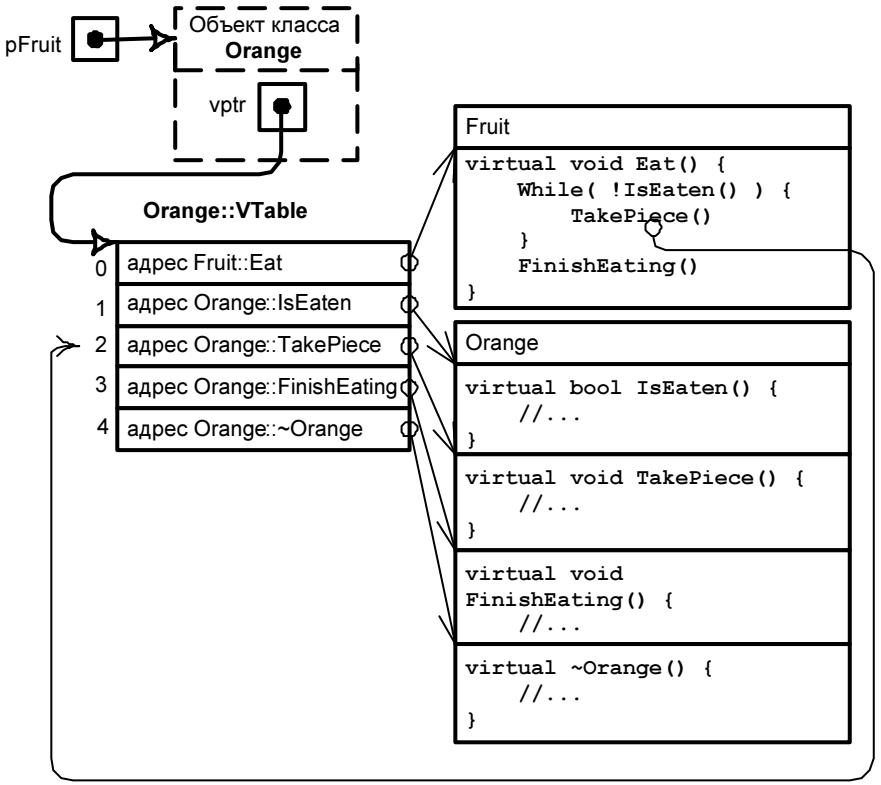


Рис. 5.5. Управление данными и действиями в "примере с поеданием апельсина"

Рис. 5.5 иллюстрирует организацию таблицы виртуальных функций для объекта класса `Orange`. Виртуальная функция `Fruit::Eat()` не замещена в классе `Orange`, поэтому здесь таблица виртуальных функций содержит адрес функции базового класса. Остальные методы класса `Fruit` замещены в классе `Orange`, поэтому таблица виртуальных функций содержит адреса соответствующих функций-членов класса `Orange`. Рассмотрим порядок разрешения полиморфных вызовов на примере вызова функции `TakePiece()` из функции

`Eat()`. Поскольку функция `TakePiece()` объявлена в классе `Fruit` виртуальной (более того — чисто виртуальной), транслятор вычисляет адрес тела функции, используя информацию об адресе таблицы виртуальных функций и индексе функции `TakePiece()` в таблице:

```
адрес TakePiece = *(pFruit->vptr+2)
```

Поскольку создается объект класса `Orange`, указатель `vptr` указывает на версию таблицы виртуальных функций класса `Orange`. Таким образом обеспечивается вызов правильной версии функции-члена класса `Orange`, замещая обращение к чисто виртуальному методу.

В результате исполнения этой программы на консоль будет выведен следующий текст:

```
Got an orange of 4 pieces
Inside Fruit::Eat()
Inside Orange::IsEaten()
Inside Orange::TakePiece(): 3 piece(s) left
Inside Orange::IsEaten()
Inside Orange::TakePiece(): 2 piece(s) left
Inside Orange::IsEaten()
Inside Orange::TakePiece(): 1 piece(s) left
Inside Orange::IsEaten()
Inside Orange::TakePiece(): 0 piece(s) left
Inside Orange::IsEaten()
Whole orange has been eaten
Orange destructed
Fruit destructed
```

Анализируя вывод программы, отмечаем, что, наряду с обеспечением правильного управления вызовами виртуальных функций, при разрушении объекта сначала вызван деструктор класса `Orange`, а затем — деструктор базового класса `Fruit`.

Можно ли в производном классе определить невиртуальную функцию вместо виртуальной функции базового класса?

C++ не предоставляет стандартного встроенного в язык механизма, позволяющего программисту указать, что объявляемая функция-член производного класса не должна участвовать в процессе полиморфных разрешений вызовов *(см. также разд. 11.1)*.

Если это все же необходимо сделать, можно воспользоваться объявлением функции, которое отличается от объявления виртуальной функции хотя бы

одним параметром. Этот параметр может быть фиктивным и не использоваться в теле функции (его стоит объявить с возможностью задания аргумента по умолчанию). Этот подход, возможно, не слишком изящен, но работает. Например, если в классе `Student` функция `Print()` объявлена виртуальной, а в классе `GraduateStudent` нам нужна обыкновенная переопределяющая (а не замещающая) функция, можно определить ее следующим образом:

```
class GraduateStudent : public Student {
public:
    // ...

    void Print( int x = 0 ) { // Параметр x — фиктивный
        // ...
    }
};
```

Теперь списки параметров функций `Print()` в классах `Student` и `GraduateStudent` не совпадают, а значит функция `GraduateStudent::Print()` является совершенно самостоятельной и не замещает функцию `Student::Print()`:

```
Student *grad1 = new GraduateStudent;
grad1->Print(); // Вызывается Student::Print!
                // (несмотря на то, что она виртуальная)
GraduateStudent *grad2 = new GraduateStudent;
grad2->Print(); // Вызывается GraduateStudent::Print
```

Функции-не члены класса, работающие подобно виртуальным функциям

В некоторых случаях может возникнуть потребность в функции-не члене класса, которая внешне ведет себя подобно виртуальным функциям. Рассмотрим пример, представленный листингом 5.9.

Листинг 5.9. Виртуальные перегруженные операции-члены класса

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

class UserAbstract {
public:
    virtual ostream& operator<<( ostream& stream ) const = 0;
};
```

```
class UserType1 : public UserAbstract {
public:
    ostream& operator<<( ostream& stream ) const {
        stream << "UserType1 object data printed" << endl;
        return stream;
    }
};

class UserType2 : public UserAbstract {
public:
    ostream& operator<<( ostream& stream ) const {
        stream << "UserType2 object data printed" << endl;
        return stream;
    }
};

void f( UserAbstract& refObj ) {
    refObj << cout; // Неестественная форма использования операции
                    // (работает, но путает читателя)
}

void main() {
    UserType1 obj1;
    UserType2 obj2;

    f( obj1 );
    f( obj2 );
}
```

В данном примере реализована иерархия классов, при этом обеспечивается полиморфный вызов перегруженных операций вывода *(см. также информацию в разд. П2.2)*. В результате исполнения программы, как и следовало ожидать, будут выведены следующие сообщения:

```
UserType1 object data printed
UserType2 object data printed
```

Проблема данной программы заключается в том, что, несмотря на формальную корректность предложенных определений, обращение к виртуальной операции-члену класса из функции `f()` выглядит неестественно. Действительно, вопреки стандартным соглашениям, потоковый объект располагается справа (а не слева) относительно операции <<, а выводимый объект — слева (а не справа). Все это, в конечном счете, запутывает читателя программы (да и выглядит не очень привлекательно).

Решение проблемы может быть достигнуто с использованием определения вспомогательных виртуальных функций (вместо виртуальных операций) и

последующем обращении к этим функциям. При этом перегрузка операции осуществляется в виде функции-не члена класса [Meyers, 1995, Item 25]. Перегруженная операция, используя ссылку на выводимый объект, просто переадресует вызов соответствующей виртуальной функции (листинг 5.10).

Листинг 5.10. "Виртуализация" функций-не членов класса

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

class UserAbstract {
public:
    virtual ostream& print( ostream& stream ) const = 0;
};

class UserType1 : public UserAbstract {
public:
    ostream& print( ostream& stream ) const {
        stream << "UserType1 object data printed" << endl;
        return stream;
    }
};

class UserType2 : public UserAbstract {
public:
    ostream& print( ostream& stream ) const {
        stream << "UserType2 object data printed" << endl;
        return stream;
    }
};

inline
ostream& operator<<( ostream& stream, const UserAbstract& refObj ) {
    return refObj.print( stream ); // Обращаемся к виртуальной функции
}

void f( UserAbstract& refObj ) {
    cout << refObj; // Работает и выглядит естественно
}

void main() {
    UserType1 obj1;
    UserType2 obj2;
```

```
    f( obj1 );
    f( obj2 );
}
```

В этом случае обращение к операции << выглядит совершенно естественно и полностью согласуется с общепринятым стандартом. Объявление операции в качестве подставляемой функции позволяет избежать накладных расходов в связи с вызовом функции, все содержание которой заключается в полиморфном вызове функции `print()`.

5.3. Динамическое связывание и приведение типов

Реализация механизма динамического связывания позволяет при необходимости использовать информацию о типах объектов в ходе выполнения программы и осуществлять преобразования типов во время исполнения (runtime casts).

RTTI (определение типа во время выполнения) и понижающее приведение типов

Вернемся к примеру с геометрическими фигурами и рассмотрим текст программы листинга 5.3. Как следует из определения класса `Triangle`, он предоставляет не только замещающие функции для `Draw()` и `Erase()`, но также добавляет новую функцию `Rotate90()`.

Напрямую воспользоваться этой функцией посредством указателя на `Shape` нельзя (интерфейс `Shape` не поддерживает метод `Rotate90`). Статическое преобразование типа не годится, т. к. элемент массива `s` вовсе не обязательно указывает на объект типа `Triangle`. Выход из ситуации обеспечивает динамическое преобразование типа `dynamic_cast`, оперирующее с типами времени выполнения. Отсюда и название механизма — RTTI, Run Time Type Information. Использование данного механизма позволяет осуществить преобразование указателя на базовый тип к указателю на производный тип, т. е. выполнить так называемое понижающее преобразование типа (downcast) во время выполнения программы (рис. 5.6).

В отличие от повышающего преобразования (которое является успешным по определению, и поэтому оно осуществляется неявно), попытка понижающего преобразования может окончиться неудачей, поэтому результат понижающего преобразования следует проверять (см. выделенный фрагмент в листинге 5.11).

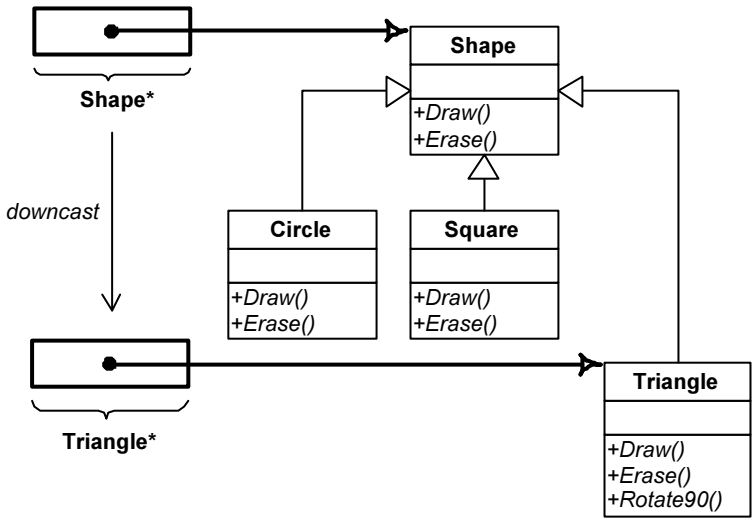


Рис. 5.6. Понижающее преобразование типа (downcast)

Листинг 5.11. Динамическое приведение типа и RTTI

```
// До функции main — как в листинге 5.3

// Тестирующая программа
void main() {

    PShape s[ 5 ]; // Массив указателей на геом. фигуры
    int i;         // Счетчик при переборе объектов из s

    // Заполнение массива указателей на геометрические фигуры
    Shapes::ShapesInit();
    for( i = 0; i < 5; i++ ) {

        // Фигура генерируется случайным образом!
        Shapes::RandomShape( s[ i ] );
    }

    for( i = 0; i < 5; i++ ) {

        // Вызов виртуальных методов
        s[ i ]->Draw();
        s[ i ]->Erase();

        // Динамическое приведение типа (требует RTTI)
        Triangle *tr;
```

```
        tr = dynamic_cast<Triangle *>( s[ i ] );
        if( tr != 0 ) {

            // Вызов метода Rotate90
            tr->Rotate90();
        }
    }

    // Освобождение памяти, занятой объектами
    for( i = 0; i < 5; i++ ) {
        delete s[ i ];
    }
}
```

Отметим также, что приведение типа между производными классами одного базового класса называют перекрестным приведением (crosscast). По сути, перекрестное приведение является комбинацией повышающего и понижающего преобразований, поэтому также требует поддержки механизма RTTI.

Для выполнения понижающего или перекрестного приведения требуется, чтобы объект допускал полиморфное поведение. С одной стороны, это требование связано с тем, что упрощается задача разработчика компилятора (указатель на объект, содержащий информацию о типе, может быть просто добавлен в таблицу виртуальных функций). С другой стороны, использование понижающего преобразования только с полиморфными типами выглядит довольно логичным, т. к. если у объекта нет виртуальных функций, им нельзя небезопасно манипулировать, не зная его тип, следовательно, нужно стараться не использовать объект в контексте, в котором его тип неизвестен. Если же тип известен, то нет нужды использовать преобразование типа времени выполнения (подробности см. в [Stroustrup, 2000], разд. 15.4). Если все же нужно использовать RTTI, а у объекта нет виртуальных функций, следует определить одну фиктивную виртуальную функцию в базовом классе, которая содержит пустой код. Это форсирует создание таблиц виртуальных функций для всех объектов производных типов (что и будет платой за использование RTTI в контексте, в котором, может быть, можно было обойтись и без определения типа во время выполнения).

Включение в таблицы виртуальных функций указателей на объекты, содержащие информацию о типе, происходит только при установке специального ключевого параметра транслятора. Таким образом, подключение механизма RTTI требует явного разрешения (по умолчанию возможность определения типа во время выполнения отключена). Например, при проектировании в среде Visual Studio 6.0 необходимо в настройках проекта (команда **Settings** меню **Project**) выбрать вкладку **C/C++**, далее категорию **C++ Language** и от-

метить флажком пункт *"Enable Run-Time Type Information (RTTI)"* (эквивалентно ключу транслятора `/GR`).

RTTI: проблемы использования

Возможность получить информацию о типе объекта во время выполнения при неверном использовании может, вообще говоря, порождать код, повторное использование которого затруднено или невозможно. Эккель отмечает, что у многих новичков, работавших прежде с процедурными языками, появляется соблазн превратить свои программы в каскад предложений `switch`, теряя те преимущества объектно-ориентированного подхода, которые обеспечиваются полиморфизмом [Eckel, 2000-1], [Eckel, 2000-2], [Stroustrup, 2000]. Наличие программ, использующих RTTI подобным образом, является одной из причин, по которой RTTI подвергается критике, поскольку-де порождает трудноперенос имый и немодифицируемый код. В связи с этим важно подчеркнуть, что проблемой является не использование RTTI как такового, а его использование в неправильном контексте, его использование там, где, вообще говоря, без RTTI можно было бы обойтись.

В действительности, в большинстве случаев, когда требуется обеспечить реализацию различных действий в зависимости от типов объектов, следует использовать виртуальные функции. Единственное ограничение в этом случае заключается в необходимости иметь доступ к представлению базового класса. Если по ходу развития программы выясняется, что базовый класс не содержит некоторых необходимых методов, можно просто расширить интерфейс базового класса, добавляя в него новые методы. Например, в случае с геометрическими фигурами можно написать (листинг 5.12).

Листинг 5.12. Обеспечение альтернативного поведения объектов без RTTI

```
#include <cstdlib>     // для srand() и rand()
#include <ctime>       // для time()

#include <iostream>
using namespace std;

class Shape {
public:
    virtual void Draw() = 0;
    virtual void Erase() = 0;
    virtual void Rotate90() {
    // В общем случае эта операция невозможна:
    // реализуем обработчик ошибок
        cout << "Error. No rotation supported" << endl;
```

```
        // Примечание: в данном случае вместо обработки ошибки
        //             на месте лучше воспользоваться
        //             генерацией исключения (см. главу 6)
    }
};

class Circle : public Shape {
public:
    void Draw() {
        cout << "Circle::Draw()" << endl;
    }
    void Erase() {
        cout << "Circle::Erase()" << endl;
    }
};

class Square : public Shape {
public:
    void Draw() {
      cout << "Square::Draw()" << endl;
    }
    void Erase() {
      cout << "Square::Erase()" << endl;
    }
};

class Triangle : public Shape {
public:
    void Draw() {
      cout << "Triangle::Draw()" << endl;
    }
    void Erase() {
      cout << "Triangle::Erase()" << endl;
    }
    // Обычная виртуальная функция
    void Rotate90() {
        cout << "Triangle::Rotate90()" << endl;
    }
};
typedef Shape *PShape;

class ShapesWithoutRTTI {
public:
    static ShapesInit() {
        srand( ( unsigned )time( NULL ) );
    }
```

```
    static void randShape( PShape& s ) {
        switch( (rand() % 3) ) {
        default:
        case 0 : s = new Circle;
                 break;
        case 1 : s = new Square;
                 break;
        case 2 : s = new Triangle;
                 break;
        }
    }
};
```

В этом случае обращение к методу `Rotate90()` выглядит так же, как и обращение к другим виртуальным функциям:

```
void main() {

    PShape s[ 5 ];
    int i;

    // Fill up the array with shapes:
    ShapesWithoutRTTI::ShapesInit();
    for( i = 0; i < 5; i++ )
        ShapesWithoutRTTI::randShape( s[ i ] );

    for( i = 0; i < 5; i++ ) {

        s[ i ]->Draw();
        s[ i ]->Erase();

        s[ i ]->Rotate90();
    }

     for( i = 0; i < 5; i++ )
        delete s[ i ];
}
```

Поскольку в классах `Circle` и `Square` нет специфической реализации метода `Rotate90()`, при обращении к этому методу на объектах класса `Circle` и `Square` будет использоваться версия, определенная в классе `Shape`: происходит обработка ошибки. Если же переменная `s[i]` указывает на объект класса `Triangle`, то, благодаря полиморфизму, вызывается версия виртуальной функции, определенная в классе `Triangle`. Недостаток реализации в связи с обработкой ошибки на месте легко преодолевается с использованием механизма обработки исключений. *Организация работы с исключениями подробно рассматривается в гл. 6.*

Итак, без RTTI вроде бы можно обойтись. Однако это не совсем так. Даже рассмотренный простой пример дает пищу для размышления. Для того чтобы избежать RTTI, приходится модифицировать базовые классы. Это оправдано, если только подобная модификация используется многократно, а не покрывает необходимость реализации одной-двух нестандартных особенностей производных классов. Ведь в этом случае практически неизбежна необходимость написания фиктивных виртуальных функций в классах, в которых они не нужны или попросту не имеют смысла. Как справедливо замечает Эккель, "это делает интерфейс базового класса запутанным и раздражает тех, кому приходится переопределять ненужные абстрактные методы при наследовании от базового класса" [Eckel, 2000-1]. Таким образом, даже если программист имеет доступ к представлению базового класса, он может не захотеть расширять его интерфейс по указанной выше причине.

Наконец, программист может вообще не иметь возможности модифицировать определение базового класса (например, если этот класс является библиотечным). В этом случае использование RTTI не только неизбежно, но и полезно: такое решение не нарушает основ построения иерархий классов на базе использования полиморфизма и не ведет к неоправданному разветвлению программы. Отдельные условные инструкции с `dynamic_cast` появляются только в связи с некоторыми специфическими операциями, актуальными только для работы с отдельными типами.

В заключение следует заметить, что, как показано в [Легалов, 2002], реализация концепции полиморфизма не обязательно должна опираться на объектно-ориентированную модель управления действиями и данными. Использование механизма RTTI позволяет реализовать безопасную по отношению к типам модель процедурно-параметрического полиморфизма, обеспечивающего создание внешних функций (мультиметодов), работающих с родственными объектами. Такое решение может способствовать упрощению интерфейса производных классов и устранению жестких связей между классами, составляющими иерархию классов.

Основной вывод, вытекающий из материала, рассмотренного в данном разделе, заключается в том, что *важно различать разумное и неразумное использование механизма RTTI.*

Дополнительным условием применимости RTTI является известность всех типов на момент компиляции. То есть все типы, которые потенциально могут потребовать понижающих преобразований времени выполнения, должны быть известны заранее. В современных условиях это не всегда так. Объект может использоваться из внешнего динамически связываемого компонента, объект может создаваться и использоваться удаленно, и т. д. Таким образом, возможно возникновение ситуаций, когда само *имя* типа становится извест-

ным только во время выполнения, при этом этот тип не был известен во время трансляции, так что использовать его для динамического преобразования на основе механизма RTTI невозможно. Для решения подобной проблемы в современных технологиях используется механизм отражения, или рефлексии (reflection). *Основные элементы механизма отражения вкратце обсуждаются в гл. 11 (см. разд. 11.5—11.6).*

5.4. Несколько замечаний о преобразованиях типа в C++

В примерах данной главы несколько раз встречаются явные преобразования типов, поэтому, учитывая учебный характер книги, здесь уместно немного отклониться от основного повествования для обсуждения некоторых деталей преобразований типов в C++.

В C++ с момента его появления существовали две синонимичные формы преобразования типов. Первая (так называемая операторная запись) была унаследована от языка C. Вторая (функциональная запись) появилась в C++.

Дальнейшее развитие языка C++ и анализ ошибок, возникающих при приведениях типов, обусловили создание новой нотации приведения типов, допускающей однозначные интерпретации преобразований типов и обеспечивающие бóльшую строгость с точки зрения безопасности преобразований.

Страуструп приводит следующий пример, иллюстрирующий недостатки старых приведений типов [Stroustrup, 2000]:

```
const X* pc = new X;
// ...
Y pv = ( Y* ) pc;
```

Какой цели хотел достичь программист, совершая это преобразование? Получить указатель на тип, никак не связанный с `X`? Избавиться от атрибута `const`? Получить доступ к классу `Y`, который является базовым для `X`?

Для исправления подобных ситуаций было решено вычленить логически различные приведения типов и поддержать их с помощью специальных языковых конструкций. В C++ определено четыре вида явных преобразований типов. Рассмотрим их подробнее.

Преобразование типов во время компиляции (преобразование *static_cast*)

Преобразование типа `static_cast` выполняется компилятором и должно осуществляться только в том случае, когда это действительно необходимо. Если

оно достигается стандартным преобразованием C++, не следует использовать `static_cast` только для того, чтобы избавиться от предупреждений транслятора. Предупреждения позволяют избежать ошибок, явное же преобразование скрывает возможные ошибки.

Примером "хорошего" преобразования `static_cast` может быть преобразование указателя на `void` к указателю на необходимый тип (например, при управлении резервированием памятью). Это как раз тот случай, когда присваивание без явного преобразования запрещено правилами языка. Явным преобразованием программист заявляет о том, что ему известна потенциальная опасность преобразования указателей. В случае ошибки конструкцию `static_cast` легко обнаружить в программе (если, конечно, программист не использует их там, где не нужно).

Другим примером приемлемого преобразования `static_cast` может быть приведение целого типа к типу "перечисление" или для других преобразований родственных типов.

Эккель приводит следующую краткую классификацию преобразований [Eckel, 2000-1]:

1. Продвигающие и стандартные преобразования. Могут быть выполнены неявно. Преобразование в стиле `static_cast` не нужно.
2. Стандартные сужающие преобразования (например, из типа `long` в тип `int`, из типа `double` в тип `float` и т. п.). Осуществляются автоматически, при возможности потери значащих разрядов выдается предупреждение. Преобразование в стиле `static_cast` позволяет только избавиться от предупреждений или привлечь внимание читателя.
3. Преобразования указателей на `void` к указателю на произвольный тип. Преобразование в стиле `static_cast` обязательно.

Прежде чем воспользоваться явным преобразованием, задумайтесь о его необходимости. В своей книге "Язык программирования C++", в ней более тысячи страниц, Страуструп помечает чуть более десятка страниц (sic!, важно), содержащих явные преобразования типов в ситуациях, которые он считает реалистичными [Stroustrup, 2000].

Преобразование типов во время выполнения (преобразование *dynamic_cast*)

Преобразование `dynamic_cast` является безопасным по отношению к типам преобразованием времени выполнения. Это означает, что код, написанный с его использованием, работает правильно с классами, находящимися в отношении полиморфного наследования, а также допускает дальнейшее расширение.

Преобразование dynamic_cast необходимо в тех случаях, когда правильность преобразования невозможно установить на этапе компиляции. Преобразование dynamic_cast позволяет реализовать понижающее приведение типов как для указателей на объекты, так и для ссылок. Отличие применения dynamic_cast для указателей и для ссылок заключается в том, что в случае преобразования указателей следует учитывать случай, когда преобразование закончилось неудачей. В этом случае dynamic_cast возвращает 0. При преобразовании ссылок такой возможности нет (поскольку ссылка не может существовать сама по себе). Страуструп замечает, что для некоторой ссылки ref преобразование dynamic_cast<T&>(ref) является уже не вопросом (как в случае с указателями), а утверждением [Stroustrup, 2000]. Поэтому при преобразованиях ссылок ошибочность этого утверждения можно установить только посредством обработки исключений *(см. гл. 6)*.

Преобразования для избавления от константности *(const_cast)*

Преобразование const_cast применяется для того, чтобы аннулировать действие модификатора const:

```
const int i;
int *j = const_cast <int *>( &i );
```

Необходимость преобразований такого рода связана с тем, что при применении операции & к константному объекту результатом является константный указатель, т. е. такой указатель, который не позволяет модифицировать связанный с ним объект. Если требуется получить обычный указатель, применяется преобразование в стиле const_cast.

Преобразование "на свой страх и риск" *(reinterpret_cast)*

Преобразование reinterpret_cast является наиболее редким видом преобразования, которое встречается тогда, когда нужно осуществить экзотические преобразования типов. В результате преобразования reinterpret_cast получается значение нового типа, состоящее из той же цепочки двоичных разрядов, что и аргумент преобразования. Вот пример такого преобразования.

```
// Печать двоичного кода представления значений с
// плавающей точкой (кода IEEE 754) в восьмеричном формате
//
// Функция работает для 32-разрядной архитектуры
// (значение int имеет длину 32 двоичных разряда)
```

```
void PrintFloatAsHex( float value ) {
    unsigned int *pValue =
        reinterpret_cast <unsigned int *> ( &value );
    cout << oct << *pValue << endl;
}
```

Комментарий к приведенной функции лишний раз подчеркивает, что код, содержащий преобразование в стиле `reinterpret_cast`, как правило, является непереносимым (но не обязательно "нестерпимым" ☺).

5.5. Вопросы и упражнения

1. Проанализируйте приведенную ниже программу и, если она корректна, напишите, что будет напечатано в результате ее исполнения. Если, по Вашему мнению, в результате трансляции будут обнаружены ошибки, укажите, какие.

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

class A {
public:
    A() {
        cout << "Constructor A::A()" << endl;
    }
    virtual void print( int ) = 0;
};

class B : public A {
public:
    B() {
        cout << "Constructor B::B()" << endl;
    }
    void print( int x ) {
        cout << "From B::print(): x=" << x << endl;
    }
};

class C : public B {
public:
    C() {
        cout << "Constructor C::C()" << endl;
    }
```

```
    void print( int x ) {
        cout << "From C::print(): x=" << x << endl;
    }
};

void print ( A *obj, int x ) {
    obj->print( x );
}

void main() {
    A *obj1;
    B *obj2;

    obj1 = new B;
    obj2 = new C;

    print( obj1, 10 );
    print( obj2, 20 );
}
```

2. Проанализируйте приведенную ниже программу и, если она корректна, напишите, что будет напечатано в результате ее исполнения. Объясните понятия статического и динамического типов и проиллюстрируйте Ваше объяснение на примере объектов, используемых в программе. Повышающее или понижающее приведение типов используется в данном примере?

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

class A {
public:
    A() { cout << "Constructor A::A()" << endl; }
    virtual void print( int ) = 0;
};

class B : public A {
public:
    B() { cout << "Constructor B::B()" << endl; }
    void print( int x ) {
        cout << "From B::print(): x=" << x << endl;
    }
};

class C : public A {
public:
    C() { cout << "Constructor C::C()" << endl;}
```

```
    void print(int x, int y = 0) {
        cout << "From C::print(): x=" << x;
        if( y != 0 ) cout << " y=" << y;
        cout << endl;
    }
};

void print ( A *obj, int x ) {
    obj->print( x );
}

void main() {
    A *obj1 = new B, *obj2 = new C;
    print( obj1, 10 );
    print( obj2, 20 );
    delete obj1;
    delete obj2;
}
```

3. Разработайте программу, печатающую на консоль двоичное представление заданного числа в формате `double`.
4. [Eckel, 2000-1] Вообразим, что виртуальные функции не поддерживаются C++, однако преобразование `dynamic_cast` доступно (кстати, объясните странность этого предположения). С использованием динамического преобразования типов напишите код для работы с геометрическими фигурами, эквивалентный по выполняемым действиям программе листинга 5.3. Объясните, почему это плохая идея.
5. Напишите, что будет напечатано в результате исполнения предложенной программы. Объясните особенности обращения к чисто виртуальным функциям из функций-членов абстрактных классов.

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

class A {
public:
    virtual void print( int ) = 0;
    void callPrint( int x ) {
        print( x );
    }
};

class B : public A {
public:
```

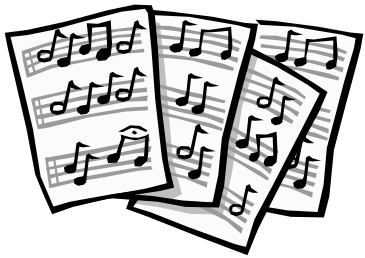
```
    void print( int x ) {
        cout << "From B::print(): x=" << x << endl;
    }
};

class C : public B {
public:
    void print( int x ) {
        cout << "From C::print(): x=" << x << endl;
    }
};

void main() {
    A *obj1 = new C;
    B *obj2 = new C;
    obj1->callPrint( 1 );
    obj2->callPrint( 2 );
    delete obj1;
    delete obj2;
}
```

6. Напишите, что будет напечатано в результате исполнения предложенной программы. Объясните особенности использования виртуальных деструкторов. Когда они являются необходимыми?

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

class A {
public:
    A() { cout << "Constructor A::A()" << endl; }
    ~A() { cout << "Destructor A::A()" << endl; }
};

class B : public A {
public:
    B() { cout << "Constructor B::B()" << endl; }
    virtual ~B() { cout << "Destructor B::B()" << endl; }
};

class C : public B {
public:
    C() { cout << "Constructor C::C()" << endl; }
    ~C() { cout << "Destructor C::C()" << endl; }
};
```

```
void main() {
    A *obj1 = new C;
    B *obj2 = new C;
    C c;
    // ...
    delete obj1;
    delete obj2;
}
```

ГЛАВА 6



Обработка ошибок на основе использования механизма исключений

1. Всегда найдется пользователь, не знающий, в чем состоит задача.
2. Всегда найдется пользователь, знающий, в чем состоит задача, но на самом деле задача совсем другая.
3. Всегда найдется пользователь, знающий, в чем состоит задача, но программист не хочет понять или услышать пользователя.
4. Всегда есть более чем один пользователь, и каждый находится в одной из трех перечисленных ситуаций.
5. Популяция пользователей изменяется со временем.
6. Задача изменяется со временем.

Джералд Вейнберг[1]

Существенным элементом любого приложения является обработчик ошибок. Ответственность за непредвиденное завершение программы полностью лежит на программисте, а не на пользователе. Надежная программа порождает правильный результат в тех случаях, когда обрабатываемые данные и действия пользователя корректны, а так же в тех случаях, когда это не так (в этом случае правильным результатом является завершение программы с выдачей информационных сообщений и (или) корректным разрушением среды исполнения приложения).

В данной главе рассматривается механизм обработки ошибок на основе генерации и обработки исключений (exception handling), становящийся де-факто стандартом обработки ошибок в объектно-ориентированном проектировании.

---

[1] Gerald Weinberg. Understanding the Professional Programmer [Weinberg, 1988] (*пер. с англ.* — Е. П.).

6.1. Варианты обработки ошибок, не связанные с использованием исключений

Прежде чем перейти к рассмотрению механизма генерации и обработки исключений и особенностей его реализации различными языками программирования, проанализируем основные стандартные подходы, применяющиеся к обработке ошибочных ситуаций в процедурных программах.

Процесс обработки ошибки можно представить в виде последовательности, состоящей из трех действий:

1. Выявление ошибочной ситуации.
2. Генерация информации о возникновении ошибочной ситуации.
3. Обработка ошибки (определение действий в связи с выявленной ситуацией).

Первый этап (выявление ошибочной ситуации) связан с особенностями предметной области и вытекающими из этих особенностей требованиям к данным и организации вычислительного процесса.

Остальные этапы представляют собой более или менее стандартные элементы архитектуры приложения и могут быть реализованы с определенной долей унификации. Основная сложность реализации корректной обработки ошибок заключается не в том, как обнаружить ошибку, а в том, как сообщить об этой ошибке отдельным модулям проекта и как определить набор действий, обрабатывающих ошибочную ситуацию в том случае, когда эта обработка не может быть осуществлена в пределах одной функции и даже в пределах отдельного модуля.

В связи с организацией обработки ошибки наиболее распространены следующие подходы, применяемые независимо или в комбинации.

- Обработка ошибки на месте.
- Использование возвращаемых значений.
- Использование глобальных переменных или переменных-членов класса, исполняющих роль индикаторов состояния приложения или объекта.
- Использование специально разработанных функций.
- Использование функций обратного вызова (callback functions).
- Использование генерации и обработки исключений.

Рассмотрим вкратце основные особенности перечисленных подходов.

Обработка ошибки на месте

Данный подход применен в программе листинга 3.4. Вспомним, что при обсуждении недостатков реализации класса `QueueInt` особое внимание было уделено недостаткам в связи с использованной схемой обработки ошибок *(см. разд. 3.3)*. Основной недостаток подобного подхода заключается в том, что, находясь в пределах области видимости функции, где была диагностирована ошибочная ситуация, мы можем не иметь доступа к данным и функциям, необходимым для корректной и полной обработки возникшей ситуации. Обработка ошибок на месте затрудняет разработку унифицированных компонентов, которые могли бы использоваться в различных приложениях.

Еще одним фактором, обуславливающим недостатки такого подхода, является трудность реализации обработки ошибок разного типа (например, ошибок, требующих или не требующих остановки выполнения программы).

Использование возвращаемых значений

Применение в качестве носителя информации об ошибке возвращаемого значения дает возможность перенести обработку ошибки за пределы функции, где была диагностирована ошибочная ситуация (листинг 6.1).

Листинг 6.1. Обработка ошибок на основе использования возвращаемых значений

```
// ReadData.h
#ifndef _ReadData_h
#define _ReadData_h

#include <fstream>
using namespace std;

// Модельная функция чтения (считывает целое число)
int ReadData( ifstream&, int& );

// Коды состояния процесса чтения данных
#define SUCCESS              0x0000
#define UNEXPECTED_EOF       0x0001
#define BAD_FORMAT           0x0002

#endif

// ReadData.cpp
#include "ReadData.h"
```

```
// Модельная функция чтения (считывает целое число)
int ReadData( ifstream& fInput, // Входной поток
              int& x            // Считываемое данное
            ) {

    // Опережающее чтение символа
    // (не конец ли файла?)
    char ch;
    fInput >> ch;
    if( fInput.eof() ){
        return UNEXPECTED_EOF;
    }

    // Возврат символа во входной поток
    fInput.putback( ch );

    // Чтение целочисленного данного
    fInput >> x;
    if( fInput.fail() ||  fInput.bad() ) {
        return BAD_FORMAT;
    }

    return SUCCESS;
}

// Main.cpp
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;

#include "ReadData.h"

ifstream fInput;
ofstream fOutput;

void main() {
    int x;

    fInput.open( "input.txt", ios_base::in );
    if( !fInput ) {
        cout << "Error while opening input file" << endl;
        exit( 1 );
    }

    fOutput.open( "output.txt", ios_base::out );
    if( !fOutput ) {
```

```
        cout << "Error while opening output file" << endl;
        exit(2);
    }

    switch( ReadData( fInput, x ) ) {

        case SUCCESS :        break;

        case UNEXPECTED_EOF :// Обработка ошибки
                              // ...
                              break;

        case BAD_FORMAT :     // Обработка ошибки
                              // ...
                              break;
    }

    fOutput << "x = " << x << endl;
    fOutput << "Terminated successfully" << endl;

    fOutput.close();
    exit( 0 );
}
```

Использование возвращаемого значения позволяет отделить реализацию выявления ошибки от обработчика ошибки. Однако при этом усложняется интерфейс функций, в которых диагностируется ошибка. В ряде случаев может усложняться реализация не только той функции, которая диагностировала ошибку, а также и вызвавшей ее функции, поскольку вполне вероятна ситуация, когда эта вызывающая функция, в свою очередь, тоже не является подходящим местом для обработки ошибки.

При этом некоторые функции приложения могут использовать возвращаемое значение для реализации элементов решения, не связанных с обработкой ошибок. Некоторые функции могут заканчиваться с разными кодами ошибок — все это требует подробного сопровождения и затрудняет модификацию программ (рис. 6.1).

Разные разработчики и группы разработчиков используют разные подходы к описанию ошибочных ситуаций. Наконец, приемлемость данного подхода в объектно-ориентированных программах ограничена невозможностью сообщить об ошибке в ходе выполнения конструктора или деструктора, которые вообще не имеет возвращаемого значения.

Таким образом, в качестве основы для унификации процесса обработки ошибок данный подход в чистом виде не очень годится.

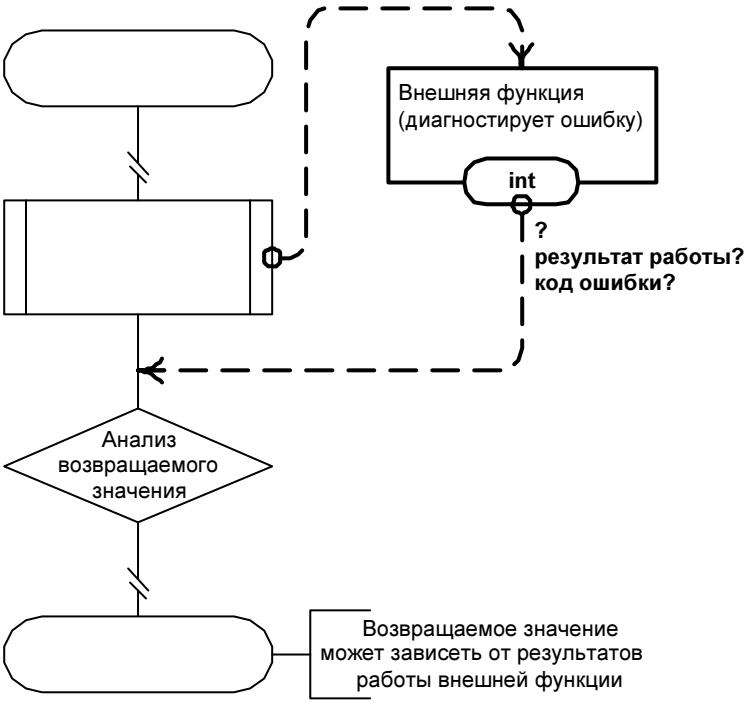


Рис. 6.1. Возвращаемые значения как инструмент обработки ошибок

Использование глобальных переменных или переменных-членов класса, исполняющих роль индикаторов состояния приложения или объекта

Простейшим методом отделения процесса обнаружения ошибки от процесса ее обработки является использование специальных переменных, которые могут быть глобальными (если речь идет об ошибках, связанных с функционированием приложения в целом) или локальными (если речь идет об ошибках, связанных с функционированием отдельных объектов). Фактически, функциональные модули взаимодействуют посредством использования разделяемого ресурса. В случае обнаружения ошибочной ситуации данной переменной присваивается определенное кодовое значение, которое может быть проанализировано за пределами функции, диагностировавшей ошибку. Таким образом, вместо возвращаемого значения (или наряду с ним) используется своего рода обособленный интерфейс для взаимодействия функций в связи с обработкой ошибок (рис. 6.2).

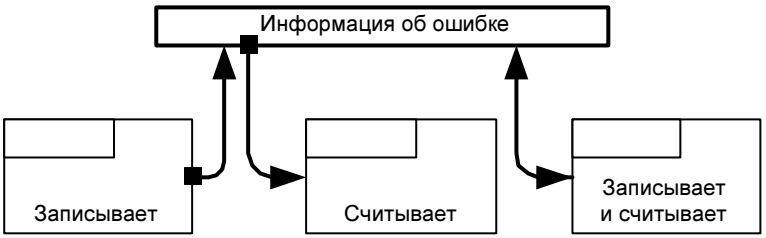


Рис. 6.2. Использование переменных состояния в качестве инструмента обработки ошибок

Пример реализации программы, использующей переменную состояния, иллюстрирует листинг 6.2.

Листинг 6.2. Обработка ошибок на основе использования переменных состояния

```
// ReadData.h
#ifndef _ReadData_h
#define _ReadData_h

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

// Модельная функция чтения (считывает целое число)
void ReadData( ifstream&, int& );

// Коды ошибок при чтении
#define NONE                0x0000
#define PASSED              0x0000
#define UNEXPECTED_EOF      0x0001
#define BAD_FORMAT          0x0002

#endif

// ReadData.cpp
#include "ReadData.h"

int ErrorCode = NONE; // Код ошибки:
                      // начальное состояние: "ошибок нет"

void ReadData( ifstream& fInput, int& x ) {

    char ch;
    fInput >> ch;
    if( fInput.eof() ) {
        ErrorCode = UNEXPECTED_EOF;
```

```
        return;
    }

    fInput.putback( ch );

    fInput >> x;
    if( fInput.fail() ||  fInput.bad() ) {
        ErrorCode = BAD_FORMAT;
        return;
    }

    ErrorCode = NONE;
    return;
}

// Main.cpp
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;

#include "ReadData.h"

ifstream fInput;
ofstream fOutput;

extern int ErrorCode;

void main() {
    int x;

    fInput.open( "input.txt", ios_base::in );
    if( !fInput ) {
        cout << "Error while opening input file" << endl;
        exit( 1 );
    }

    fOutput.open( "output.txt", ios_base::out );
    if( !fOutput ) {
        cout << "Error while opening output file" << endl;
        exit( 1 );
    }

    ReadData( fInput, x );
    switch( ErrorCode ) {

        case NONE :          break;
```

```
        case UNEXPECTED_EOF :// Обработка ошибки
                             // ...
                             break;

        case BAD_FORMAT :    // Обработка ошибки
                             // ...
                             break;
    }
    ErrorCode = PASSED;

    fOutput << "x = " << x << endl;
    fOutput << "Terminated successfully" << endl;

    fOutput.close();
}
```

В случае возможности возникновения нескольких ошибок можно использовать подход с установкой битов ошибок, для анализа которых можно применять поразрядное "И" или специальные функции. Например, пусть определен набор констант, каждая из которых имеет ровно один единичный двоичный разряд:

```
#define EOF_DETECTED         0x0001
#define BAD_FORMAT           0x0002
#define READ_FAILED          0x0004
#define WRITE_FAILED         0x0008
```

Пусть определена переменная состояния, используемая для записи сведений об обнаруженных ошибочных ситуациях:

```
int errorState = 0;
```

Тогда для установки кода ошибки чтения в связи с некорректным форматом данных выполняются следующие действия (рис. 6.3):

```
errorState = BAD_FORMAT | READ_FAILED;
```

Функция проверки ошибки чтения может выглядеть следующим образом:

```
// Возвращает 0, если нет ошибок
// Возвращает ненулевое значение,если зафиксирована ошибка чтения
int readFail() {
    return errorState & READ_FAILED;
}
```

Подобное решение используется в стандартной библиотеке для работы с потоковыми объектами: методы `eof()`, `fail()`, `bad()` анализируют биты ошибок при работе с потоком ввода.

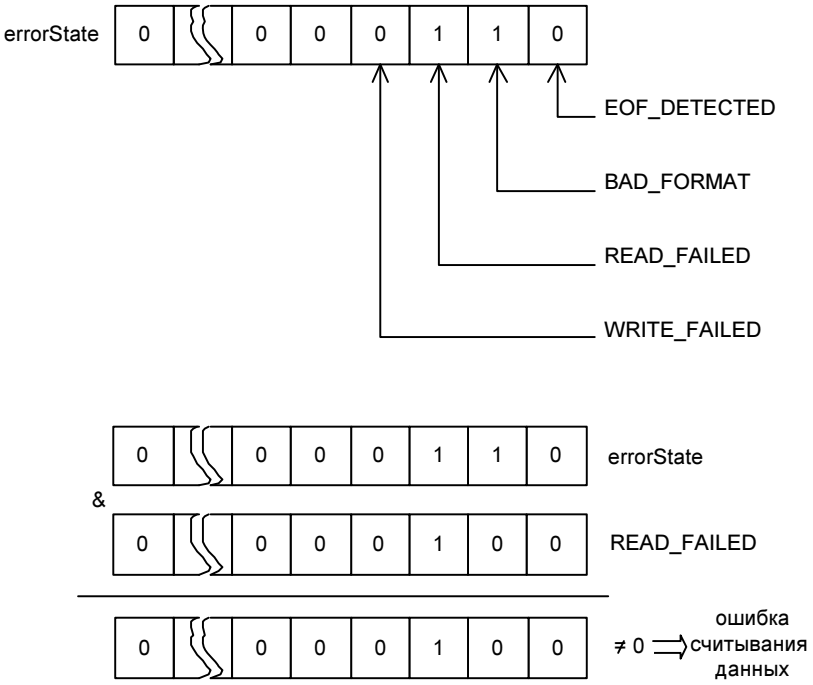


Рис. 6.3. Формирование битов, сигнализирующих об ошибках

Этот подход имеет два основных недостатка. Во-первых, любая глобальная переменная ограничивает степени свободы при разработке отдельных модулей, увеличивая зависимость модулей друг от друга. Во-вторых, нельзя гарантировать, что значение такой переменной будет своевременно проанализировано, и в ходе работы приложения не возникнет новых ошибок в процессе обработки "старых" ошибок.

Использование специально разработанных функций

Относительно простым и во многих случаях подходящим решением является разработка комплекта специальных функций, обеспечивающих обработку ошибок разного типа. В этом случае при возникновении ошибочной ситуации функция, диагностировавшая эту ситуацию, просто вызывает подходящую функцию-обработчик, передавая ей код ошибки. Коды ошибок обычно глобальны на уровне проекта (или компонента) и должны тщательно документироваться. При определенном опыте проектирования разработчику обычно удается реализовать отдельный модуль обработчика ошибок, используемый им (возможно, с незначительными изменениями) во многих проектах. Однако

современные программы редко пишутся разработчиками-одиночками. Кроме того, сложные проекты могут предполагать совместное использование различных компонентов, каждый из которых реализует собственную стратегию обработки ошибок.

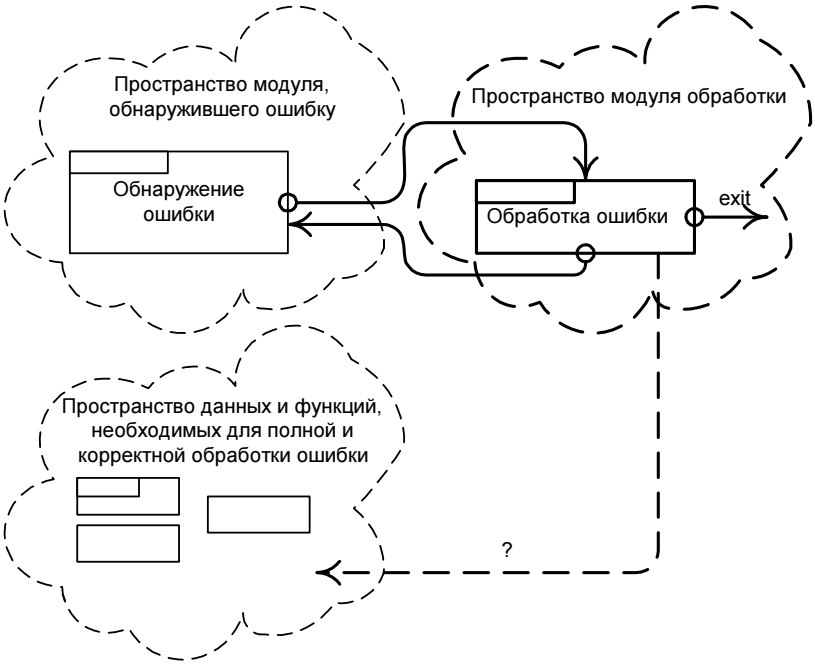


Рис. 6.4. Обработка ошибок с использованием специально разработанных функций

Недостаток данного подхода заключается, прежде всего, в том, что он не очень подходит для реализации многоуровневых систем обработки ошибок, в которых каждый уровень разбирается с теми ошибками, для обработки которых имеет достаточную информацию (рис. 6.4). Однако, поскольку не всегда удается снабдить каждую функцию необходимым числом проверок, гарантирующих ее успешное выполнение, а также в связи с тем, что внешние компоненты могут использовать другие стратегии обработки (что может нарушать тщательно спроектированную многоуровневую модель обработки), этот подход не всегда может обеспечить универсальный характер обработки ошибочных ситуаций.

Использование функций обратного вызова

Для того чтобы обеспечить привязку модуля обработки ошибок к контексту обработки данных, используемых в приложении, можно использовать известный подход, основанный на так называемых функциях обратного вызова.

Пользователь функции-обработчика ошибок в этом случае должен сначала зарегистрировать функцию обратного вызова, определенную в том модуле, в котором имеется доступ к данным, необходимым для корректной и полной обработки возникшей ситуации, например, так, как это представлено в листинге 6.3.

Листинг 6.3. Определение, регистрация и использование функций обратного вызова при обработке ошибок

```
// Error.h
#ifndef _Error_h_
#define _Error_h_

// Тип функции обратного вызова (указатель на функцию):
//     в данном примере не имеет параметров;
//     не возвращает значения
typedef void ( *CallbackTermination )();

// Функция обработки ошибки
void error( int, CallbackTermination = 0 );

#endif;

// Error.cpp
#include <cstdlib>
#include <iostream>
using namespace std;
#include "Error.h"

// Сообщения в связи с обработкой ошибок
static char *msg[ 5 ] = {
    /* 0 */ "",
    /* 1 */ "Error while opening input file",
    /* 2 */ "Error while opening output file",
    /* 3 */ "Input file contains no data",
    /* 4 */ "Wrong input data format"
};

// Функция обработки ошибки
void error( int code, // Код ошибки
            CallbackTermination CallbackFunction
                      // Внешний обработчик
                      // (указатель на функцию
                      //  обратного вызова)
          ) {
```

```
    cout << "Error No. " << code << ": ";
    cout << msg[ code ] << endl;

    if( CallbackFunction == 0 ) exit( 0 );

    CallbackFunction(); // Вызов внешнего обработчика
}

// Main.cpp
#include <cstdlib>
#include <fstream>
using namespace std;

#include "Error.h"

void ReadData( ifstream&, int& );

// Функции обратного вызова
void Terminate1(); // Первый вариант завершения
void Terminate2(); // Второй вариант завершения

ifstream fInput;
ofstream fOutput;

void main() {
    int x;

    fInput.open( "input.txt", ios_base::in );
    if( !fInput ) {
        error( 1 ); // Вызывается обработчик ошибок
                    // без передачи указателя
                    // на функцию обратного вызова
                    // (никаких дополнительных действий
                    //  не требуется)
    }

    fOutput.open( "output.txt", ios_base::out );
    if( !fOutput ) {
        error( 2 );
    }

    ReadData( fInput, x );

    fOutput << "x = " << x << endl;
    fOutput << "Terminated successfully" << endl;

    fOutput.close();
}
```

```
void ReadData( ifstream& fInput, int& x ) {

    char ch;
    fInput >> ch;
    if( fInput.eof() ) {
        // Вызывается обработчик ошибок:
        // в качестве внешнего обработчика указывается
        // функция Terminate1()
        error( 3, Terminate1 );
    }

    fInput.putback( ch );

    fInput >> x;
    if( fInput.fail() ||  fInput.bad() ) {
        // Вызывается обработчик ошибок:
        // в качестве внешнего обработчика указывается
        // функция Terminate2()
        error( 4, Terminate2 );
    }
}

// Реализация первого варианта завершения программы
void Terminate1() {
    fOutput << "Unexpected end of input file" << endl;
    fOutput << "Terminated with error" << endl;

    // Обратите внимание, что доступа к fInput и fOutput
    // функция error() не имеет
    fInput.close();
    fOutput.close();

    exit( 1 );
}

// Реализация второго варианта завершения программы
void Terminate2() {
    fOutput << "Input data wrong format" << endl;
    fOutput << "Terminated with error" << endl;

    fInput.close();
    fOutput.close();

    exit( 2 );
}
```

При вызове функции обработки ошибки `error()` ей, наряду с кодом ошибки, передается информация об адресе функции обратного вызова, ответственной за ту часть обработки ошибки, которая не может быть по каким-либо причинам реализована в пределах функции `error()` (рис. 6.5).

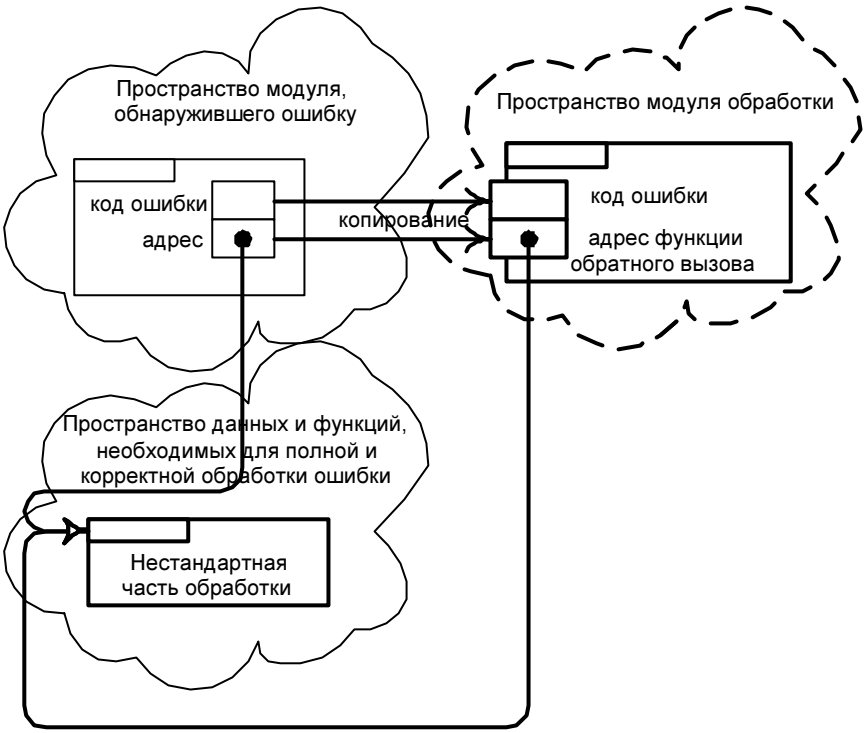


Рис. 6.5. Использование функции обратного вызова
как элемента реализации двухуровневой обработки ошибки

Механизм с использованием функций обратного вызова частично разрешает проблему "разграничения полномочий" при обработке ошибок, однако вряд ли может считаться полноценной реализацией многоуровневой модели. Сигнатура функций обратного вызова должна в точности соответствовать сигнатуре, объявленной в интерфейсном файле модуля обработки ошибок (в рассмотренном примере эту роль выполняет файл Error.h). Использование функций с переменным числом параметров сильно усложнит реализацию, а определение различных сигнатур функций обратного вызова с последующей перегрузкой функции `error()`, очевидно, не является унифицирующим решением.

6.2. Обработка исключений

Настоящая унификация возможна в том случае, когда механизм обработки ошибок встроен в язык программирования. Современные объектно-ориентированные языки используют в качестве такого механизма подход, основанный на обработке исключений. При этом следует заметить, что впервые поддержка обработки исключений средствами языка была реализована в языке PL/1 (см., например, [Фролов, 1987]). Несмотря на то, что сегодня многие специалисты считают ту реализацию неудачной из-за трудностей в понимании и, как следствие, в правильном использовании, следует отметить, что обработка исключительных ситуаций в PL/1 реализована довольно гибко. Программисту предоставляются как встроенные типы исключений (например, исключения, вызываемые переполнением при выполнении арифметических операций, исключения, вызываемые выходом индекса массива за пределы корректного диапазона), так и возможности для определения собственных исключительных ситуаций.

Основная идея подхода

Основная идея обработки исключений заключается в следующем (рис. 6.6). Пусть некоторая функция в иерархии вызовов (например, функция `F2()`) передает управление другой функции `F3()`. В ходе работы функции `F3()` выявляется некоторая ошибочная ситуация (т. е. функция `F3()` обнаруживает ошибку). Функция, обнаружившая ошибку, генерирует (throws) специальный объект, описывающий возникшую исключительную ситуацию. Иными словами, функция `F3()` создает объект-исключение. Выполнение функции, обнаружившей ошибку, на этом прекращается, и управление передается в вызвавшую ее функцию (в нашем случае — в функцию `F2()`) для реализации действий по преодолению возникшей проблемы. В этом случае говорят, что функция `F2()` перехватывает (catches) исключение (рис. 6.6). При этом необходимо заметить, что обработкой исключения может заниматься как функция, непосредственно вызвавшая функцию-генератор исключения, так и предшествующие в иерархии вызовов функции (например, функция `F1()`).

Таким образом, с одной стороны, механизм обработки исключений способствует решению проблемы отделения кода, диагностирующего ошибку, от кода, обрабатывающего ошибку, некоторым единым способом, поддерживаемым языком.

С другой стороны, исключения предоставляют возможность коду, обнаружившему проблему, последствия которой не могут быть преодолены на месте, передать эту проблему в другую часть системы. В частности, исключения предоставляют инструмент, позволяющий сообщить об ошибке, произошед-

шей в ходе выполнения конструктора. При этом следует заметить, что генерация исключений в конструкторе не лишена некоторых подводных камней *(см. подраздел "Исключения в конструкторах и деструкторах" разд. 6.3)*.

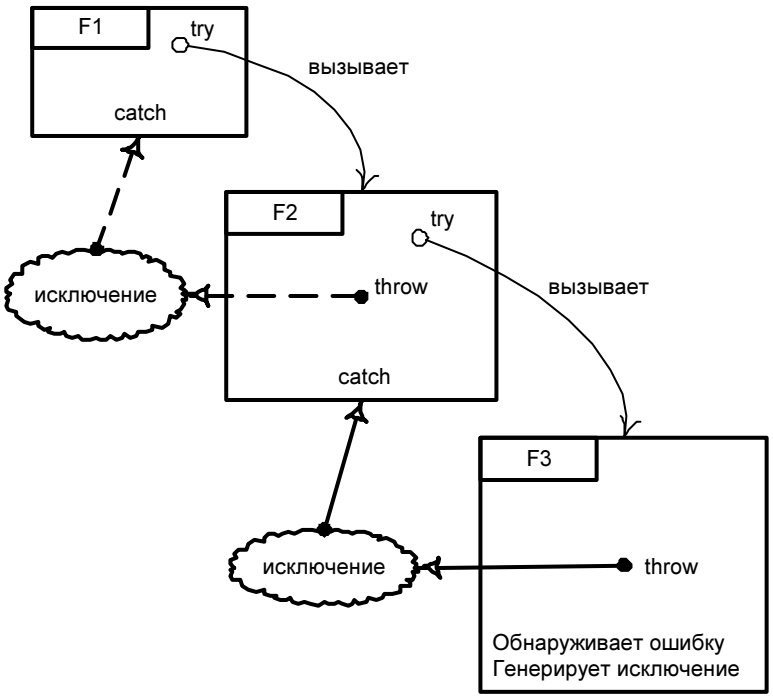


Рис. 6.6. Генерация и перехват исключений

В разговоре об исключениях заслуживает быть полностью процитированным следующий абзац из книги Страуструпа "Язык программирования C++":

"Надо понимать, что обработка ошибок остается сложной задачей и что механизм обработки исключений — несмотря на бóльшую формализацию, чем альтернативные методы — относительно менее структурирован по сравнению со средствами языка, обеспечивающими локальное управление выполнением. Механизм обработки исключений C++ предоставляет программисту средство обработки ошибок в том месте, где их наиболее естественно обрабатывать при данной структуре системы. Исключения делают сложность обработки ошибок более явной [не вполне преодолевая ее — Е. П.]. Однако исключения не являются причиной этой сложности. Не обвиняйте того, кто принес плохую новость" [Stroustrup, 2000, разд. 14.1].

Обработка исключений — тоже в своем роде механизм абстракции, поскольку позволяет абстрагироваться в основном коде от потенциальных проблем,

не концентрируя все внимание на обнаружении и обработке ошибок [Дубовцев, 2004].

Что такое "исключение" и как оно образуется

Как было отмечено выше, генерация исключения представляет собой создание объекта-исключения. Объект характеризуется типом, например:

```
// Класс, определяющий тип объекта-исключения
class Exception {
    // ...
public:
    Exception() { // Конструктор
        // ...
    }
};
```

Процесс генерации и обработки исключения графически представлен на рис. 6.7.

Блок, обнаруживающий ошибку, инициирует процесс создания объекта-исключения с использованием специального синтаксиса:

```
// Функция, обнаруживающая ошибку и генерирующая исключение
void F3(){
    if( /* ошибка... */ ) {
        throw Exception(); // Генерирует исключение
    }
    // Действия при отсутствии ошибок
    // ...
}
```

Вызывающая функция может перехватить исключение и выполнить обработку ошибки:

```
// Функция, перехватывающая исключение
void F2(){
    try {
        F3(); // Может генерировать исключение
    }
    catch( Exception ) {
        // Обработка ошибки
        // ...
    }
}
```

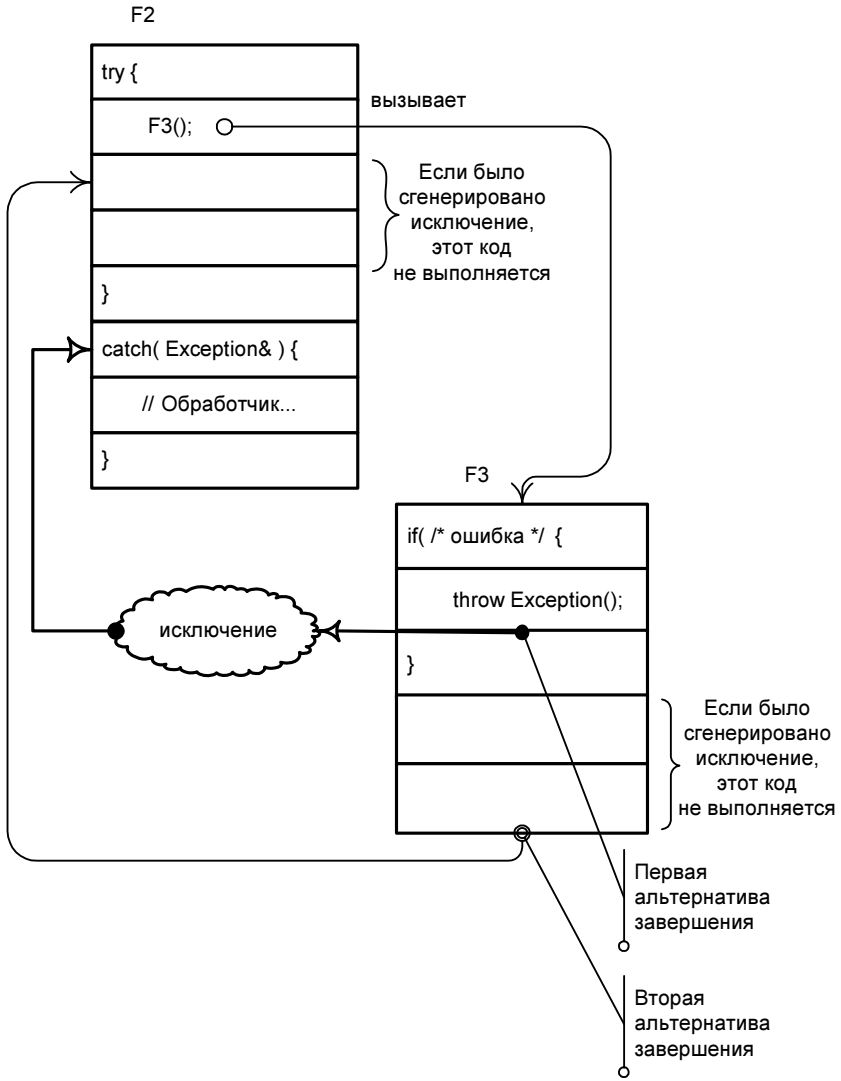


Рис. 6.7. Управление действиями программы с учетом возможности возникновения исключения

Блок `try` включает обращение к функциям, которые могут генерировать исключения. Поскольку обработка исключения осуществляется в блоке `catch`, то именно он и называется обычно обработчиком исключения. Если в контексте работы функции `F2()` исключение не может быть обработано полностью, эта функция может инициировать повторное исключение с тем, чтобы оно было обработано на следующих уровнях иерархии вызовов (см. также рис. 6.6).

```
// Функция, перехватывающая и повторно
// инициирующая исключение
void F2(){
    try {
        F3(); // Может генерировать исключение
    }
    catch( Exception ) {
        // Частичная обработка ошибки
        //...
        throw; // Повторно генерирует исключение
    }
}
```

Если функция `F3()` может генерировать несколько типов исключений, то используется несколько последовательно записываемых друг за другом обработчиков исключений, которые в этом случае подобны перегруженным функциям, однако это подобие весьма условно *(см. подраздел "Порядок записи обработчиков" в разд. 6.3)*. При перехвате исключения выбирается нужный обработчик в зависимости от типа сгенерированного исключения.

Таким образом, общая схема генерации и обработки исключения выглядит следующим образом. Функция, диагностировавшая ошибку, генерирует объект исключения с помощью инструкции `throw`. Далее происходит раскручивание стека приложения до тех пор, пока не будет обнаружен подходящий обработчик исключения. Еще раз обращаем внимание на то, что блок, начинающийся с ключевого слова `try`, и подходящий обработчик не обязательно располагаются в функции, непосредственно вызвавшей код, сгенерировавший исключение.

Исключения следует перехватывать по ссылке

При возникновении исключительной ситуации исполнение функции, инициировавшей объект исключения, прекращается, поэтому, для того чтобы объект исключения не был потерян, создается копия объекта, которая и связывается с параметром обработчика `catch`. При следующей за этим передаче значения объекта-исключения в обработчик создается еще одна копия. Таким образом, копирование объекта-исключения происходит дважды. Данную ситуацию иллюстрирует программа, представленная в листинге 6.4.

Листинг 6.4. Двойное копирование объекта-исключения

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

```
class Exception {
public:
    Exception() {
        cout << "Exception constructed" << endl;
    }
    Exception( const Exception& ) {
        cout << "Exception copied" << endl;
    }
    void Check() {
        cout << "Exception checked" << endl;
    }
};

void F3() {
    throw Exception();
}

void F2() {
    try {
        F3();
    }
    catch( Exception e ) {
        cout << "Exception caught in F2 and rethrown" << endl;
        throw e;
    }
}

void F1() {
    try {
        F2();
    }
    catch( Exception e ) {
        cout << "Exception caught in F1" << endl;
        e.Check();
    }
}

void main() {
    F1();
}
```

В программе листинга 6.4 класс исключения снабжен конструктором копирования, наличие которого позволяет наблюдать за возникновением копий объекта исключения. В результате исполнения программы будут выведены следующие сообщения:

```
Exception constructed
Exception copied
Exception copied
Exception caught in F2 and rethrown
Exception copied
Exception copied
Exception caught in F1
Exception checked
```

Данный вывод наглядно демонстрирует, что при генерации и последующем перехвате объекта-исключения по значению (равно как и при повторной генерации) объект класса `Exception` копируется дважды.

Для того чтобы избежать повторного копирования, исключения следует перехватывать по ссылке (листинг 6.5).

Листинг 6.5. Перехват исключений по ссылке

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

class Exception {
public:
    Exception() {
        cout << "Exception constructed" << endl;
    }
    Exception( const Exception& ) {
        cout << "Exception copied" << endl;
    }
    void Check() {
        cout << "Exception checked" << endl;
    }
};

void F3() {
    throw Exception();
}

void F2() {
    try {
        F3();
    }
    catch( Exception& e ) {
        cout << "Exception caught in F2 and rethrown" << endl;
```

```
        throw e;
    }
}

void F1() {
    try {
        F2();
    }
    catch( Exception& e ) {
        cout << "Exception caught in F1" << endl;
        e.Check();
    }
}

void main() {
    F1();
}
```

На этот раз вывод программы свидетельствует об отсутствии повторного копирования объекта-исключения:

```
Exception constructed
Exception copied
Exception caught in F2 and rethrown
Exception copied
Exception caught in F1
Exception checked
```

Таким образом, следует организовывать перехват исключений по ссылке.

Преимущества встраивания в язык обработки исключительных ситуаций

Себеста отмечает следующие основные преимущества встраивания в язык обработки исключительных ситуаций [Sebesta, 2002]:

1. Без обработки исключений код, необходимый для обнаружения ошибок, может сильно запутать программу. Вирт приводит пример с выражением, содержащим множество операций деления, каждая из которых может иметь знаменатель, равный нулю. Без встроенной обработки исключений каждой из этих операций должна предшествовать проверка. Наличие в языке встроенной обработки исключений позволяет компилятору генерировать код, содержащий такие проверки по требованию программиста (отметим, впрочем, что C++ не генерирует исключения при возникновении ошибок вычисления арифметических операций).

2. Возможность передавать исключительные ситуации для обработки в другие модули.
3. Язык поощряет программиста рассматривать все события, которые могут возникнуть в процессе выполнения программы, а не игнорировать их в надежде на то, что "ничего плохого не случится".

Некоторые важные подробности реализации обработки исключений в C++ обсуждаются в разд. 6.3.

6.3. Особенности обработки исключений в языке C++

Несмотря на общность идей, лежащих в основе реализаций механизма обработки исключений в разных объектно-ориентированных языках, некоторые детали реализаций могут различаться весьма существенно. Начнем с рассмотрения особенностей реализации обработки исключений в языке C++.

Типы исключений

Язык C++ позволяет использовать в качестве типов исключений встроенные типы языка и типы, определенные пользователем (в том числе типы исключений из стандартной библиотеки C++). Рассмотрим оба этих варианта подробнее.

Использование встроенных типов языка

Использование встроенных типов в качестве типов исключений очень просто в реализации, однако является не очень хорошей практикой, т. к. обычно ведет к различению диагностируемых ошибочных ситуаций не по типу исключения, а по значению объекта исключения, например (листинг 6.6).

Листинг 6.6. Использование встроенных типов в качестве типов исключений

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

void ReadData( ifstream& fInput, int& x ) {

    char ch;
    fInput >> ch;
    if( fInput.eof() )
        throw( 1 ); // Генерация экземпляра
                    // исключения типа int
```

```
    fInput.putback( ch );

    fInput >> x;
    if( fInput.fail() ||  fInput.bad() )
        throw( 2 ); // Генерация экземпляра
                    // исключения типа int
}

void UseData( ifstream& fInput ) {
    int x;
    try {
        ReadData( fInput, x );
        // Use data...
    }
    catch( int exception ) { // Перехват исключений
                             // типа int
        // Анализ значения
        switch( exception ) {
            case 1 : // Обработка ошибки пустого файла
                     // ...
                     break;
            case 2 : // Обработка ошибки формата данных
                     // ...
                     break;
        }
    }
    // ...
}

void main() {

    ifstream fInput( "input.txt", ios_base::in );
    if( !fInput ) {
        cout << "Error while opening input file" << endl;
        return;
    }

    UseData( fInput );
}
```

В маленькой программе такая практика не создаст серьезных проблем (за исключением явного нарушения стиля, ибо в таком виде обработка исключений мало отличается от анализа возвращаемого значения). Однако в больших проектах подобная "плоская" структура представления исключений быстро приведет к выходу управления вычислительным процессом при обнаружении

ошибочных ситуаций из-под контроля программиста. Нет гарантии, что разные обработчики не смешаются друг с другом. Невозможность различения типов исключений не позволяет написать полноценные самостоятельные обработчики исключений для разных типов. Наконец, при использовании в качестве носителей типов исключений встроенных типов языка не получится организовать иерархию типов исключений, для которых можно было бы определять обработчики по принципу "от общего к частному".

Использование типов, определяемых пользователем

Предпочтительнее использовать в качестве типов исключений типы, определяемые пользователем (листинг 6.7).

Листинг 6.7. Типы исключений, определяемые программистом

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

class EmptyFileException {};
    // Тип-исключение "пустой файл"
class BadFormatException {};
    // Тип-исключение "ошибка формата данных"

void ReadData( ifstream& fInput, int& x ) {

    char ch;
    fInput >> ch;
    if( fInput.eof() )
        throw EmptyFileException();
            // Генерация экземпляра исключения
            // типа EmptyFileException

    fInput.putback( ch );

    fInput >> x;
    if( fInput.fail() ||  fInput.bad() )
        throw BadFormatException();
            // Генерация экземпляра исключения
            // типа BadFormatException
}

void UseData( ifstream& fInput ) {
    int x;
    try {
        ReadData( fInput, x );
```

```
        // Use data...
    }
    catch( EmptyFileException& ) {
        // Обработка ошибки пустого файла
        cout << "Empty file exception" << endl;
        cout << "Terminated with error" << endl;
        // ...
    }
    catch( BadFormatException& ) {
        // Обработка ошибки формата данных файла
        cout << "Bad format exception" << endl;
        cout << "Terminated with error" << endl;
        // ...
    }
    //...
}

void main() {

    ifstream fInput( "input.txt", ios_base::in );
    if( !fInput ) {
        cout << "Error while opening input file" << endl;
        return;
    }

    UseData( fInput );
}
```

В классе исключения можно реализовать часть функциональности в связи с обработкой данного типа исключения, например, получение описания исключительной ситуации (листинг 6.8).

Листинг 6.8. Реализация в классе исключения элементов функциональности в связи с обработкой исключения

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

class EmptyFileException {
public:
    const char *what() {
        return "Attempt to read data from empty file";
    }
};
```

```
class BadFormatException {
public:
    const char *what() {
        return "Wrong data format";
    }
};

void ReadData( ifstream& fInput, int& x ) {
    // см. листинг 6.7
    // ...
}

void UseData( ifstream& fInput ) {
    int x;
    try {
        ReadData( fInput, x );
        // ...
    }
    catch( EmptyFileException& e ) {
        // Обработка ошибки пустого файла

        // Вывод сообщения
        cout << e.what() << endl;

        // Дополнительные действия
        // ...
    }
    catch( BadFormatException& e ) {
        // Обработка ошибки формата данных файла

        // Вывод сообщения
        cout << e.what() << endl;

        // Дополнительные действия
        // ...
    }
    // ...
}

void main() {

    // см. листинг 6.7
    // ...
}
```

Как и другие классы, классы исключений могут иметь конструкторы с параметром (листинг 6.9).

Листинг 6.9. Реализация класса исключения, содержащего конструктор с параметром

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

class EmptyFileException {
public:
    const char *what() {
        return "Attempt to read data from empty file";
    }
};

class BadFormatException {
    const char *cause; // Описание причины, вызвавшей
                       // исключение
public:
    BadFormatException( const char *_cause = "" ) {
        cause = _cause;
    }
    // Получение информации о содержании исключения
    const char *what() {
        return "Wrong data format";
    }

    // Получение информации о причине исключения
    const char *caused() {
        return cause;
    }
};

void ReadData( ifstream& fInput, int& x ) {

    char ch;
    fInput >> ch;
    if( fInput.eof() )
        throw EmptyFileException();
            // Генерация экземпляра исключения
            // типа EmptyFileException

    fInput.putback( ch );

    fInput >> x;
    if( fInput.fail() ||  fInput.bad() )
        throw BadFormatException( "Not integer" );
```

```
            // Генерация экземпляра исключения
            // типа BadFormatException
            // с использованием аргумента конструктора

    // ...
}

void UseData( ifstream& fInput ) {
    int x;
    try {
        ReadData( fInput, x );
        // Use data...
    }
    catch( EmptyFileException& e ) {
        // Обработка ошибки пустого файла

        // Вывод сообщения
        cout << e.what() << endl;

        // Дополнительные действия
        // ...
    }
    catch( BadFormatException& e ) {
        // Обработка ошибки формата данных файла

        // Вывод сообщения
        cout << e.what() << endl;
        cout << "Explanation: " << e.caused() << endl;

        // Дополнительные действия
        // ...
    }
}

void main() {

    // см. листинг 6.7
    // ...
}
```

Если поместить в исходный файл input.txt текст, не являющийся целым числом, то в результате исполнения программы будет получен следующий вывод:

```
Wrong data format
Explanation: Not integer
```

Группирование исключений

Мощным средством поддержки модели многоуровневой обработки исключений является группирование исключений, основанное на выделении общих свойств для представления на более высоком уровне абстракции (листинг 6.10).

Листинг 6.10. Проектирование иерархий исключений

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

// Базовый (наиболее общий) тип исключения
class ReadFileException {
public:
    virtual const char *what() {
        return "Some error while reading file";
    }
};

// Производные (частные) типы исключений
class EmptyFileException : public ReadFileException {
public:
    const char *what() {
        return "Attempt to read data from empty file";
    }
};
class BadFormatException : public ReadFileException {
public:
    const char *what() {
        return "Wrong data format";
    }
};
class CorruptedFileException : public ReadFileException {
public:
    const char *what() {
        return "Wrong file structure";
    }
};
// ...

void ReadData( ifstream& fInput, int& x ) {
    // см. листинг 6.7
    // ...
}
```

```
void UseData( ifstream& fInput ) {
    int x;
    try {
        ReadData( fInput, x );
        // ...
    }
    catch( BadFormatException& e ) {
        // Обработка ошибки некорректного формата

        // Вывод сообщения
        cout << e.what() << endl;

        // Дополнительные действия
        // ...
    }
    catch( ReadFileException& e ) {
        // Обработка других ошибок считывания из файла

        // Вывод сообщения
        cout << e.what() << endl;

        // Дополнительные действия
        // ...
    }
    // ...
}

void main() {

    // см. листинг 6.7
    // ...
}
```

Обратите внимание, что при этом передача объекта-исключения по ссылке позволяет воспользоваться полиморфным поведением объектов, если такое поведение допускает определение базового класса иерархии исключений (т. е. если в базовом классе иерархии есть виртуальные функции).

Организация определений классов исключений в виде иерархии имеет большое значение для надежности и модифицируемости кода. Наличие в рассмотренном примере обработчика исключения базового класса `ReadFileException` позволяет гарантировать, что никакое исключение из данной иерархии не будет пропущено, даже если у класса `ReadFileException` появятся новые производные классы.

Итак, приведенные примеры демонстрируют, что объект исключения может передаваться обработчику как по значению, так и по ссылке.

Если объект-исключение размещается в динамической памяти с помощью операции new, то он передается обработчику по значению указателя. Предыдущий пример, переработанный таким образом, чтобы объекты исключения передавались обработчику по значению указателя, будет выглядеть примерно так (обратите внимание на то, как теперь выглядит генерация исключений) (листинг 6.11).

Листинг 6.11. Связывание с объектами исключений посредством указателей

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

// Определения классов исключений см. в листинге 6.10
// ...

void ReadData( ifstream& fInput, int& x ) {

    char ch;
    fInput >> ch;
    if( fInput.eof() )
        throw new EmptyFileException();

    fInput.putback( ch );

    fInput >> x;
    if( fInput.fail() ||  fInput.bad() )
        throw new BadFormatException();
}

void UseData( ifstream& fInput ) {
    int x;
    try {
        ReadData( fInput, x );
        // Use data...
    }
    catch( BadFormatException *e ) {
        // Обработка ошибки некорректного формата

        // Вывод сообщения
        cout << e->what() << endl;

        // Дополнительные действия
        // ...
    }
```

```
    catch( ReadFileException *e ) {
        // Обработка других ошибок считывания из файла

        // Вывод сообщения
        cout << e->what() << endl;

        // Дополнительные действия
        // ...
    }
    // ...
}
void main() {

    // см. листинг 6.7
    // ...
}
```

Вообще говоря, передача исключений по значению указателя не свободна от проблем в связи с управлением памятью, занимаемой объектом исключения. В общем случае, внутри обработчика исключения нет ясности относительно того, следует ли освобождать память, связанную с указателем на объект исключения, поскольку нельзя гарантировать, что указатель действительно указывает на объект, размещенный в динамической памяти [Meyers, 1996, Item 13]. Таким образом, остается лишь еще раз повторить ранее сделанный вывод: по крайней мере, в том, что касается кода программы, находящегося под вашим контролем, перехватывайте исключения по ссылке.

Обработчики исключений

В данном разделе обсуждается взаимное расположение обработчиков исключений в исходном тексте программы и организация многоуровневой обработки исключений.

Порядок записи обработчиков

Так как производные исключения могут перехватываться обработчиками, предназначенными для обработки базовых исключений, то порядок записи обработчиков очень важен. В общем случае справедливо правило: чем больше типов исключений может перехватывать обработчик, тем дальше он должен быть расположен от блока `try` *(примеры см. в предыдущем подразделе "Группирование исключений")*.

Повторная генерация исключения

Как уже было отмечено выше, повторная генерация исключения применяется тогда, когда исключение не может быть полностью обработано в контексте

данной функции. Для повторной генерации исключения применяется инструкция `throw` без операнда.

Отличие повторной генерации исключения от генерации исключения указанного типа иллюстрирует пример, представленный в листинге 6.12.

Листинг 6.12. Типы объектов при повторной генерации исключения и при генерации исключения указанного типа

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

class ReadFileException {
public:
    virtual const char *what() {
        return "Read file exception";
    }
};
class EmptyFileException : public ReadFileException {
public:
    const char *what() {
        return "Empty file exception";
    }
};
class BadFormatException : public ReadFileException {
public:
    const char *what() {
        return "Bad format exception";
    }
};


void ThrowsEmptyFileException( void ) {
    throw EmptyFileException();
}
void ThrowsBadFormatException( void ) {
    throw BadFormatException();
}

void Check1( void ) {
    try {
        ThrowsEmptyFileException();
    }
    catch( EmptyFileException& e ) {
        cout << e.what() << endl;
```

```
        throw ReadFileException();
    }
}
void Check2(void) {
    try {
        ThrowsBadFormatException();
    }
    catch( ReadFileException& e ) {
        cout << e.what() << endl;
        throw; // или throw e;
    }
}

void main() {

    try {
        Check1();
    }
    catch( ReadFileException& e ) {
        cout << e.what() << endl;
    }
    try {
        Check2();
    }
    catch( ReadFileException& e ) {
        cout << e.what() << endl;
    }
}
```

В программе определены три класса исключений, причем класс `ReadFileException` является базовым классом, а классы `EmptyFileException` и `BadFormatException` — производными от него. Поскольку функция `what()` — виртуальная, объекты классов-исключений поддерживают полиморфное поведение. Таким образом, при использовании ссылок на объекты всегда будет вызываться версия `what()`, объявленная в том классе, объект которого используется.

Далее определены две функции, гарантированно генерирующие исключения. Функция `ThrowsEmptyFileException()` в точном соответствии со своим наименованием генерирует исключение класса `EmptyFileException`. Функция `ThrowsBadFormatException()` генерирует исключение класса `BadFormatException`.

Функция `Check1()` вызывает функцию, генерирующую исключение `EmptyFileException`, и перехватывает его. На консоль выводится сообщение:

```
Empty file exception
```

Далее генерируется исключение класса `ReadFileException`, перехватываемое уже в функции `main()`, о чем свидетельствует печатаемое сообщение:

```
Read file exception
```

Собственно, до этого момента никаких новых особенностей этот пример не иллюстрирует. Далее начинает работать функция `Check2()`, перехватывающая исключение класса `BadFormatException`, генерируемое функцией `ThrowsBadFormatException()`. Функция `Check2()` умышленно написана таким образом, что исключение перехватывается обработчиком с параметром `ReadFileException& e`. Тем не менее, благодаря полиморфизму, на консоль выводится сообщение, подтверждающее, что в памяти находится объект класса `BadFormatException`, производного от `ReadFileException`:

```
Bad format exception
```

Далее функция `Check2()` повторно генерирует исключение посредством инструкции `throw`. Это исключение перехватывается в функции `main()`, при этом на консоль печатается следующее сообщение:

```
Bad format exception
```

Итак, ссылка `e` во втором обработчике в функции `main()` ссылается на объект класса `BadFormatException`, несмотря на то, что повторная генерация осуществлялась в блоке обработчика с параметром `ReadFileException& e`. Это означает, что при повторной генерации выбрасывается такой же объект, что и был создан первоначально при генерации исключения функцией `ThrowsBadFormatException()`. Ничего не изменится, если вместо `throw;` с "подразумеваемым" `e` явно записать:

```
throw e;
```

Таким образом, в целом вывод программы имеет следующий вид:

```
Empty file exception
Read file exception
Bad format exception
Bad format exception
```

Использование обработчика *catch(...)*

Обработчик данного вида позволяет перехватить исключения любого типа. Такой обработчик можно использовать, когда содержание обработки ошибки не зависит от типа исключения. Однако не следует увлекаться практикой использования таких "универсальных обработчиков" вместо нескольких отдельных обработчиков. Рихтер отмечает, что если задуматься над смыслом данной конструкции, то она означает, что мы пытаемся убедить компилятор

в способности написать код, обрабатывающий любое исключение, известное нам или неизвестное [Richter, 2002]. Но это неверно и может служить источником трудно обнаруживаемых ошибок. Поэтому следует минимизировать использование таких обработчиков. Возможно, лучше позволить обработать исключение исполнительной системе (и мы, по крайней мере, увидим, что нештатная ситуация возникла), чем скрыть за подобной конструкцией отсутствующую обработку или обработку с ошибками.

Область применения обработчика `catch(...)`, таким образом, ограничена следующими типовыми ситуациями *(см. также упражнения в разд. 6.5)*:

- освобождение занятых ресурсов и повторная генерация исключения [Stroustrup, 2000, разд. 14.3.2];
- повторная генерация исключений, которые не могут быть обработаны данным исключением (обработчик `catch(...)` должен быть записан последним) [Stroustrup, 2000, разд. 14.3.2.1];
- обработка исключений в случае, если функция, генерирующая исключение, нарушила контракт, в соответствии с которым может генерировать только определенные виды исключений [Stroustrup, 2000, разд. 14.6].

Когда исключение считается обработанным?

В соответствии с подходом, реализованным в C++, исключение считается обработанным сразу после входа в обработчик. Поэтому нижеследующая конструкция не приводит к бесконечному циклу:

```
try {
    // ...
}
catch( Exception& ) {
    // ...
    throw Exception();
}
```

Обратите внимание, генерация исключения типа `Exception` в данном обработчике не трактуется как повторная генерация исключения (см. листинг 6.10).

Исключения в связи с ошибкой выделения памяти

До появления в C++ механизма обработки исключений, в случае, если выделение памяти завершалось неудачей, в результате операции `new` формировалось значение 0. В соответствии с требованиями современного стандарта C++ при ошибке выделения памяти `new` по умолчанию генерирует исключение `bad_alloc`. Исключение `bad_alloc`, равно как и соответствующая реализация `new`, определено в заголовочном файле new в пространстве имен `std`.

Поэтому рекомендации стандарта C++ предписывают преобразовывать код, использующий механизм с анализом значения указателя, к форме, ориентированной на обработку исключений. Если по каким-либо причинам желательно сохранить прежний стиль обработки ошибки резервирования динамической памяти, можно изменить программу, используя распределитель nothrow, при котором исключение не генерируется. Обе возможности иллюстрирует пример, представленный в листинге 6.13.

Листинг 6.13. Ошибки выделения памяти с исключением и без исключения

```
#include <iostream>
#include <climits>
#include <new> // Подключает библиотеку, обеспечивающую реализацию
               // new с генерацией исключения bad_alloc
               // в случае ошибки
using namespace std;

// Функция запрашивает слишком много динамической памяти
void AskLotOfMemory() {
    int *p;
    p = new int[ UINT_MAX ];
}

void main() {

    try {
        AskLotOfMemory();
    }
    catch( bad_alloc& e ) {
        // Будет напечатано сообщение "bad allocation"
        cout << e.what() << endl;
    }

    // Использование nothrow подавляет генерацию исключения
    int *p = new ( nothrow ) int[ UINT_MAX ];
    if( p == 0 ) {
        // Будет напечатано сообщение "no exception thrown"
        cout << "no exception thrown" << endl;
    }
}
```

В результате исполнения программы на консоль будут выведены следующие сообщения:

```
bad allocation
no exception thrown
```

В первом случае (при резервировании динамической памяти в функции `AskLotOfMemory()`) реализуется подразумеваемое стандартом поведение с генерацией исключения `bad_alloc`, перехватываемого в функции `main()`. Если вы работаете в интегрированной среде программирования, то в режиме отладки возникшая исключительная ситуация (которая трактуется как опасная), возможно, будет перехвачена отладчиком. Например, при отладке в Microsoft Visual Studio появится окно **Debug Assertion**. Для того чтобы убедиться, что программа работает, нажмите кнопку **Ignore**, дав программе возможность завершить работу, и убедитесь в том, что исключение успешно перехватывается.

Во втором случае (фрагмент резервирования памяти в функции `main()`) использование `nothrow` подавляет генерацию исключения, и работает "старый" механизм, подразумевающий необходимость анализа значения указателя `p`.

Исключения в конструкторах и деструкторах

Поскольку конструкторы и деструкторы не имеют возвращаемого значения, исключения предоставляют, по сути, единственный целостный механизм организации полной обработки ошибки, произошедшей в ходе работы конструктора или деструктора.

Конструкторы и исключения

Наиболее актуальны два случая возникновения исключений в конструкторах.

Первый вариант связан с обнаружением ошибочной ситуации непосредственно в конструкторе, т. е. инструкция `throw` находится в теле конструктора. Если до обнаружения ошибки в конструкторе осуществлялся захват каких-либо ресурсов (например, резервирование динамической памяти), следует восстановить исходное положение до вызова конструктора с целью избежания утечек памяти.

Примечание

Обратите внимание, что это один из немногих случаев, когда в пределах одного функционального блока могут встретиться все три элемента механизма обработки исключений (`try`, `throw` и `catch`) применительно к одному объекту исключения!

Листинг 6.14. Исключение в конструкторе

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

```
class MemoryErrorException {};
class InitializationException {};
int f( int x ) {
    // Функция выполняет какие-то сложные вычисления,
    // причем значение "нуль" нас не устраивает
    // ...
    return 0; // Моделируем ситуацию, когда возвращается 0
}
class Array {
    int *body;
    int size;
public:
    // Конструктор: резервирует память под массив
    // и инициализирует его вычисленными значениями
    Array( int customSize, int *init ) {
        try {
            if( customSize <= 0 ) {
                // Это исключение не требует дополнительных действий
                throw MemoryErrorException();
            }
            size  = customSize;
            body = new int[ size ];
            if( body == 0 ) {
                // Это исключение не требует дополнительных действий
                throw MemoryErrorException();
            }
            for( int ix = 0; ix < size; ix++ ) {
                 body[ ix ] = f( init[ ix ] );
                 if( body[ ix ] == 0 ) {
                     // Это исключение требует освободить занятые ресурсы
                     throw InitializationException();
                 }
             }
        }
        catch( InitializationException& ) {
            delete [] body;
            throw;
        }
        catch( ... ) {
            throw;
        }
    }
    // Деструктор
    ~Array() {
```

```
        // Вывод сообщения нужен нам для того, чтобы убедиться,
        // что деструктор не вызывается
        cout << "Destructor called" << endl;
        delete [] body;
    }
};

void main() {
    Array *pArray;
    int init[] = { 1, 2, 3 };
    try {
        pArray = new Array( 3, init );
        // ...
    }
    catch( MemoryErrorException& ) {
        cout << "MemoryErrorException caught" << endl;
        // ...
    }
    catch( InitializationException& ) {
        cout << "InitilizationException caught" << endl;
        // ...
    }
}
```

При отсутствии ошибок резервирования динамической памяти программа должна напечатать единственное сообщение:

```
InitializationException caught
```

Генерация исключения означает, что работа конструктора не завершена нормальным образом, следовательно, объект не создан. Поэтому деструктор не будет вызван, а, следовательно, позаботиться о корректном разрушении "созданной части" объекта нужно непосредственно перед генерацией исключения.

Существует, однако, второй вариант возникновения исключения, при котором не удается так просто выйти из положения. Речь идет о ситуации, когда конструктор обращается к некоторой функции, в ходе исполнения которой может быть выброшено "неизвестное" исключение, например, так, как это представлено в программе листинга 6.15.

Листинг 6.15. Исключение, генерируемое функцией, вызываемой в конструкторе

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

```
class MemoryErrorException {};
class UnknownException {};
int f( int x ) {
    // Функция выполняет какие-то сложные вычисления,
    // причем может выбросить неизвестное исключение
    // ...
    throw UnknownException(); // Моделируем ситуацию,
                              // когда генерируется исключение
}
class Array {
    int *body;
    int size;
    void Initialize( int *init ) {
       for( int ix = 0; ix < size; ix++ ) {
            body[ ix ] = f( init[ ix ] );
       }
    }
public:
    // Конструктор: резервирует память под массив
    // и инициализирует его вычисленными значениями
    Array( int customSize, int *init ) {
        if( customSize <= 0 ) {
            // Это исключение не требует дополнительных действий
            throw MemoryErrorException();
        }
        size  = customSize;
        body = new int[ size ];
        if( body == 0 ) {
            // Это исключение не требует дополнительных действий
            throw MemoryErrorException();
        }
        Initialize( init ); // Здесь может быть выброшено
                            // неизвестное исключение
    }
    // Деструктор
    ~Array() {
        // Вывод сообщения нужен нам для того, чтобы убедиться,
        // что деструктор не вызывается
        cout << "Destructor called" << endl;
        delete [] body;
    }
};
```

```
void main() {
    Array *pArray;
    int init[] = { 1, 2, 3 };
    try {
        pArray = new Array( 3, init );
        // ...
    }
    catch( ... ) {
        cout << "UnknownException caught" << endl;
        // ...
    }
}
```

При отсутствии ошибок резервирования динамической памяти программа должна напечатать единственное сообщение:

```
UnknownException caught
```

Как и в предыдущем примере, деструктор не вызывается. Однако теперь память, зарезервированная под массив `body`, не освобождена, что ведет к утечке памяти.

Приведенные примеры показывают, что ошибки в конструкторах являются реальной проблемой, из чего Голуб делает вывод о ненадежности исключений, генерируемых в конструкторах [Holub, 1995]. Заметим, что анализ примеров, представленных в листингах 6.14—6.15, действителен в предположении, что доступа к реализации функции `f()` у разработчика нет (предположим, что это некая библиотечная функция).

Решение проблемы, возникающей в программе, представленной листингом 6.15, может быть осуществлено в чисто объектно-ориентированном стиле (листинг 6.16).

Листинг 6.16. Исключение в конструкторе: как избежать утечки памяти

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

class MemoryErrorException {};
class RangeException {};
class UnknownException {};
int f( int x ) {
    // Функция выполняет какие-то сложные вычисления,
    // причем может выбросить неизвестное исключение
    // ...
```

```
    throw UnknownException(); // Моделируем ситуацию,
                              // когда генерируется исключение
}
// Этот класс отвечает за представление "абстрактного" массива
class ArrayBody {
    int *body;
    int size;
public:
    // Конструктор: резервирует память под массив
    // и инициализирует его вычисленными значениями
    ArrayBody( int customSize ) {
        if( customSize <= 0 ) {
            throw MemoryErrorException();
        }
        size  = customSize;
        body = new int[ size ];
        if( body == 0 ) {
            throw MemoryErrorException();
        }
    }
    // Деструктор
    ~ArrayBody() {
        // Вывод сообщения нужен нам для того, чтобы убедиться,
        // что деструктор вызывается
        cout << "Destructor called" << endl;
        delete [] body;
    }
    int GetSize() { return size; }
    // Перегрузка операции индексирования
    int& operator[]( int ix ) {
        if( ix < 0 || ix >= size ) throw RangeException();
        return body[ ix ];
    }
 };
// Этот класс отвечает за представление инициализируемого массива
class Array {
    ArrayBody arBody;
    void Initialize( int *init ) {
       for( int ix = 0; ix < arBody.GetSize(); ix++ ) {
            arBody[ ix ] = f( init[ ix ] );
       }
    }
```

```
public:
    // Конструктор: резервирует память под массив
    // и инициализирует его вычисленными значениями
    Array( int customSize, int *init ) : arBody( customSize ) {

        Initialize( init ); // Здесь может быть выброшено
                            // неизвестное исключение.
                            // NB! Но теперь вызывается деструктор
                            //     класса ArrayBody!
    }
};

void main() {
    Array *pArray;
    int init[] = { 1, 2, 3 };
    try {
        pArray = new Array( 3, init );
        // ...
    }
    catch( ... ) {
        cout << "Some exception caught" << endl;
        // ...
    }
}
```

При отсутствии ошибок резервирования динамической памяти программа печатает следующие сообщения:

```
Destructor called
Some exception caught
```

Прерывание исполнения конструктора `Array::Array()` приводит к выходу объектной переменной `arBody` из области видимости класса `Array`, что ведет к вызову деструктора класса `ArrayBody` и, следовательно, к корректному освобождению памяти, которую не удалось инициализировать.

Деструкторы и исключения

В принципе, следует избегать завершения работы деструктора посредством генерации исключения (требования стандартной библиотеки предписывают считать это правилом). Это требование обусловлено следующими факторами. Во-первых, поскольку деструкторы в большинстве случаев вызываются неявно, пользователю класса может оказаться затруднительно встроить в подходящее место кода обработчик этого исключения. Во-вторых, деструкторы

могут вызываться в процессе раскрутки стека приложения (stack unwinding) при возникновении исключения и выходе объектов из областей видимости функций, находящихся в иерархии вызовов. Листинг 6.17 иллюстрирует эту ситуацию.

Листинг 6.17. Вызов деструкторов при раскрутке стека

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

class SomeException {};
class X {
public:
    ~X() {
        cout << "Destructor X called" << endl;
    }
};
class Y {
public:
    ~Y() {
        cout << "Destructor Y called" << endl;
    }
};
void f3() {
    Y y;
    throw SomeException();
}
void f2() {
    X x;
    f3();
}
void f1() {
    f2();
};
void main() {
    try {
        f1();
    }
    catch( ... ) {
        cout << "SomeException caught" << endl;
    }
}
```

При исполнении данной программы будет напечатано:

```
Destructor Y called
Destructor X called
SomeException caught
```

Таким образом, легко вообразить ситуацию, когда деструктор, вызванный в процессе раскрутки стека, сам сгенерирует исключение. Эту ситуацию легко вообразить, но не ясно, что же нужно предпринимать. Страуструп замечает, что не существует общего способа определить, вправе ли механизм обработки исключений или деструктор проигнорировать одно исключение ради обработки другого. Таким образом, если деструктор вызывает функции, которые могут сгенерировать исключения, то обработка этих исключений должна происходить в деструкторе. Иными словами, в том случае, если деструктор вызывается в процессе раскрутки стека, исключение не должно покидать деструктор.

В распоряжении программиста имеется функция стандартной библиотеки `uncaught_exception()`, объявленная в заголовочном файле exception в пространстве имен `std`. Функция возвращает `true`, если сгенерированное исключение находится в процессе обработки, т. е. еще не перехвачено (листинг 6.18).

Листинг 6.18. Использование функции `uncaught_exception()`

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

class SomeException {}; // Исключение

class X {
    int value;
public:
    X( int arg ) {
        value = arg;
    }
    ~X() {
        cout << "Destructor X( " << value << " ) called ";
        // Если деструктор вызывается в процессе раскрутки стека
        // в ходе поиска обработчика исключения,
        // печатается соответствующее сообщение
        if( uncaught_exception() )
            cout << "while unwinding stack";
```

```
        cout << endl;
    }
};

class Y {
    X x; // Объект-член класса X (инициализируется в инициализаторе
         // конструктора Y и разрушается в процессе раскрутки стека)
    void f() {
        throw SomeException();
    }
public:
    Y( int arg = 0 ) : x( arg ) {
        f(); // Вызывается функция, генерирующая исключение
    }
    ~Y() {
        cout << "Destructor Y called";
    }
};

void g() {
    X x( 5 ); // Локальный объект класса X (создается и
              // разрушается в процессе раскрутки стека)
    Y y; // Локальный объект класса Y (не удается создать,
         // поэтому деструктор не вызывается)
}

void main() {
    try {
        g();
    }
    catch( ... ) {
        cout << "SomeException caught" << endl;
        X x( 10 ); // Локальный объект класса X
                   // (создается и разрушается при выходе из
                   // области видимости)
    }
}
```

В результате исполнения программы будут напечатаны следующие сообщения:

```
Destructor X( 0 ) called while unwinding stack
Destructor X( 5 ) called while unwinding stack
SomeException caught
Destructor X( 10 ) called
```

Проанализируем вывод программы. В функции `g()` успешно создается локальный объект `x`, хранящий значение `5`. Затем начинается создание объекта `y`. Класс `Y` содержит внутренний объект класса `X`, поэтому сначала инициализируется этот объект, хранящий значение `0`. После этого начинается работа собственно конструктора класса `Y`, который обращается к методу `f()`, который генерирует исключение `SomeException`. Поскольку конструктор `Y` не содержит подходящий обработчик, его выполнение прерывается. Это означает, что объект `y` не создан, а значит, деструктор класса `Y` не вызывается. Однако подобъект `x`, инициализированный значением `0`, был создан успешно в процессе инициализации, поэтому деструктор класса `X` вызывается, о чем свидетельствует первое печатаемое сообщение.

В функции `g()` также нет обработчика исключения `SomeException`, поэтому в процессе раскрутки стека разрушается и локальный объект `x`, инициализированный значением `5`. В ходе работы деструктора печатается второе сообщение.

Наконец, управление передается обработчику `catch(...)`, который печатает сообщение `"SomeException caught"`. Внутри обработчика создается также локальный объект `x`, инициализируемый значением `10`. Этот локальный объект корректно разрушается при выходе из обработчика, причем разрушение происходит обычным порядком, о чем свидетельствует последнее сообщение.

Итак, наличие функции `uncaught_exception()` позволяет определять в деструкторе различные действия в зависимости от того, уничтожен ли объект обычным образом или в процессе раскрутки стека.

Спецификация исключений

Вопрос, который немедленно возникает при обсуждении организации обработки ошибок на основе исключений, заключается в том, как узнать, что та или иная функция может генерировать исключение.

Когда мы используем самостоятельные функции или функции-члены класса, мы получаем информацию о правилах использования функции из ее объявления. Объявление функции, содержащее информацию о том, какого типа должны быть аргументы функции, сколько их должно быть, какой тип имеет возвращаемое значение функции, служит интерфейсом между обращением к функции и ее определением. Если функция может генерировать исключение, то пользователю функции нужно знать, какие типы исключений может генерировать функция. Для того чтобы предоставить пользователю функции эту информацию, объявление функции может дополняться спецификацией исключений, которые могут быть ею сгенерированы, например:

```
void ReadData( ifstream&, int& )
    throw( EmptyFileException, BadFormatException );
```

Такое объявление означает, что функция `ReadData()` может сгенерировать только исключения типов `EmptyFileException` и `BadFormatException`, а также исключения, являющиеся производными от этих типов, но никакие другие. В случае наличия группирования исключений данное объявление могло бы выглядеть следующим образом:

```
void ReadData( ifstream&, int& )
    throw( ReadFileException );
```

Основная проблема со спецификациями исключений заключается в том, что нет общепризнанного стандарта обработки спецификаций исключений в C++. Поэтому существуют варианты, с которыми вы можете столкнуться при использовании вашего компилятора.

- Компилятор может поддерживать синтаксис спецификации исключений, но не предоставлять реализации в связи с данной спецификацией, т. е. спецификация будет в этом случае просто предоставлять дополнительную информацию читателю программы, никакого контроля соответствия кода заявленной спецификации не будет осуществляться (такой стратегии придерживается компилятор Microsoft Visual C++ 6.0).
- Компилятор может поддерживать синтаксис спецификации исключений и поддерживать весьма ограниченную реализацию. Например, компилятор Microsoft Visual C++ 7.0 поддерживает реализацию двух крайних случаев:
  - спецификация `throw()`, означающая, что данная функция вообще не может генерировать исключений;
  - спецификация `throw(...)`, означающая, что функция может генерировать исключения.

В любом случае спецификации исключений полезны для того, чтобы сделать интерфейс функций более наглядным, а их поведение в глазах пользователя более предсказуемым (но следите за тем, чтобы ваши спецификации соответствовали действительному возможному поведению ваших функций).

Неожидаемые и неперехваченные исключения

Независимо от того, снабжена функция или нет спецификацией исключения, существует вероятность того, что будет сгенерировано неожиданное исключение (функция может вызывать другую функцию, к реализации которой у разработчика нет доступа). Для перехвата такого рода исключений можно воспользоваться обработчиком `catch(...)`. Однако, если хочется избежать постоянного включения обработчика `catch(...)` в перечень обработчиков для предотвращения неожиданных исключений, можно воспользоваться известной технологией, связанной с функциями обратного вызова. Рассмотрим, как это можно сделать.

Обработка исключений реализована в C++ таким образом, что если исключение сгенерировано, но не перехвачено, управление передается глобальной функции стандартной библиотеки `terminate()`. Эта функция вызывается также и в том случае, если в процессе обработки исключения выяснится, что стек разрушен, или если деструктор, вызванный в процессе раскрутки стека, попытается завершить свою работу при помощи исключения. По умолчанию функция `terminate()` вызывает функцию `abort()` для аварийного завершения приложения. Программист может зарегистрировать собственную функцию, которая будет вызвана вместо `terminate()`, определив тем самым собственную версию аварийного завершения программы. Это осуществляется с помощью функции `set_terminate()`, объявленной в заголовочном файле eh.h или в файле exception в пространстве имен `std` (идея примера в листинге 6.19 заимствована из [Eckel, 2000-1]).

Листинг 6.19. Неожидаемые и неперехваченные исключения

```
#include <cstdlib>
#include <iostream>

#include <exception>

using namespace std;

// Примеры типов исключений
class First {};
class Second {};
class Unknown {};

void g();

void f( int i ) throw ( First, Second ) {
  switch( i ) {
    case 1: throw First();
    case 2: throw Second();
  }
  g(); // Эта функция генерирует "неожиданное" исключение
}

void g() { throw Unknown(); }

// Пользовательская функция аварийного завершения
void MyTermination () {
  cout << "Unknown exception thrown" << endl;
  exit( 1 );
}
```

```
void main() {

  // Регистрация функции MyTermination() для
  // вызова вместо terminate()
  set_terminate( MyTermination );

  for( int i = 1; i <=3; i++ ) {
    try {
      f( I );
    }
    catch( First& ) {
      // (на первой итерации цикла)
      cout << "First exception caught" << endl;
    }
    catch( Second& ) {
      // (на второй итерации цикла)
      cout << "Second exception caught" << endl;
    }

   /* ЭТО КОММЕНТАРИЙ, СОДЕРЖАЩИЙ АЛЬТЕРНАТИВНЫЙ ВАРИАНТ!
    *
    * Можно перехватить неизвестное исключение так:
    *
    * catch( ... ) {
    *     // (на третьей итерации цикла)
    *     cout << "Unknown exception caught" << endl;
    * }
    * КОНЕЦ АЛЬТЕРНАТИВНОГО ВАРИАНТА
    */
  }
}
```

В результате при исполнении данной программы будет напечатано:

```
First exception caught
Second exception caught
Unknown exception thrown
```

Появление последнего сообщения является результатом работы функции `MyTermination()`, которая вызывается неявно как функция обратного вызова.

Стандартные исключения

Ряд типов исключений, которые могут генерироваться функциями стандартной библиотеки C++, определены в заголовочных файлах, исполняющих роль интерфейса к средствам стандартной библиотеки. Подробное описание

иерархии стандартных исключений см. в [Stroustrup, 2000, разд. 14.10]. Рассмотрим в качестве примера стандартное исключение `bad_cast`, генерируемое в случае неудачного преобразования ссылки посредством `dynamic_cast` (для этого воспользуемся типами из примера с геометрическими фигурами, *рассмотренного в разд. 5.3*).

```
#include <typeinfo.h> // для bad_cast

void RotateShape( Shape& refShape ) {
    try {
        Trangle& refTr =
            dynamic_cast <Triangle&>( refShape );

        refTr.Rotate90();
    }
    catch( bad_cast& ) {
        // Это не треугольник
        // ...
    }
}
```

6.4. Особенности обработки исключений в языках Java и C#

Есть некоторые отличия в реализациях механизма обработки исключений в других объектно-ориентированных языках. В наибольшей степени эти отличия проявляются в языках, ориентированных на разработку программ, исполняющихся в контролируемых средах. *Информацию о контролируемом коде см. в разд. 10.1.*

В частности, конструкция `try – catch` в программах на этих языках иногда дополняется блоком `finally`, который содержит код, исполняемый как в случае отсутствия исключения (т. е. сразу после блока `try`), так и в случае перехваченного исключения (т. е. сразу после отработавшего блока `catch`). Необходимость такого блока обусловлена тем обстоятельством, что в блоке `try` могут быть осуществлены действия, для которых необходимо гарантировать определенные операции завершения. Примерами таких действий могут быть: открытие файла, установление сетевого соединения, подключение некоего внешнего ресурса и т. п. Приводимый здесь фрагмент программы на C# иллюстрирует работу механизма с блоком `finally`.

```
// System.IO namespace needed
static void UseFile( string fileName ) {
    FileStream fin = null;
```

```
    try {
        // Открытие файла
        fin = new FileStream( filename, FileMode.Open );
        // Загрузка и обработка данных...
        // ...
    }
    catch( SomeDataflowException ) {
        // Обработка исключения в связи с ошибкой в данных
        // ...
    }
    finally {
        // Действия осуществляются в любом случае
        if( fin != null ) fin.Close();
    }
}
```

В C++ подобный механизм не поддерживается, поскольку обеспечение аналогичных действий реализуется правильным определением деструкторов объектов.

Отличия в реализации механизма обработки исключений могут быть связаны с правилами спецификации исключений и требованиями обязательности обработки исключений в клиентском коде. Рассмотрим эту проблему на примере языков Java и C#.

Java: обязательная спецификация и обработка исключений в Java

Модель программирования, используемая в Java, налагает жесткие требования на спецификацию программных объектов. Это относится и к спецификации исключений. Спецификация исключений в Java является обязательной. Если в C++ объявление функции без спецификатора генерируемых ей исключений означает: "Эта функция может генерировать исключения и может не генерировать", то в Java подобное объявление метода означает, что он не может генерировать исключений. Соблюдение спецификации проверяется на этапе трансляции программы в интерпретируемый байт-код. Благодаря обязательности спецификации исключений, пользователь класса в точности знает, какой код он должен написать для обработки исключений, генерируемых методами этого класса.

Говоря более детально, это означает следующее. Если некоторый метод `f1()` вызывает некоторый метод `f2()`, который может генерировать исключение (это должно быть указано в обязательной спецификации), то при реализации метода `f1()` у программиста есть две альтернативы:

- реализовать в методе `f1()` обработчик исключений, генерируемых `f2()`;
- поместить в спецификацию исключений, генерируемых `f1()`, все необработанные исключения, генерируемые `f2()`.

Посредством использования такого подхода гарантируется, что непротиворечивость спецификаций будет установлена на стадии компиляции.

Итак, если некоторый метод генерирует исключение, это обязательно должно быть указано в спецификации. В спецификации не указываются только исключения, производные от типа `RuntimeException`, поскольку они генерируются исполнительной системой и не зависят от воли программиста. К категории таких исключений относятся все ситуации, связанные с проверками времени выполнения (преобразования типов, выход за границы массивов, деление на нуль, неинициализированные ссылки, переполнения и т. д.). Исключения `RuntimeException` могут не обрабатываться, для остальных исключений наличие обработчиков в программе обязательно.

C#: необязательная обработка исключений

C# не поддерживает спецификаторов исключений, вместо этого разработчикам предлагается использовать документирующие комментарии. Более того, наличие обработчиков также не обязательно. Если подходящий обработчик не будет найден, исключение обрабатывается исполнительной системой. На практике это означает вывод информационного сообщения и трассы стека.

Такой подход гарантирует, что любое исключение будет обработано: программиста, не желающего включить в свой код обработчик для конкретного исключения, никто не обязывает это делать, в то время как при программировании на Java у такого программиста может возникнуть соблазн написать "пустой" обработчик, не содержащий никаких действий. В то же время, наличие "пустого" обработчика (фактически, означающего, что исключение никем не обрабатывается) может оказаться гораздо более опасным, чем отсутствие обработчика (в этом случае обработкой займется исполнительная система).

6.5. Вопросы и упражнения

1. Нужно ли использовать исключения, если ошибка может быть полностью обработана в том контексте исполнения программы, в котором она произошла?
2. Спецификация C++ не запрещает считать триаду `throw – try – catch` просто вариантом управляющей структуры, которая дополняет операторы цикла, условный оператор и оператор `goto`. Рассмотрите в качестве примера подобного использования `throw – try – catch` реализацию выхода из

рекурсии или прерывание обработки вложенных циклов (напишите соответствующие примеры программ). Как Вы считаете, разумно ли использовать обработчики исключений в ситуациях, не связанных с ошибками?

3. Переработайте пример листинга 5.11 так, чтобы вместо обработки ошибки на месте использовался механизм обработки исключений. Не забудьте изменить организацию вызова функции `Rotate90()`, используя блок `try` и блок обработки исключения.
4. Напишите исправленную версию класса `QueueInt`, использующего генерацию исключений при возникновении ошибок в работе с очередью (ошибка инициализации, ошибка в связи попыткой занесения данных в заполненную очередь, ошибка в связи с извлечением данных из пустой очереди). Рассмотрите следующие альтернативы: помещение определений классов исключений внутри определения класса `QueueInt`; реализация самостоятельных классов исключений. Выберите один из вариантов и обоснуйте свой выбор. Напишите тестирующую программу, иллюстрирующую процесс перехвата исключений и обработки ошибок.
5. Существуют случаи, когда некоторая функция нуждается в явном управлении доступом к некоторым ресурсам. Например, в начале своей работы функция блокирует доступ к некоторому ресурсу, а по окончании работы освобождает его:

```
void TakeResource() {
    SomeResoure *rsc = GetResource( ... );

    // Использование rsc
    UseResource( rsc );

    // ...

    ReleaseResource( rsc );
}
```

Между вызовами функций `TakeResource()` и `ReleaseResource()` может возникнуть какая-то нештатная ситуация, требующая завершения `UseResource()`, однако при этом необходимо обеспечить гарантированное корректное освобождение ресурса. Напишите решение этой задачи, основанное на генерации и обработке исключений. Исходите из того, что тип исключения, который может быть сгенерирован между вызовами функций `TakeResource()` и `ReleaseResource()`, не обязательно известен разработчику функции `TakeResource()`, но может быть известен ее пользователю. Поэтому используйте обработчик `catch(...)` и обеспечьте повторную генерацию исключения. Напишите "игрушечное" приложение, моделирующее работу с ресурсами на основе реализованного механизма.

6. Решение задачи 5 на основе механизма обработки исключений, по выражению Страуструпа, является "многословным, утомительным и потенциально дорогостоящим". Что, если в проекте имеется довольно много функций, работающих с ресурсами подобным образом? Попробуйте реализовать решение задачи 5, не прибегая к обработке исключений, а используя чисто объектно-ориентированный подход. Подсказка: определите класс `ResourceHandler` и реализуйте захват ресурса в виде конструктора, а освобождение ресурса в виде деструктора. Перепишите функцию `TakeResource()` с использованием локального объекта класса `ResourceHandler`. Каким образом теперь обеспечивается гарантированное освобождение ресурса? Напишите "игрушечное" приложение, моделирующее работу с ресурсами на основе реализованного механизма.

7. Усовершенствуйте модель электрической лампочки, использованную в проекте `Garland Application` *(см. гл. 2)*, определив абстракцию "перегорающей лампочки". Используйте генератор случайных чисел для инициирования момента перегорания лампочки при включении. В случае наступления события "лампочка перегорела" инициируйте соответствующее исключение. Напишите пример программы, перехватывающей это исключение.

8. [Eckel, 2000-1] Напишите определение класса, содержащего деструктор, генерирующий исключение. Напишите тестирующую программу, позволяющую увидеть, что наличие подобного деструктора может привести к тому, что новое исключение генерируется до того, как будет достигнут (в процессе раскрутки стека) обработчик ранее сгенерированного исключения. Какая глобальная функция при этом вызывается?

9. [Eckel, 2000-1] Докажите, что все сгенерированные объекты-исключения правильно разрушаются. Убедитесь в этом, написав программу, содержащую определения классов исключений с деструкторами, обеспечивающими печать на консоль информационных сообщений о разрушении объектов.

10. [Stroustrup, 2000] Определите класс `SafeInt`, который ведет себя точно так же, как встроенный тип `int`, за исключением того, что он генерирует исключения, не допуская переполнения сверху или снизу и деления на 0.

11. Напишите библиотеку работы с файлами (открытие, закрытие, чтение, запись), подобную средствам библиотеки C, интерфейс к которой обеспечивается файлом stdio.h. Используйте генерацию исключений вместо формирования возвращаемых значений функций библиотеки при возникновении ошибочных ситуаций. Определите иерархию исключений в связи с организацией ввода-вывода. Напишите тестирующую программу.

12. Рассмотрите модель пула ресурсов.

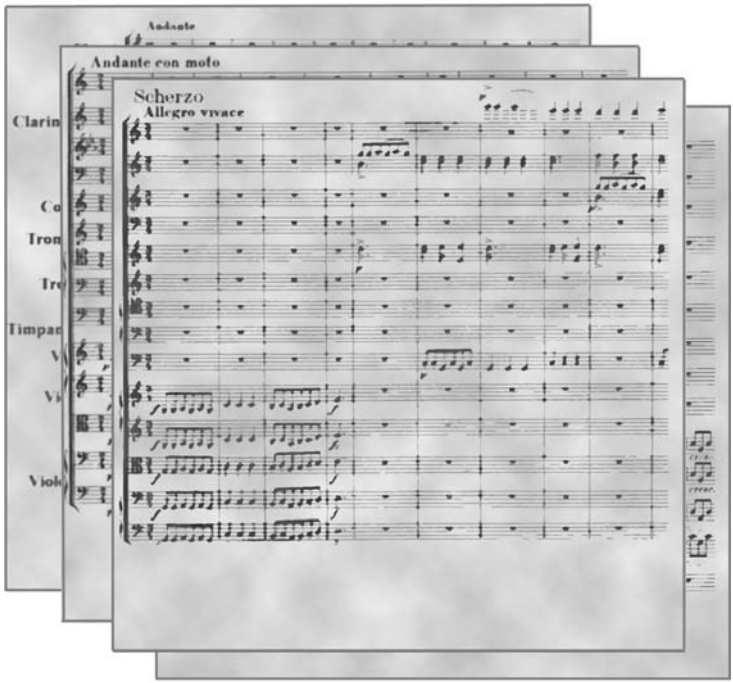
```
#include <iostream>
using namespace std;

class MemoryErrorException {}; // Исключение при ошибке выделения памяти
class ResourceException {}; // Исключение при ошибке инициализации
                            // ресурса

// Модель "ресурс 1"
class Resource1 {
public:
    Resource1() {
        cout << "Resource1 constructed" << endl;
    }
    ~Resource1() {
        cout << "Resource1 destroyed" << endl;
    }
};

// Модель "ресурс 2": резервирует динамическую память
class Resource2 {
    int *body;
    int size;
public:
    Resource2( int _size ) {
        size = _size;
        body = new int[ size ];
        if( body == 0 ) throw MemoryErrorException();
            cout << "Resource2[" << size << "] constructed" << endl;
    }
    ~Resource2() {
        delete [] body;
        cout << "Resource2[" << size << "] destroyed" << endl;
    }
};

// Модель "ресурс 3": определяет собственные версии
//                    операций new и delete
class Resource3 {
public:
    Resource3() {
        cout << "Resource3 constructed" << endl;
    }
```

```
    ~Resource3() {
        cout << "Resource3 destroyed" << endl;
    }
    void *operator new( unsigned int _size ) {
        cout << "Resource3: " << _size << " items allocated" << endl;
        throw ResourceException();
    }
    void operator delete( void *ptr ) {
       cout << "Resource3: deletion" << endl;
       ::delete ptr;
    }
};

// Пул ресурсов
class ResourcePool {
    Resource1 *rs1;
    Resource2 *rs2;
    Resource3 *rs3;
public:
    ResourcePool( int limit ) {
        cout << "Initializing resource pool" << endl;
        rs1 = new Resource1[ limit ];
        rs2 = new Resource2( 10 );
        rs3 = new Resource3;
    }
    ~ResourcePool() {
        cout << "Destroying resource pool" << endl;
        delete [] rs1;
        delete rs2;
        delete [] rs3;
    }
};

void main() {
    try {
        ResourcePool pool( 2 );
    }
    catch( ... ) {
        cout << "Some exception caught" << endl;
    }
}
```

Попробуйте разобраться, что напечатает данная программа. Обратите внимание на то, что при возникновении "неожиданного" исключения, генерируемого в перегруженной операции `new` в классе `Resource3`, для корректно созданных объектов классов `Resource1` и `Resource2` деструкторы не вызываются. Предложите вариант организации классов, в котором не возникает утечка памяти. При этом постарайтесь, чтобы перехват исключений не происходил в конструкторе класса `ResourcePool`.

ЧАСТЬ III

Организация и взаимодействие типов

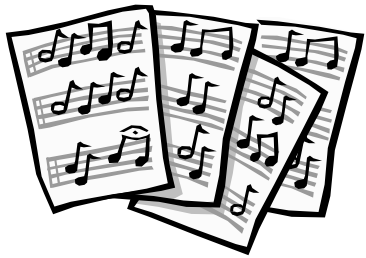


Глава 7. Множественное наследование и интерфейсы

Глава 8. Пространства имен в связи с модульностью и иерархией

Глава 9. Введение в обобщенное программирование

ГЛАВА 7



Множественное наследование и интерфейсы

Множественное наследование предоставляет возможность определять классы, наследующие более чем одному классу. Множественное наследование является довольно сложной концепцией ООП. При этом следует иметь в виду, что многие объектно-ориентированные языки не поддерживают множественное наследование. Основной причиной этого является, вероятно, тот факт, что некорректное использование множественного наследования может порождать трудные для обнаружения логические ошибки проектирования. Однако справедливо также и то, что в ряде случаев использование множественного наследования представляется вполне оправданным. В этой главе мы рассмотрим несколько стандартных примеров иерархий классов, построенных с использованием множественного наследования, а также обсудим варианты реализации подобных иерархий с ограниченным использованием множественного наследования.

Здесь мы выделяем три стандартных варианта использования концепции множественного наследования:

1. Множественное наследование для реализации суммы функциональностей двух или более классов.
2. Вариант множественного наследования, при котором базовые классы имеют, в свою очередь, общий базовый класс.
3. Множественное наследование, при котором реализация наследуется однократно, а интерфейсы могут наследоваться многократно.

Данная классификация довольно условна, и зачастую иерархия классов может содержать элементы, характерные для различных вариантов множественного наследования, однако такое подразделение помогает систематизировать изложение учебного материала и облегчить студентам его восприятие.

7.1. Множественное наследование для реализации суммы функциональностей

В этом разделе мы в основном сосредоточимся на том варианте множественного наследования, при котором основной задачей является получение класса, суммирующего функциональности других классов. Впрочем, исходные классы при этом вполне могут иметь общий базовый класс.

Обычное множественное наследование

Для иллюстрации первого типа множественного наследования обратимся снова к примеру с гирляндой из цветных лампочек и рассмотрим подробности реализации фрагмента кода, соответствующего фрагменту иерархии классов, представленной на рис. 2.11. Для удобства дальнейшего изложения на рис. 7.1 мы воспроизвели этот фрагмент отдельно от прочих классов.

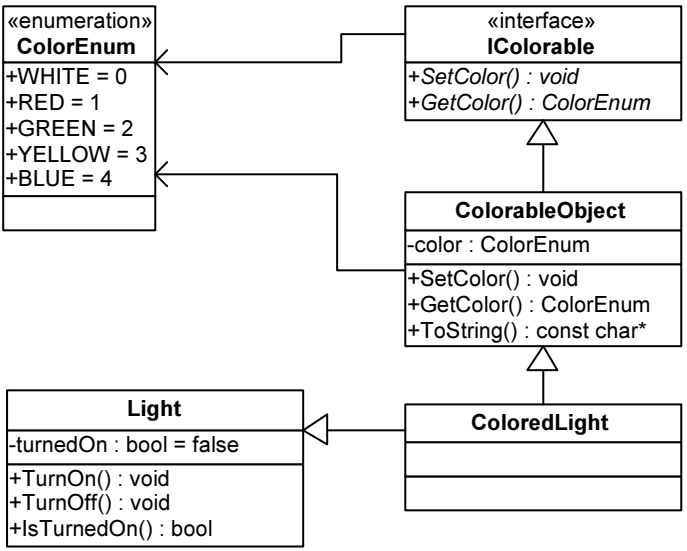


Рис. 7.1. Фрагмент иерархии классов из Garland Application

В соответствии с предложенным в *гл. 2* проектом иерархии классов, определение класса `ColoredLight` получается наследованием от двух базовых классов: класса `Light` и класса `ColorableObject`. Напомним, что класс `Light` реализует представление электрической лампочки в самом общем плане, а класс `ColorableObject` реализует интерфейс, определяемый абстрактным классом `IColorable`. `ColorableObject` и `IColorable` зависят от определения типа-

перечисления, моделирующего различные цвета в виде целочисленных констант.

Реализация методов класса `ColorableObject` очень проста, и поэтому они могут быть определены как встраиваемые (inline) функции (см. листинг 7.1). Напомним, что методы, задаваемые в пределах определения класса, являются встроенными функциями-членами в соответствии с правилами языка C++. В листинге 7.1 представлены также определения классов `Light` и `ColoredLight`.

Для наблюдаемости процесса создания подобъектов базовых классов при множественном наследовании все конструкторы классов листинга 7.1 содержат печать информационных сообщений.

Листинг 7.1. Классы из иерархии классов `Garland Application`

```
// Colorable.h

#ifndef _Colorable_H
#define _Colorable_H

#include <iostream>
using namespace std;

enum ColorEnum {
    WHITE  = 0,
    RED    = 1,
    GREEN  = 2,
    YELLOW = 3,
    BLUE   = 4
};

class IColorable {
public:
    virtual void SetColor( ColorEnum color ) = 0;
    virtual ColorEnum GetColor() = 0;
};

class ColorableObject : public IColorable {
protected:
    ColorEnum color;
public:
    ColorableObject( ColorEnum color = WHITE ) {
        cout << "ColorableObject constructed" << endl;
        this->color = color;
    }
```

```
    void SetColor( ColorEnum color ) {
        this->color = color;
    }

    ColorEnum GetColor() {
        return color;
    }

    const char * ToString() {
        switch( GetColor() ) {
            case WHITE  : return "white";
            case RED    : return "red";
            case GREEN  : return "green";
            case YELLOW : return "yellow";
            case BLUE   : return "blue";
            default     : return "white";
        }
    }
};
#endif

// Light.h
#ifndef _Light_H
#define _Light_H

#include <iostream>
using namespace std;

class Light {

    bool turnedOn;

public:

    Light() {
        cout << "Light constructed" << endl;
        turnedOn = false;
    }

    void TurnOn() {
        if( !turnedOn ) turnedOn = true;
    }

    void TurnOff() {
        if( turnedOn ) turnedOn = false;
    }
```

```
    bool IsTurnedOn() {
        return turnedOn;
    }
};

#endif

// ColoredLight.h
#ifndef _ColoredLight_H
#define _ColoredLight_H

#include <iostream>
using namespace std;

#include "Light.h"
#include "Colorable.h"

class ColoredLight : public Light,
                     public ColorableObject {

public:
    ColoredLight( ColorEnum color ) :
           // Инициализатор: явный вызов конструктора
           // базового класса
           ColorableObject( color )
    {
        cout << "ColoredLight constructed" << endl;
    }

};
#endif

// MainTest.cpp
#include <iostream>
using namespace std;

#include "ColoredLight.h"

void main()
{
    ColoredLight cl( GREEN );
    cout << "Current color is " << cl.ToString() << endl;

     cl.SetColor( RED );
    cout << "Color is changed to " << cl.ToString() << endl;
}
```

Отметим, что поскольку класс `Light` содержит единственный конструктор, и этот конструктор является конструктором по умолчанию, излишне упоминать конструктор класса `Light` в списке инициализаторов в заголовке конструктора класса `ColoredLight`.

Простая тестирующая программа, представляемая здесь функцией `main()`, позволяет наблюдать за процессом создания объекта класса `ColoredLight`. Этот объект содержит в себе подобъекты двух базовых классов. Члены и подобъекты базовых классов конструируются в порядке их объявления в классе, а уничтожаются в обратном порядке. Таким образом, порядок указания базовых классов в заголовке производного класса существенен. Результаты исполнения приведенной в листинге 7.1 программы выглядят следующим образом:

```
Light constructed
ColorableObject constructed
ColoredLight constructed
Current color is green
Color is changed to red
```

Если в базовых классах, использованных в ходе множественного наследования, существуют доступные члены с одинаковыми именами, то при их применении необходимо явно указывать область видимости каждого такого имени (листинг 7.2).

Листинг 7.2. Конфликты имен при множественном наследовании

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

class Base1 {
    // ...
public:
    void f() {
        cout << "Base1::f()" << endl;
    }
};

class Base2 {
    // ...
public:
    void f() {
        cout << "Base2::f()" << endl;
    }
};
```

```
class Derived : public Base1, public Base2 {
public:
    void g() {
        f(); // Неправильно: какая из f()?=> ошибка компиляции
        Base1::f(); // Правильно
    }
};

void main() {
    Derived d;
    d.g();
    d.f();// Неправильно: какая из f()? => ошибка компиляции
    d.Base2::f(); // Правильно
}
```

Строки, содержащие неправильные варианты обращения к унаследованным методам, вызовут ошибки компиляции.

Для того чтобы минимизировать или устранить использование явных квалификаторов класса при работе с объектами производного класса, при определении производного класса можно переопределить метод `f()` для необходимого в требуемом контексте разрешения конфликта имен (соответствующий пример приводится в листинге 7.3).

Листинг 7.3. Разрешение конфликтов имен при множественном наследовании

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

class Base1 {
    // ...
public:
    void f() {
        cout << "Base1::f()" << endl;
    }
};

class Base2 {
    // ...
public:
    void f() {
        cout << "Base2::f()" << endl;
    }
};
```

```
class Derived : public Base1, public Base2 {
public:
    void f() {
        Base2::f(); // Вызов f() всегда приводит к вызову
                    // версии f() из Base2
    }
};

void main() {
    Derived d;
    d.f();// Правильно, опосредованно вызывается Base2::f()
    d.Base2::f(); // Тоже правильно
    d.Base1::f(); // Правильно, если требуется Base1::f()
}
```

Теперь рассмотрим другой несложный пример множественного наследования, вариации которого используются и в следующих разделах.

В языках Java и C# концепция консоли представлена на более высоком уровне абстракции, чем в C++. В программах на C++ используется целый набор предопределенных объектов для консольного ввода и вывода (`cin`, `cout`, `clog`, `cerr`). В программах на C# используется класс `Console`, предоставляющий необходимую функциональность ввода и вывода в консольных приложениях. Набросок определения подобного класса в C++ иллюстрирует листинг 7.4.

Листинг 7.4. Определение модели "универсального" консольного объекта

```
#include <iostream.h>

// Класс для консольного ввода
class Reader {
    int item; // Считываемая величина
public:
    int ReadLine() { // Ввод целого числа с консоли
        cin >> item;
        return item;
    }
};

// Класс для консольного вывода
class Writer {
public:
    // Методы для вывода целого числа
    void WriteLine( int item ) {
        cout << item << endl;
    }
```

```
    void Write( const char* text ) {
        cout << text;
    }
};

// Класс для консольного ввода и вывода
class ReWriter : public Reader, public Writer {};

ReWriter Console; // Глобальный объект "консоль"

void main() {
    Console.Write( "Enter integer value: " );
    int somevalue = Console.ReadLine();

    Console.Write( "Entered value: " );
    Console.WriteLine( somevalue );
}
```

Класс `ReWriter` получен в результате "склеивания" классов `Reader` и `Writer` (рис. 7.2).

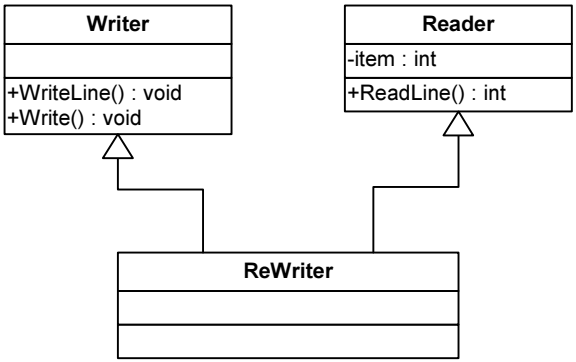


Рис. 7.2. Множественное наследование как сумма функциональностей

Определив глобальный объект `Console` (конструкторы глобальных объектов вызываются раньше конструкторов локальных объектов), мы получили экземпляр абстрактного представления консоли, совмещающего в себе как возможности в связи с консольным вводом, так и возможности в связи с консольным выводом. Сам по себе этот пример, может быть, и не представляет большого интереса, однако его развитие поможет нам рассмотреть ряд тонких моментов множественного наследования.

Множественное наследование с общим базовым классом

Теперь попытаемся развить основную конструктивную идею предыдущего примера, принимая во внимание тот факт, что организация консольного ввода-вывода может рассматриваться как частный случай файлового ввода-вывода. Для реализации механизма файлового ввода-вывода расширим определения классов `Reader` и `Writer` таким образом, чтобы их объединение позволило создавать объекты, нацеленные, в том числе, и на возможность ввода данных из файла и вывода данных в файл. При этом воспользуемся имеющимися в стандартной библиотеке C++ возможностями определения потоков ввода-вывода на основе концепции буферизованного ввода-вывода *(информацию об организации ввода-вывода с буферизацией см. в приложении П2.1)*.

Для обеспечения буферизованного ввода в классе `Reader` объявим объектную переменную `fbr` типа `filebuf` (файловый поток для буферизованного ввода-вывода) и объектную переменную `inputStream` типа `istream_withassign`, представляющую абстракцию входного потока с разрешенным присваиванием объектных переменных этого типа. Аналогично для обеспечения буферизованного вывода в классе `Writer` объявим объектные переменные `fbw` типа `filebuf` и `outputStream` типа `ostream_withassign`, представляющую абстракцию выходного потока (рис. 7.3).

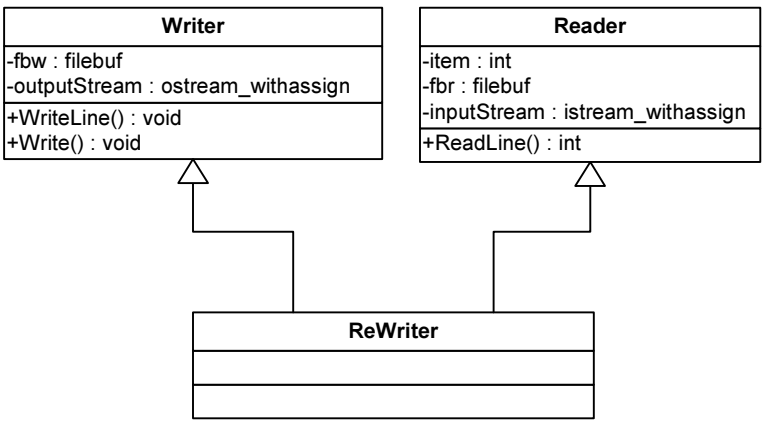


Рис. 7.3. Конструирование класса ввода-вывода с буферизацией

В связи с такими дополнениями, определения классов `Reader` и `Writer` потребуют введения дополнительных конструкторов для инициализации различных видов объектов (как для потокового, так и для файлового ввода-вывода). Результат модификации программы приведен в листинге 7.5.

Листинг 7.5. Определение универсальных объектов потокового и файлового ввода-вывода

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

// Класс для потокового ввода с буферизацией
class Reader {

    int item;                        // Считываемая величина
    filebuf fbr;                     // Файловый буфер
    istream_withassign inputStream; // Поток для ввода

public:
    Reader( istream_withassign& is = cin ) {

        inputStream = is;
    }

    Reader( const char* filename ) {

        if( fbr.open( filename, ios::in |
                                ios::nocreate ) == NULL ) {

            //Обработка ошибки открытия файла...
            return;
        }
        inputStream = &fbr;
    }

    ~Reader() {

        if( fbr.is_open() ) fbr.close();
    }
    int ReadLine() {
        inputStream >> item;
        return item;
    }
};

// Класс для потокового вывода с буферизацией
class Writer {

    filebuf fbw;                      // Файловый буфер
    ostream_withassign outputStream; // Поток для вывода

public:
    Writer( ostream_withassign& os = cout ) {
```

```
        outputStream = os;
    }

    Writer( const char* filename ) {

        if( fbw.open( filename, ios::out ) == NULL ) {

            // Обработка ошибки открытия файла...
            return;
        }
        outputStream = &fbw;
    }

    ~Writer() {
        if( fbw.is_open() ) fbw.close();
    }

    void WriteLine( int item ) {

        outputStream << item << endl;
        outputStream.flush(); // Для очистки буфера
    }

    void Write( const char *text ) {

        outputStream << text;
        outputStream.flush(); // Для очистки буфера
    }
};

// Класс для потокового ввода-вывода с буферизацией
class ReWriter : public Reader, public Writer {

public:
    ReWriter( istream_withassign& is = cin,
              ostream_withassign& os = cout ) :
        Reader( is ),
        Writer( os )
    {}

    ReWriter( const char* ifilename,
              const char* ofilename ) :
        Reader( ifilename ),
        Writer( ofilename )
    {}
};
```

```
ReWriter Console; // Глобальный объект "консоль"

void main() {

    int somevalue;

    // Работаем с консолью
    Console.Write( "Enter integer value: " );
    somevalue = Console.ReadLine();
    Console.Write( "Entered value: " );
    Console.WriteLine( somevalue );

    // Создаем объект файлового ввода-вывода:
    //   ввод из файла "input.txt";
    //   вывод в файл "output.txt"
    ReWriter rw( "input.txt", "output.txt" );

    // Работаем с объектом файлового ввода-вывода
    somevalue = rw.ReadLine();

    rw.Write( "Read value: " );
    rw.WriteLine ( somevalue );
}
```

Примечание

Приведенная в листинге 7.5 программа может быть объектом критики в связи с использованием "старой" версии библиотеки ввода-вывода. Однако, поскольку здесь обсуждаемым моментом является взаимодействие классов, построенных с использованием множественного наследования, а не концепции обеспечения ввода-вывода данных, это обстоятельство вряд ли имеет решающее значение. Кроме того, заметим, что обсуждение возможностей версий классов, обслуживающих ввод-вывод и определенных в пространстве имен `std`, неминуемо приводит к необходимости подробного рассмотрения концепции обобщенного программирования и ее реализации в C++ в виде шаблонных классов. Анализу этих проблем посвящена в книге отдельная глава *(см. гл. 9)*.

Анализируя этот текст, можно заметить, что действия, выполняемые в конструкторах `Reader(const char*)` и `Writer(const char*)`, весьма сходны. Вообще, создание объекта, поддерживающего буферизованную передачу данных из файла или в файл на диске логично (с точки зрения инкапсуляции) реализовать на основе отдельного класса. В конечном счете, это не только приведет к небольшому упрощению кода для классов `Reader` и `Writer`, но также обеспечит разделение уровней абстракций при вводе-выводе (рис. 7.4).

Классы `Reader` и `Writer` в этом случае практически полностью определяются на уровне абстракций, связанных с потоками ввода-вывода. Буферизация как внутренний механизм при этом скрывается в отдельном классе (листинг 7.6).

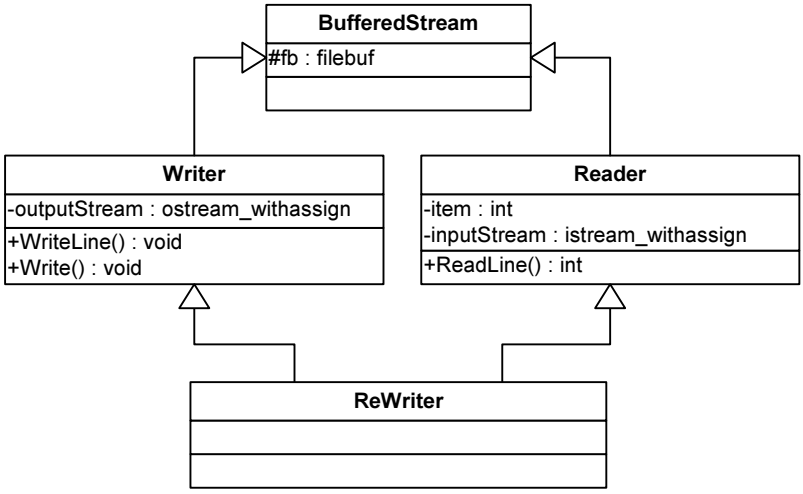


Рис. 7.4. Множественное наследование с общим базовым классом

Листинг 7.6. Выделение абстракции буферизованного потока в отдельный класс

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

// Класс "буферизованный поток"
class BufferedStream {

    filebuf fb; // Файловый буфер

public:
    BufferedStream() {}

    BufferedStream( const char* filename, int mode ) {

        if( fb.open( filename, mode ) == NULL ) {

            // Обработка ошибки открытия файла...
            return;
        }
    }

    ~BufferedStream() {

        if( fb.is_open() )   fb.close();
    }
```

```
    filebuf *GetFBAddress() {

        return &fb;
    }
};


// Класс для потокового ввода с буферизацией
class Reader : public BufferedStream {

    int item;
    istream_withassign inputStream;

public:
    Reader( istream_withassign& is = cin ) {

        inputStream = is;
    }

    Reader( const char* filename ) :
        BufferedStream( filename, ios::in | ios::nocreate ) {

        inputStream = GetFBAddress();
    }
    int ReadLine() {

        inputStream >> item;
        return item;
    }
};

// Класс для потокового вывода с буферизацией
class Writer : public BufferedStream {

    ostream_withassign outputStream;

public:
    Writer( ostream_withassign& os = cout ) {

        outputStream = os;
    }

    Writer( const char* filename ) :
        BufferedStream( filename, ios::out ) {

        outputStream = GetFBAddress();
    }
```

```
    void WriteLine( int item ) {

        outputStream << item << endl;
        outputStream.flush();
    }

    void Write( const char *text ) {

        outputStream << text;
        outputStream.flush();
    }
};

// Класс для потокового ввода-вывода с буферизацией
class ReWriter : public Reader, public Writer {
public:
    ReWriter( istream_withassign& is = cin,
              ostream_withassign& os = cout ) :
        Reader( is ),
        Writer( os )
    {}

    ReWriter( const char* inName, const char* outName ) :
        Reader( inName ),
        Writer( outName )
    {}
};

ReWriter Console;

void main() {

    int somevalue;

    // Работаем с консолью
    Console.Write( "Enter integer value: " );
    somevalue = Console.ReadLine();
    Console.Write( "Entered value: " );
    Console.WriteLine( somevalue );

    // Создаем объект файлового ввода-вывода:
    //   ввод из файла "input.txt";
    //   вывод в файл "output.txt"
    ReWriter rw( "input.txt", "output.txt" );

    // Работаем с объектом файлового ввода-вывода
    somevalue = rw.ReadLine();
```

```
    rw.Write( "Read value: " );
    rw.WriteLine ( somevalue );
}
```

Класс `BufferedStream` в этом примере дважды оказывается базовым классом для класса `ReWriter` (см. рис. 7.4). Подобную иерархию наследования часто называют ромбовидным, или "бриллиантовым", наследованием (diamond inheritance).

На уровне взаимодействия объектов (рис. 7.5) картина выглядит следующим образом. Объект класса `ReWriter` содержит в себе в качестве подобъектов члены базовых классов `Reader` и `Writer`. Подобъекты `Reader` и `Writer`, в свою очередь, содержат по одному подобъекту `BufferedStream`. В результате, объект класса `ReWriter` содержит два подобъекта класса `BufferedStream`. В данном случае это именно то, что требуется, поскольку нам и нужны два независимых объекта `BufferedStream` для потока ввода и потока вывода.

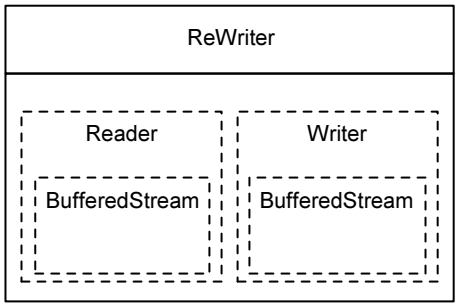


Рис. 7.5. Взаимодействие объектов при ромбовидном наследовании

В этом примере существенно и то, что по обоим маршрутам наследования класс `ReWriter` наследует классу `BufferedStream` опосредованно (через другой класс), а не напрямую. Повторное наследование от класса, от которого производный класс уже наследует напрямую, в C++ запрещено.

Внимательным нужно быть в тех случаях, когда функция-член подобного дважды наследуемого класса замещается или переопределяется в производных от него классах (которые являются базовыми для нижнего класса в иерархии). В таких случаях требуется явное разрешение конфликтов имен, аналогично тому, как это демонстрируется в листингах 7.2—7.3.

7.2. Виртуальные базовые классы

Иногда недопустимо, чтобы в составе объекта производного класса существовали два независимых подобъекта базового класса иерархии (как на рис. 7.5). Вместо этого требуется, чтобы все части объекта использовали один

экземпляр базового класса. Виртуальные базовые классы являются основным механизмом, позволяющим определить такой способ использования объектов базового класса.

Вернемся к примеру с представлением информации о студентах и попытаемся реализовать макет "более рафинированной" иерархии классов (рис. 7.6).

На вершине иерархии класс `AbstractStudent`. Он представляет абстракцию записи о студенте в самом общем плане. Классы `Storable` и `Displayable` увязывают отношения между сущностями объектно-ориентированной модели данных и отношения между подсистемами, оперирующими этими данными. Задачи, связанные с хранением информации и организацией логической структуры данных, могут в общем случае отличаться от задач представления информации, ее визуализации [Власовских, Пышкин, 2003]. Поэтому при написании модулей, оперирующих представлением информации, достаточно опираться на спецификации, показывающие те элементы модели данных, которые важны для корректного представления данных. Наоборот, те элементы спецификации, относительно которых нужна уверенность в том, что они не будут случайно модифицированы в неподходящем месте, должны быть скрыты. Целостная модель данных и поведения конструируется на основе класса `BasicStudent`, наследующего и классу `Storable`, и классу `Displayable`.

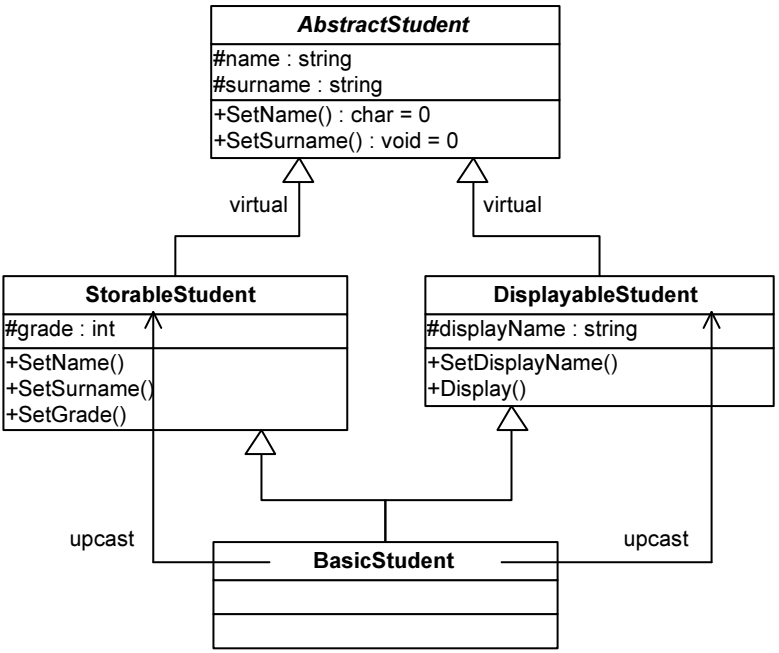


Рис. 7.6. Множественное наследование с виртуальным базовым классом

Несмотря на графическую схожесть этой диаграммы с диаграммой отношений классов `BufferedStream` — `Reader` — `Writer` — `ReWriter`, существенное различие между ними заключается в том, что объекты классов `Storable` и `Displayable` должны совместно использовать единственную копию `AbstractStudent`. В этом случае используется механизм виртуальных базовых классов (см. листинг 7.7).

Это означает, что виртуальный базовый класс будет представлен в производном классе одним и тем же совместно используемым объектом (рис. 7.7).

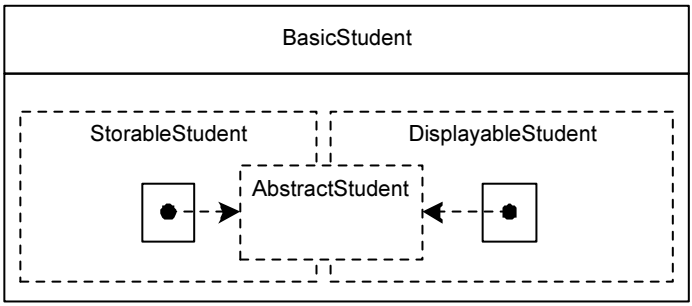


Рис. 7.7. Отношения между объектами при множественном наследовании с виртуальными базовыми классами

Иерархия наследования, при которой взаимодействие объектов организовано подобным образом, является наиболее сложной формой ромбовидного наследования. Листинг 7.7 представляет схематичную реализацию иерархии классов, представленной на рис. 7.6. Дополнительно в листинге 7.7 представлены модельные функции, работающие на разных уровнях абстракции при обработке информации о студентах.

Листинг 7.7. Наследование с виртуальными базовыми классами и отделение обработки данных на логическом уровне от обработки на уровне представления

```
// StudentsDiamond.h

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

class AbstractStudent {
protected:
    string name;
    string surname;
```

```
public:
    virtual void SetName( string ) = 0;
    virtual void SetSurname( string ) = 0;
};

class Storable : public virtual AbstractStudent {
protected:
    int  grade;

public:
    Storable( string nm, string surnm ) {
        SetName( nm );
        SetSurname (surnm);
    }

    void SetName( string nm ) {
        name = nm;
    }

    void SetSurname( string surnm ) {
        surname = surnm;
    }

    void SetGrade( int grd ) {
        grade = grd;
    }
};

class Displayable : public virtual AbstractStudent {
protected:
    string displayName;

public:
    void display() {
        cout << displayName << endl;
    }

    void SetDisplayName( string degree = "" ) {
        displayName = name;
        displayName = displayName + " ";
        displayName = displayName + surname;

        if( degree.length() != 0 ) {
            displayName = displayName + ", ";
            displayName = displayName + degree;
        }
    }
};
```

```
class BasicStudent : public Storable,
                     public Displayable {
public:
    BasicStudent( string nm, string surnm ) :
        Storable( nm, surnm ) {
    }
};

// StudentsDiamond.cpp
#include <cassert>

#include "StudentsDiamond.h"

typedef Storable *PStorable;
typedef Displayable *PDisplayable;
typedef BasicStudent *PBasicStudent;

void ProcessingStorable( PStorable storStud, int grade ) {
    storStud->SetGrade( grade );
}

void PrintingDisplayable( PDisplayable dispStud ) {
    dispStud->display();
}

void main() {
    PBasicStudent *all = new PBasicStudent[ 2 ];
    assert( all != 0 );

    all[ 0 ] = new BasicStudent( "John", "Smith" );
    assert( all[ 0 ] != 0 );
    all[ 0 ]->SetDisplayName();
    ProcessingStorable( all[ 0 ], 2 );

    all[ 1 ] = new BasicStudent( "George", "Black" );
    assert( all[ 1 ] != 0 );
    all[ 1 ]->SetDisplayName( "B.Sc" );
    ProcessingStorable( all[ 1 ], 3 );

    for( int i=0; i<2; i++ ) {
        PrintingDisplayable( all[ i ] );
    }
}
```

Функции `ProcessingStorable()` и `PrintingDisplayable()` листинга 7.7 являются крошечными моделями подсистем, каждая из которых работает с инфор-

мацией, доступной на соответствующем уровне. Первая подсистема имеет дело с информацией, нужной для обработки хранящихся в базе сведений. Вторая подсистема ограничена информацией, требующейся для корректного отображения этих сведений, при этом, не имея возможности изменять данные. Обратите внимание на неявные повышающие преобразования указателя на `BasicStudent` к указателю на `Storable` (при вызове функции `ProcessingStorable()`) и к указателю на `Displayable` (при вызове функции `PrintingDisplayable()`).

В данной реализации виртуальный базовый класс `AbstractStudent` не имеет явно определенного конструктора, поэтому конструктор по умолчанию генерируется автоматически. Если виртуальный базовый класс имеет явно определенный конструктор, не являющийся конструктором по умолчанию, то его вызов должен быть обязательно помещен в список инициализаторов конструктора для каждого класса из иерархии классов, производных от виртуального базового класса (листинг 7.8).

Листинг 7.8. Инициализация объектов при наследовании с виртуальными базовыми классами

```
#include <iostream>
#include <string>
#include <cassert>

using namespace std;

class SuperBase {
protected:
    string initializedBy; // Кем инициализирован?
public:
    SuperBase( string from ) : initializedBy ( from ) {
        // (Фиксируем, кем инициализирован)
    }
    virtual void WhoAmI() {
        cout << initializedBy << endl;
    }
};

class Derived1 : virtual public SuperBase {
public:
    Derived1() : SuperBase( "Derived1" ) {}
};

class Derived2 : virtual public SuperBase {
public:
```

```
    Derived2() : SuperBase( "Derived2" ) {}
};

class MostDerived : public Derived1, public Derived2 {
public:
    MostDerived() : SuperBase( "MostDerived" ) {}
};

class MostMostDerived : public MostDerived {
public:
    MostMostDerived() : SuperBase( "MostMostDerived" ) {}
};

typedef SuperBase *PSuperBase;

void main() {
    PSuperBase all[ 5 ];

    all[ 0 ] = new SuperBase( "SuperBase" );
    assert( all[ 0 ] != 0 );

    all[ 1 ] = new Derived1;
    assert( all[ 1 ] != 0 );

    all[ 2 ] = new Derived2;
    assert( all[ 2 ] != 0 );

    all[ 3 ] = new MostDerived;
    assert( all[ 3 ] != 0 );

    all[ 4 ] = new MostMostDerived;
    assert( all[ 4 ] != 0 );

    for( int ix = 0; ix < 5; ix++ ) {
        all[ ix ]->WhoAmI();
    }
}
```

Если бы класс `MostDerived` не содержал явного инициализатора `SuperBase`, компилятор не смог бы разобраться, какой инициализатор выбрать (в классе `Derived1` или в классе `Derived2`). Та же ситуация и с классом `MostMostDerived`. Таким образом, ответственность за инициализацию подобъекта `SuperBase` несет тот класс, объект которого создается, а значит — все производные от `SuperBase` классы должны содержать соответствующий инициализатор. Необычная реализация функции `WhoAmI()` позволяет убедиться в том, что объект, обеспечивший инициализацию `SuperBase`, и есть размещаемый в данный мо-

мент в памяти объект (что было бы совсем не так при обычном невиртуальном наследовании). Поэтому программа выведет следующее:

```
SuperBase
Derived1
Derived2
MostDerived
MostMostDerived
```

Например, размещение в динамической памяти объекта `all[4]` (имеющего тип `MostMostDerived`) форсирует инициализацию класса `SuperBase` с формированием строки `initializedBy`, которая получает значение `"MostMostDerived"`.

Указанное обстоятельство, связанное с необходимостью инициализации конструктора виртуального базового класса во всех производных классах (даже самых удаленных), требует от разработчика базового класса по возможности предоставлять конструктор по умолчанию с тем, чтобы явный вызов конструктора в инициализаторах производных классов мог быть опущен (по крайней мере, в наиболее типичных случаях инициализации).

Напомним, что в случае наследования без виртуальных базовых классов, конструктор базового класса может вызываться только в инициализаторе класса, непосредственно наследующего базовому классу *(см. гл. 3)*!

7.3. Дискуссия о множественном наследовании. Понятие интерфейса

С момента реализации множественного наследования в языке C++, а также в некоторых других языках, например, в языке Eiffel (см. [Meyer, 1992]), не прекращается дискуссия о проблемах множественного наследования [Stroustrup, 1994], [Труб, 2001], [Легалов, 2001]. Некоторые специалисты называют множественное наследование "оператором goto" 90-х годов [Eckel, 2000-1]. Основной вывод, который можно сделать из этих дискуссий, состоит в том, что написание такого кода является довольно непростым делом. Причем, независимо от того, является ли код, сконструированный с использованием множественного наследования, надежным, наглядным, непротиворечивым, расширяемым и сопровождаемым. В то же время многие задачи успешно решаются и без применения концепции множественного наследования (подтверждением тому служит успешность языка Java, не поддерживающего множественное наследование).

Несмотря на то, что C++ позволяет конструировать довольно сложные и запутанные иерархии классов на основе множественного наследования, Страуструп отмечает, что фундаментальной задачей множественного наследования

является реализация абстрактных классов. При этом один из базовых классов является открытым абстрактным классом, обеспечивающим интерфейс, другой — защищенным конкретным классом, представляющим детали реализации [Stroustrup, 2000] (рис. 7.8).

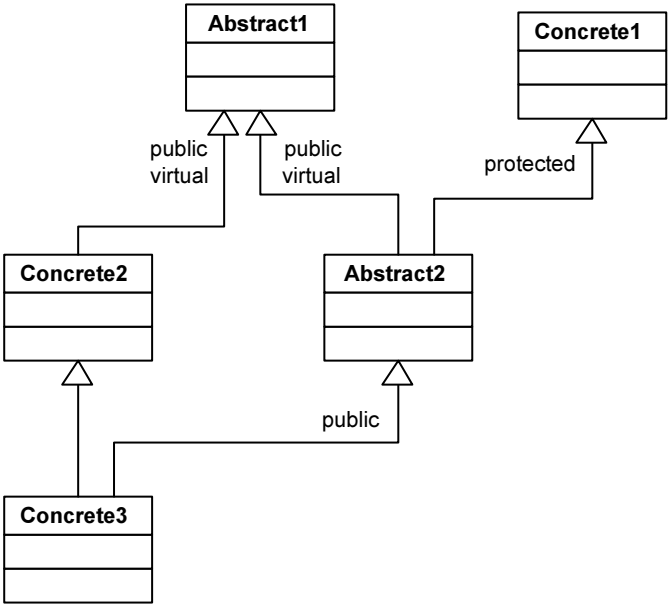


Рис. 7.8. Множественное наследование как инструмент реализации общих интерфейсов

Отметим, что зачастую класс может реализовывать не один, а несколько интерфейсов. Так, автомобиль реализует интерфейс рулевого управления, интерфейс трансмиссии, интерфейс управления световыми приборами и т. д.

При такой организации иерархии классов базовые классы неравноправны, а роли, исполняемые ими, различны. Одна из этих ролей может заключаться в том, чтобы обеспечить представление "чистого" интерфейса, вообще не содержащего элементов реализации. Это почти эквивалентно абстрактному классу, содержащему только открытые чисто виртуальные функции. Понимание такой роли ряда элементов конструируемой иерархии привело к появлению в объектно-ориентированных языках концепции интерфейса (interface), поддерживаемой на уровне языка.

В отличие от класса, интерфейс не предоставляет каких-либо элементов реализации — это в чистом виде форма без содержания. Интерфейс не просто не содержит переменные и функции-члены класса, он не может их содержать. В этом существенное отличие интерфейса от абстрактного класса. Абстрактный класс может не содержать элементов реализации, но может и содержать.

Наследование от абстрактных или конкретных классов влечет за собой необходимость хранения в памяти данных, определенных в базовом классе, в виде так называемого подобъекта производного класса. Некоторые особые ситуации в связи с управлением подобъектами были рассмотрены в предыдущей главе. Таким образом, при компиляции кода, содержащего наследование абстрактных классов, разработчик компилятора исходит из того, что абстрактный класс может содержать элементы реализации (хотя в действительности их может и не быть).

В интерфейсе же ничего подобного нет в принципе. Это означает, что если разрешить наследование только от одного базового класса и от произвольного числа интерфейсов, не будет никаких коллизий при управлении памятью, связанной с подобъектами базовых классов в производных классах (рис. 7.9). Другая особенность интерфейса, которая должна быть здесь выделена, заключается в том, что класс, определяемый как реализация интерфейса, обязан содержать реализации всех методов, объявленных в интерфейсе. Заметим, что интерфейсы могут конструироваться (и часто конструируются) на базе существующих интерфейсов обычным отношением наследования интерфейсов (которое называют отношением расширения интерфейса). Наследование, при котором разрешено построить производный класс на основе только одного базового класса, но при этом реализовать несколько интерфейсов, называется одиночным наследованием с интерфейсами.

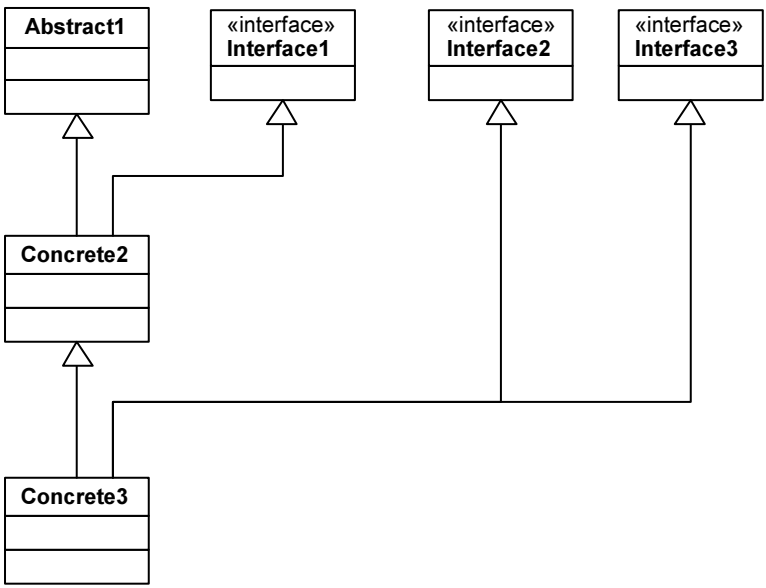


Рис. 7.9. Одиночное наследование классов и интерфейсы

Реализация семантики наследования с интерфейсами на языке C++

Несмотря на то, что стандартный C++ не содержит элементов синтаксиса, позволяющих объявить интерфейс, в программах на C++ можно реализовать семантику наследования, идентичную семантике одиночного наследования с интерфейсами. Рассмотрим в качестве примера переработанную версию иерархии классов, представленных в листинге 7.7. Результат реорганизации определений классов приведен в листинге 7.9.

Листинг 7.9. Реализация семантики наследования с интерфейсами на C++

```
//StudentsInter.h

#ifndef _StudentsInter_h_
#define _StudentsInter_h_

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

class IStorable { // Interface
public:
    virtual void SetName( string nm ) = 0;
    virtual void SetSurname( string surnm ) = 0;
    virtual void SetGrade( int grd ) = 0;
};

class IDisplayable { // Interface
public:
    virtual void display() = 0;
    virtual void SetDisplayName( string degree = "" ) = 0;
};

class AbstractStudent : /* implements */ public IStorable {
protected:
    string name;
    string surname;
    int grade;

public:
    AbstractStudent( string nm, string surnm ) {
        SetName( nm );
        SetSurname ( surnm );
    }
```

```
    void SetName( string nm ) {
        name = nm;
    }
    void SetSurname( string surnm ) {
        surname = surnm;
    }
    void SetGrade( int grd ) {
        grade = grd;
    }
};

class BasicStudent : /* extends    */ public AbstractStudent,
                     /* implements */ public IDisplayable {
protected:
    string displayName;

public:
    BasicStudent( string nm, string surnm ) :
        AbstractStudent( nm, surnm ) {
    }

    void display() {

        cout << displayName << endl;
        cout << "\tGrade: " << grade << endl;
    }

    void SetDisplayName( string degree = "" ) {
        displayName = name;
        displayName = displayName + " ";
        displayName = displayName + surname;

        if( degree.length() != 0 ) {
            displayName = displayName + ", ";
            displayName = displayName + degree;
        }
    }
};

#endif
```

На рис. 7.10 приведена иерархия классов и интерфейсов, соответствующая данному варианту реализации представления и обработки информации о студентах.

В качестве тестирующей программы может быть использована программа листинга 7.7. Никаких изменений, за исключением имени подключаемого заголовочного файла, в ней не требуется!

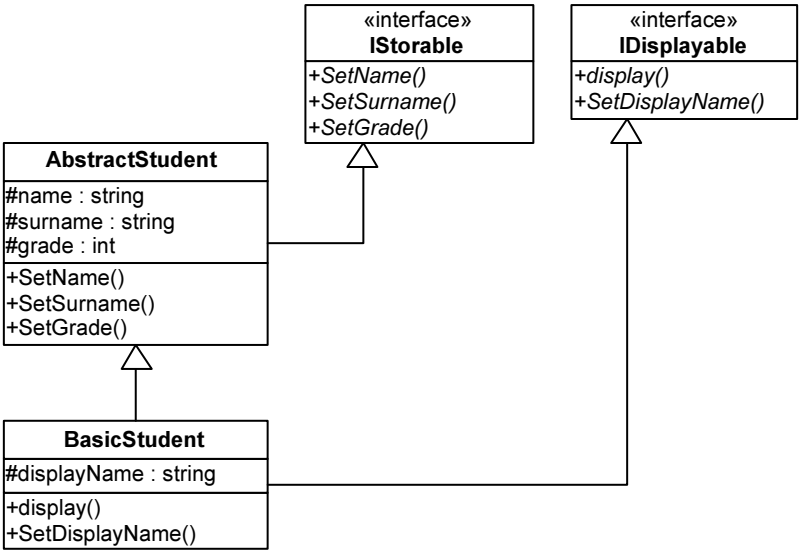


Рис. 7.10. Наследование и реализация интерфейсов

Некоторые специалисты полагают, что для выражения семантики интерфейса средствами языка C++ правильно использовать при объявлении интерфейса структурный тип (см., в частности, [Rogerson, 1997]):

```
struct IStorable { // Interface
    virtual void SetName( string nm ) = 0;
    virtual void SetSurname( string surnm ) = 0;
    virtual void SetGrade( int grd ) = 0;
};
```

В этом случае спецификатор `public` при объявлении сигнатур методов не требуется, поскольку члены структуры по определению доступны отовсюду. Если пойти еще дальше, то можно в одном из общих заголовочных файлов вашего проекта определить макроопределение следующего содержания:

```
#define interface struct
```

В этом случае определение интерфейса приобретет некоторую дополнительную наглядность (с первого слова определения ясно, что речь идет об интерфейсе, а не какой-то другой структуре):

```
interface IStorable {
    virtual void SetName( string nm ) = 0;
    virtual void SetSurname( string surnm ) = 0;
    virtual void SetGrade( int grd ) = 0;
};
```

Собственно, так поступили и разработчики Microsoft, включив подобное макроопределение в заголовочный файл objbase.h, поставляемый как часть Microsoft Win32 SDK.

Реализация семантики наследования с интерфейсами на языке C#

Для сравнения приведем реализацию иерархии классов, представленной на рис. 7.10, на языке C#. Язык C# является представителем языков, поддерживающих концепцию одиночного наследования классов и множественного наследования интерфейсов. Поскольку интерфейс содержит только сигнатуры методов, а не их реализацию, при наследовании интерфейса тип должен предоставить собственную реализацию унаследованных из интерфейса методов. Если тип не обеспечивает реализацию методов интерфейса, он является абстрактным типом, и, следовательно, экземпляры такого типа не могут быть созданы. При этом в программе могут существовать указатели на этот тип (если язык поддерживает указатели) или ссылки на него.

Анализ программы листинга 7.10, общее содержание которой соответствует примеру из предшествующего раздела, не составит большого труда, поскольку синтаксис C# во многом основан на синтаксисе C++.

Листинг 7.10. Одиночное наследование с интерфейсами средствами языка C#

```
using System;

namespace StudentsCSharp {

    public interface IStorable {

        void SetName( string nm );
        void SetSurname( string surnm );
        void SetGrade( int grd );
    }

    public interface IDisplayable {

        void display();
        void SetDisplayName( string degree );
    }

    public class AbstractStudent :  IStorable {

        protected string name;
        protected string surname;
        protected int  grade;
```

```
    public AbstractStudent( string nm, string surnm ) {

        SetName( nm );
        SetSurname( surnm );
    }

    public void SetName( string nm ) {

        name = nm;
    }

    public void SetSurname( string surnm ) {

        surname = surnm;
    }

    public void SetGrade( int grd ) {

        grade = grd;
    }
}

public class BasicStudent : AbstractStudent,
                            IDisplayable {

    protected string displayName;

    public BasicStudent( string nm, string surnm ) :
        base( nm, surnm ) {}

    public void display() {

        Console.WriteLine( displayName );
        Console.WriteLine( "\tGrade: {0}", grade );
    }

    public void SetDisplayName( string degree ) {

        displayName = name + " " + surname;
        if( degree.Length != 0 ) {

            displayName = displayName + ", " + degree;
        }
    }
}

public class Students {

    public static void ProcessingStorable(
        IStorable storStud, int grade ) {
```

```
            storStud.SetGrade( grade );
        }

        public static void PrintingDisplayable(
            IDisplayable dispStud ) {

            dispStud.display();
        }

        // ...

        public static void Main() {

            BasicStudent [] all = new BasicStudent[ 2 ];

            all[ 0 ] = new BasicStudent( "John", "Smith" );
            all[ 0 ].SetDisplayName( "" );
            ProcessingStorable( all[ 0 ], 2 );

            all[ 1 ] = new BasicStudent( "George", "Black" );
            all[ 1 ].SetDisplayName( "B.Sc" );
            ProcessingStorable( all[ 1 ], 3 );

            for( int i=0; i<2; i++ ) {

                 PrintingDisplayable( all[ i ] );
            }
        }
    }
}
```

Результат исполнения этой программы ничем не отличается от результата исполнения аналогичной программы на C++. Как явствует из приведенной программы, язык C# предоставляет встроенные средства, позволяющие определить интерфейс. При этом у программиста сохраняется возможность использовать абстрактные классы там, где это требуется. Таким образом, концепция интерфейса полностью отделена от концепции абстрактного класса, несмотря на то, что обеспечиваемые ими механизмы частично пересекаются.

Обратите внимание на методы класса `Students`. Метод `ProcessingStorable()` принимает в качестве аргумента ссылку на интерфейс `IStorable`. Неявное повышающее преобразование ссылки на объект класса `BasicStudent` к ссылке на `IStorable` успешно по определению класса `BasicStudent`. Класс `BasicStudent` наследует классу `AbstractStudent`, содержащему реализацию метода, определяемого интерфейсом `IStorable`.

Если в ходе исполнения программы происходит попытка обращения к интерфейсному методу на объекте класса, не предоставляющего реализацию дан-

ного интерфейса, генерируется исключение `InvalidCastException`. Возникновение и обработку данной ситуации иллюстрирует пример листинга 7.11.

Листинг 7.11. Возникновение и обработка исключения при вызове интерфейсного метода

```
using System;

namespace InterfaceErrorTest {

    public interface Interface1 {
        void f();
    }

    // Этот класс предоставляет реализацию интерфейса Interface1
    public class Implement : Interface1 {
        public void f() {
            Console.WriteLine( "Implement1.f() called" );
        }
    }

    // Этот класс не имеет отношения к интерфейсу Interface1
    public class NoImplement {
        public void f() {
            Console.WriteLine( "NoImplement.f() called" );
        }
    }

    public class App {
        // Функция принимает ссылку на объект,
        // поддерживающий интерфейс Interface1
        static void ExpectInterface1( Interface1 inter ) {
            inter.f(); // Вызывается метод, реализованный
                       // в классе, поддерживающем интерфейс
        }

        // Тестирующая программа
        static void Main() {
            Implement implObj = new Implement();
            ExpectInterface1( implObj ); // Верно: будет вызван метод
                                         // Implement.f()

            try {
                NoImplement noimpl = new NoImplement();
                // "Прикидываемся", что класс NoImplement
                // поддерживает интерфейс
```

```
                ExpectInterface1( (Interface1) noimpl );
                                // Ошибка: генерируется исключение
            }
            catch( InvalidCastexception e ) {
                Console.WriteLine( e.Message );
            }
        }
    }
}
```

Справедливость сказанного выше подтверждают результаты исполнения программы:

```
Implement.f() called
Specified cast is not valid
```

Описанные возможности контроля корректности преобразования типа во время выполнения автоматически обеспечиваются только в управляемых средах *(см. гл. 10)*. В программе на языке C++ для обеспечения подобного контроля исполнения программист должен воспользоваться явными понижающими преобразованиями типов и механизмом RTTI.

Конфликты имен и сигнатур при реализации интерфейсов

При определении типа, реализующего несколько интерфейсов, могут возникать ситуации (вообще говоря, нечасто), когда реализуемые интерфейсы содержат объявления методов с одинаковыми именами и сигнатурами. В этом случае необходимо явное указание типов интерфейсов при определении соответствующих методов (листинг 7.12).

Листинг 7.12. Разрешение конфликтов при реализации нескольких интерфейсов

```
// Пример на языке C#
using System;

namespace ResolvingInterConflicts {

    public interface Interface1 {
        void f();
    }

    public interface Interface2 {
        void f();
    }
```

```
    // Этот класс предоставляет реализацию интерфейсов
    // Interface1 и Interface2
    public class Implement : Interface1, Interface2 {
        // Реализация метода, определяемого интерфейсом Interface1
        // NB! Нет спецификатора доступа public или private!
        void Interface1.f() {
            Console.WriteLine( "Interface1.f() implemented" );
        }
        // Реализация метода, определяемого интерфейсом Interface2
        // NB! Нет спецификатора доступа public или private!
        void Interface2.f() {
            Console.WriteLine( "Interface2.f() implemented" );
        }
        // Реализация собственного метода класса
        public void f() {
            Console.WriteLine( "Implement.f() own method" );
        }
    }

    public class App {
        // Тестирующая программа
        static void Main() {
            Implement implObj = new Implement();

            // Так можно вызвать только собственный метод
            implObj.f();

            // Так вызывается реализация метода, определяемого
            // интерфейсом Interface1
            ((Interface1) implObj).f();

            // Так вызывается реализация метода, определяемого
            // интерфейсом Interface2
            ((Interface2) implObj).f();
        }
    }
}
```

В результате исполнения программы сначала вызывается собственный метод класса `Implement`, затем — методы, определяемые интерфейсами, о чем свидетельствуют печатаемые на консоль сообщения:

```
Implement.f() own method
Interface1.f() implemented
Interface2.f() implemented
```

Для вызова методов, определяемых каждым из поддерживаемых интерфейсов, необходимо осуществить явное преобразование типа. При этом следует обратить внимание на одну особенность реализаций методов, определяемых интерфейсами — спецификатор `public` отсутствует. Рихтер говорит, что эти методы "живут двойной жизнью: то они закрытые, то они открытые" [Richter, 2002]. В буквальном смысле эти методы закрыты, поскольку их невозможно вызвать на экземпляре класса `Implement`. Однако при преобразовании ссылки на объект к типу интерфейса доступны только те методы, которые определяются этим интерфейсом. Поэтому, когда ссылка `implObj` преобразуется к типу `Interface1`, единственным доступным методом-членом класса `Implement` является реализация `Interface1.f()`. В том же случае, когда ссылка `implObj` преобразуется к типу `Interface2`, единственным доступным методомчленом класса `Implement` является реализация `Interface2.f()`. А методы интерфейса являются открытыми по определению.

7.4. Роль интерфейсов в компонентном программировании

Интерфейсы являются одним из важнейших элементов, поддерживающих функционирование компонентной модели проектирования приложений. Взаимодействие независимых компонентов, реализуемое посредством взаимодействия поддерживаемых ими интерфейсов, является одним из базовых механизмов, лежащих в основе проектирования многокомпонентных приложений. Это связано с тем, что именно интерфейсы позволяют одним объектам единообразно работать с объектами различной природы в той их части, которая поддерживает определенный интерфейс (или интерфейсы).

Отметим, что концепции компонентного программирования не обязательно предполагают использование объектно-ориентированной модели данных и поведения, однако в большинстве случаев компоненты удобнее проектировать как классы или наборы классов. Поэтому, следуя тематике данной книги, в дальнейшем мы будем рассматривать компоненты, проектируемые при помощи классов *(см. гл. 12)*.

Проектирование компонентов практически обязательно предполагает повторное использование спроектированных компонентов. При этом клиентский код может не иметь доступа к исходному коду тех компонентов, с которыми он взаимодействует. Поэтому компонентное программирование может предполагать ограничение возможностей наследования реализации (implementation inheritance), или, иными словами, наследования классов. Это объясняется тем, что наследование реализации устанавливает довольно сильную зависимость реализации производных программных объектов от реали-

зации базовых объектов. Это означает, что изменения в базовом коде могут потребовать изменений в клиентском коде. Вместо наследования реализаций для достижения большей независимости одних компонентов от других, в основу взаимодействия компонентов может быть положено наследование интерфейсов (interface inheritance). Такая схема взаимодействия компонентов применяется, например, в COM (Component Object Model — модель компонентных объектов). *Информацию о механизмах реализации взаимодействия компонентов см. в гл. 12.*

Таким образом, понимание концепции интерфейса необходимо не только для успешного освоения объектно-ориентированного программирования как такового, но и для правильного использования технологий, использующих объектно-ориентированное программирование как основной инструмент проектирования кода.

7.5. Вопросы и упражнения

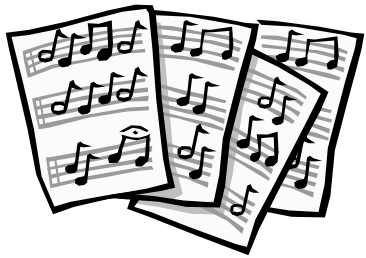
1. Страуструп считает, что члены класса (не являющиеся функциями), объявленные со спецификатором `protected`, обычно более подвержены злоупотреблениям, чем те, которые объявлены `private`, а объявление большого числа защищенных данных обычно свидетельствует об ошибках на этапе проектирования. Предложите исправления спецификации и элементов реализации классов из примера в листинге 7.7.
2. В статье [Труб, 2001] автор рассматривает пример определения классов, воплощающих концепции шахматных фигур. При этом, исходя из принципа "ферзь ходит как ладья и как слон", класс `Queen` (ферзь) наследует классам `Rook` (ладья) и `Bishop` (слон). Попытайтесь разобраться, в чем смысловая некорректность подобного определения. Предложите собственную иерархию классов, воплощающих абстракции шахматных фигур, и напишите тестирующую программу.
3. В *разд. 7.1* рассмотрены механизмы разрешения конфликтов имен при множественном наследовании. В случае одиночного наследования с интерфейсами, конфликт имен при наследовании реализации возникнуть не может, однако может возникнуть конфликт имен функций, сигнатуры которых определяются различными интерфейсами, что иллюстрирует пример листинга 7.12 (на языке C#). Теперь рассмотрите фрагмент аналогичной программы на языке C++:

```
#define interface struct

interface Interface1 {
    void f() = 0;
};
```

```
interface Interface2 {
    void f() = 0;
};

class X : public Interface1, public Interface2 {
    // ...
};
```

Реализуйте класс `X` таким образом, чтобы методы, определяемые различными интерфейсами, имели различные реализации. Напишите тестирующую программу, иллюстрирующую обращение к этим методам подобно реализации в листинге 7.12.

ГЛАВА 8



Пространства имен в связи с модульностью и иерархией

В современных языках, ориентированных на проектирование сложных многокомпонентных программ, одним из важных методов организации кода и обеспечения модифицируемости и повторного использования кода, является группирование программных объектов с учетом их логических связей. Механизм обеспечения реализации модульности программных проектов на основе логической организации программных объектов обычно связан с использованием так называемых пространств имен.

Прежде чем приступить к анализу вариантов реализации пространств имен в различных языках программирования, следует отметить, что использование пространств имен не обязательно предполагает объектную ориентированность разрабатываемых программных компонентов. Однако представляется вполне справедливым, что активное использование в проектировании пространств имен во многом обусловлено развитием объектно-ориентированного программирования. Косвенным подтверждением этого факта может служить история развития языка C++: первые реализации C++ не поддерживали пространств имен [Stroustrup, 1994].

8.1. Области действия и пространства имен

В программе на языке C используются следующие области видимости программных объектов: блок, функция, структура, единица трансляции (модуль), глобальное пространство имен. Использование единого глобального пространства порождает конфликты имен в тех случаях, когда приходится объединять фрагменты независимо разработанного кода.

В языке C++ наряду с перечисленными областями видимости используются области видимости класса. Класс также является, по существу, пространством имен, в котором определены члены класса. Однако, если необходимо

обеспечить простой механизм группирования и сокрытия имен, классы являются слишком сложным решением.

Пространства имен предоставляют механизм, обобщающий глобальные объявления в C и C++. Пространство имен — это именованная или неименованная открытая область действия имен. Это означает, что объявление программного объекта в пространстве имен не скрывает его для обращения из других областей видимости, а просто требует квалификации имени пространства имен для доступа к программному объекту (переменной, константе, типу, функции, другому пространству имен).

Программный проект: модули и интерфейсы

Другой аспект использования пространств имен тесно связан с модульностью при проектировании. Построение модуля всегда предполагает процесс логического группирования объявлений и определений в связи с решаемыми задачами и общностью используемых данных. Пространства имен предоставляют механизм отражения логического группирования программных объектов средствами языка программирования. Таким образом, необходимо различать два понятия: модуль-файл как сегмент физической организации многомодульной программы и модуль-пространство имен как сегмент логической организации.

Когда один модуль использует другой, нет необходимости знать все подробности реализации, таким образом, пространство имен часто существует в двух проявлениях:

- пространство имен как внешний интерфейс для пользователей. Такое пространство имен должно содержать объявления и определения, необходимые (и достаточные) для правильного использования функциональности, предоставляемой модулем;
- пространство имен как общая среда реализации программного проекта. Такое пространство имен должно содержать все объявления и определения, используемые в процессе разработки.

Рассмотрим в качестве примера проектирование текстового редактора, интегрируемого в какую-либо среду программирования. В современных средах разработки программного обеспечения текстовый редактор может обращаться к отдельным компонентам, присущим компиляторам, для того чтобы сделать процесс проектирования более приятным *(см. также гл. 2)*. В частности, это может быть необходимо для обеспечения следующих возможностей:

- визуализация исходных текстов в виде расцветки служебных слов и литералов;
- экспресс-анализ синтаксических конструкций и выделение фрагментов текста, написанных с нарушением правил синтаксиса;

- вывод подсказок об аргументах функции после того, как напечатана открывающая скобка, вывод подсказок о членах класса после того, как напечатана точка после имени объектной переменной и т. п.

Для того чтобы воспользоваться, скажем, лексическим анализатором с целью выяснить, является ли напечатанный идентификатор служебным словом или нет, не нужно знать все детали реализации лексического анализатора, достаточно иметь представление о том, какими функциями, типами, константами можно воспользоваться в связи с задачей лексического анализа кусочка текста.

Также и в ходе работы транслятора — синтаксический анализатор обращается к интерфейсу лексического анализатора, а не к его реализации (рис. 8.1). Явное отделение интерфейсной части модуля от его реализации способствует не только упрощению использования модуля, но и облегчает его модификацию.

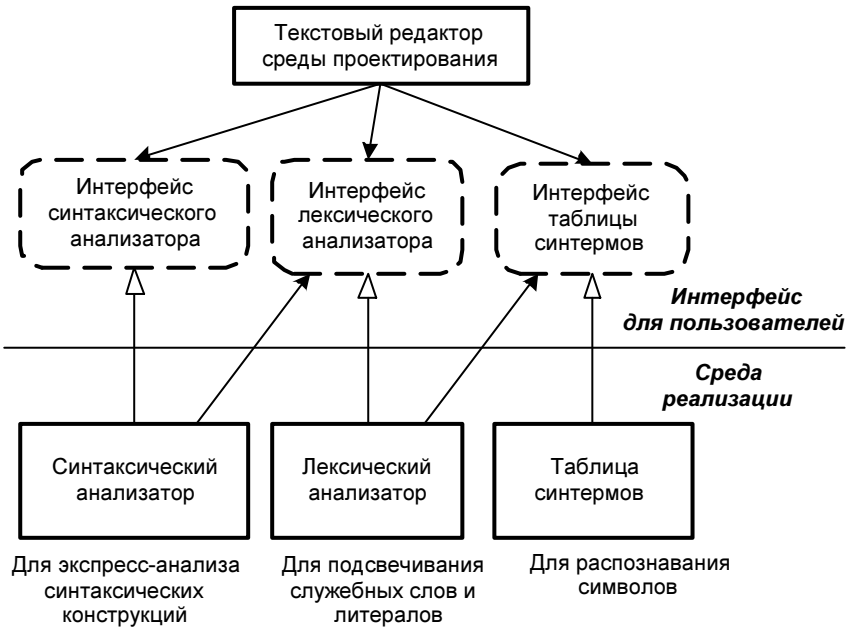


Рис. 8.1. Отделение внешнего интерфейса от общей среды реализации программного проекта

8.2. Пространства имен в C++

Та особенность языка C++, что в нем объединяются процедурное программирование в стиле C и объектно-ориентированное программирование, объясняет тот факт, что реализация пространств имен в C++ едва ли может рассмат-

риваться как универсальное решение. Однако основные элементы реализации концепции логического группирования программных объектов в том виде, в каком они поддерживаются в C++, схожим образом (с поправкой на особенности синтаксиса) реализованы и в других языках.

Помещение программных объектов в пространство имен

Определение пространства имен и помещение объявлений в пространство имен осуществляется с помощью специального служебного слова `namespace`.

Рассмотрим пример проектирования фрагмента синтаксического анализатора, ответственного за распознавание произвольных логических выражений в скобочной форме записи. Синтаксический анализатор `Parser` включает функции, обрабатывающие логическое выражение. Лексический анализатор `Lexer` обеспечивает считывание из входного потока очередной лексемы обрабатываемого текста. Нижеследующий текст иллюстрирует рассматриваемые определения:

```
// Пространство имен синтаксического анализатора
namespace Parser {

    void ParseExpression(); // Обработка выражения

    void ExprLeft (); // Обработка левой части выражения
    void ExprRight (); // Обработка правой части выражения

    // Любое логическое выражение может быть представлено как сумма
    // произведений логических выражений...
    void Term(); // Обработка слагаемого этой суммы произведений
    void Factor(); // Обработка сомножителя
    // ...
}

// Пространство имен лексического анализатора
namespace Lexer {

    class Token { // Очередная лексема сканируемого текста
        enum { IDENT,           // Идентификатор
               L_PARENT,        // Левая открывающая скобка
               R_PARENT,        // Правая открывающая скобка
               OR_OPERATOR,     // Операция ИЛИ
               AND_OPERATOR,    // Операция И
               NOT_OPERATOR,    // Операция НЕ
               ASSIGN_OPERATOR, // Операция присваивания
               // ...
```

```
        } ttype; // Тип лексемы
        // ...
    };

    Token GetToken(); // Считывание очередной лексемы из входного потока
    // ...
}

// Пространство имен модуля обработки ошибок
namespace Errors {

    class NoNameException {};
    class NoAssignException {}:
    // ...
}
```

Реализацию объявленных в пространстве имен функций и классов можно осуществить за пределами объявления пространства имен, используя явный квалификатор имени пространства имен (разумеется, само имя пространства имен должно быть в пределах области видимости), например:

```
void Parser::ParseExpression() {
    ExprLeft();
    ExprRight();
    // ...
    return;
}
```

Использование программных объектов, объявленных в пространстве имен

Имя из другого пространства имен можно использовать при помощи явного указания имени пространства имен:

```
void Parser::ExprLeft () {
    Lexer::Token token; // Использование имени типа Token
                        // из пространства имен Lexer

    token = Lexer::GetToken(); // Использование функции
                               // GetToken() из
                               // пространства имен Lexer

    switch( token.ttype ) {

        case Lexer::Token::IDENT: // Обработка идентификатора
                                  // ...
                                  break;
```

```
        default                 : throw NoNameException();
    }

    token = Lexer::GetToken();
    if( token.ttype != Lexer::Token::ASSIGN_OPERATOR )
        throw NoAssignException();

    return;
}
```

Если в тексте присутствует много обращений к объекту из другого пространства имен, то можно объявить этот программный объект в начале блока, используя *using-объявление*. Using-объявление вводит синоним для некоторого имени в локальную область действия:

```
void Parser::ExprLeft () {

    using Lexer::GetToken; // GetToken теперь является
                           // синонимом Lexer::GetToken

    using Lexer::Token; // Token теперь является
                        // синонимом Lexer::Token

    Token token;

    token = GetToken();
    switch( token.synType ) {

        case Token::IDENT : // Обработка идентификатора
                            // ...
                            break;
        default           : throw Errors::NoNameException();
    }

    token = GetToken();
    if( token.ttype != Token::ASSIGN_OPERATOR )
        throw Errors::NoAssignException();

    return;
}
```

Можно сделать доступными все имена из некоторого пространства имен. Для этого используется *using-директива*:

```
void Parser::ExprLeft () {

    using namespace Lexer;  // Доступны все имена из Lexer
    using namespace Errors; // Доступны все имена из Errors
```

```
    Token token;

    token = GetToken();
    switch( token.synType ) {

        case Token::IDENT : // Обработка идентификатора
                            // ...
                            break;
        default           : throw NoNameException();
    }

    token = GetToken();
    if( token.ttype != Token::ASSIGN_OPERATOR )
        throw NoAssignException();

    return;
}
```

Обратите внимание, что `using`-директива не вводит имена из указанного пространства имен в локальную область действия, а просто делает их доступными.

Несмотря на то, что во избежание конфликтов имен `using`-объявления и `using`-директивы следует делать как можно более локальными, в некоторых случаях доступность имен удобно объявить на уровне всего пространства имен. Например, синтаксический анализатор повсюду использует функцию `GetToken()` и класс `Token` лексического анализатора, а также функциональность обработчика ошибок. Поэтому достаточно разумным будет следующий вариант объявления пространства имен `Parser`:

```
namespace Parser {

    void ParseExpression();

    void ExprLeft ();
    // ...

    // Using-объявления, обеспечивающие возможность использования
    // программных объектов без явной квалификации пространства имен
    // Lexer
    using Lexer::GetToken;
    using Lexer::Token;

    // Using-директива, делающая доступными все имена из
    // пространства имен Errors
    using namespace Errors;
}
```

Пространство имен как общая среда реализации программного проекта и пространство имен как внешний интерфейс для пользователей

Созданное нами объявление пространства имен `Parser` является избыточным с точки зрения предоставления интерфейса для клиентских программистов, поскольку включает в себя все объявления. Если функция `ParseExpression()`, вероятнее всего, может быть вызвана извне, то остальные функции пространства имен `Parser` скорее относятся к деталям реализации. Оказывается, по правилам C++, одновременно могут существовать два пространства имен, и каждое из них может быть использовано там, где это удобнее. Иными словами, пространство имен можно рассматривать и как общую среду для программных объектов, относящихся к реализации синтаксического анализатора (пространство имен разработчика), и как внешний интерфейс для пользователей синтаксического анализатора (пространство имен пользователя).

Таким образом, могут существовать одновременно два определения пространства имен `Parser`:

```
// Файл ParserClient.h
// Заголовочный файл для пользователей

namespace Parser { // Интерфейс пользователя

    void ParseExpression();

}

// Файл ParserImplementation.h
// Заголовочный файл разработчика

#include "ParserClient.h"

namespace Parser { // Общая среда разработки

    void ParseExpression();

    void ExprLeft ();
    void ExprRight ();

    void Term();
    void Factor();
    // ...
}
```

Пространства имен: технические подробности

В данном разделе рассматриваются некоторые важные подробности создания и использования пространств имен: разрешение конфликтов имен, включение в пространство имен новых элементов, объединение пространств имен, использование пространства имен стандартной библиотеки (`std`) и некоторые другие возможности.

Разрешение конфликтов имен

Несмотря на то, что одной из целей внедрения в язык пространств имен является преодоление возможных конфликтов в глобальной области видимости имен, конфликты имен, тем не менее, возможны при подключении в одном и том же блоке одноименных объектов из разных пространств имен. Для обеспечения разрешения возможных конфликтов имен следует руководствоваться следующими правилами:

1. Имена, явно объявленные в пространстве имен (локальные имена, имена с явным квалификатором пространства имен или имена, подключенные посредством `using`-объявления), имеют приоритет по отношению к именам, сделанным доступными посредством `using`-директив.
2. Глобальные имена не имеют приоритета над именами из пространств имен, ставших доступными в глобальной области. Страуструп замечает, что этим обеспечивается значительная защищенность от случайных конфликтов имен и гарантируется отсутствие каких-либо выгод от "замусоривания" глобального пространства имен.

Нижеследующий пример иллюстрирует эти правила:

```
int y;

namespace A {
    void f();
    int x;
    int  y;
}

namespace B {
    void f();
}

void function1() {
    using namespace A; // Имена из A доступны
    using namespace B; // Имена из B доступны
```

```
    using A::f; // Явное объявление f() из A

    int x; // Действует локальное объявление из function1()
           // (скрывает A::x)

    f();   // Действует A::f()

    ::x = 1; // Глобальная x
           // NB! Операция :: без имени пространства имен
           // всегда адресует к глобальному пространству имен

    y = 10; // Ошибка: глобальная y или A::y?
}

void function2() {
    using A::x; // Явно объявляется x из A
    int x;      // Ошибка: x объявлено дважды
}
```

Псевдонимы пространств имен

Выбирая для пространств имен короткие имена, всегда есть риск получить конфликт с именем другого пространства имен. Длинные имена громоздки для восприятия, и их утомительно писать. Если в тексте программы длинное имя пространства имен приходится упоминать часто, можно воспользоваться псевдонимом пространства имен:

```
namespace VLN = Very_Long_Namespace_Name;

using VLN::SomeObject;
```

Дополнительную информацию об использовании псевдонимов пространств имен см. далее в подразделе "Пространства имен как механизм управления версиями".

Открытость пространств имен

Пространства имен C++ открыты. Это означает, что можно добавлять в пространство имен новые имена в различных объявлениях, которые могут находиться даже в разных файлах:

```
// Файл header1.h

namespace A {
    void f();
    void g();
}
```

```
// Файл header2.h
namespace A {
    void h();
} // Пространство имен A включает имена f, g, h.
```

Объединение пространств имен

Из предыдущего описания `using`-директив следует, что пространства имен можно объединять, например:

```
// Файл Lib1.h
namespace Lib1 {
    void f();
    // ...
}

// Файл Lib2.h
namespace Lib2 {
    void g();
    // ...
}

// Файл CommonLib.h
#include "Lib1.h"
#include "Lib2.h"

namespace CommonLib {
    using namespace Lib1;
    using namespace Lib2;
    // ...
}
```

Возможные конфликты имен разрешаются в этом случае при помощи `using`-объявлений.

Следует учитывать, что для определения элемента пространства имен следует знать имя исходного пространства имен, например:

```
void CommonLib::f() { // Ошибка:
                      // f() — не является элементом
                      // пространства имен CommonLib
    // ...
}

void Lib1::f() { // Правильно
    // ...
}
```

Вложенные пространства имен

> Конструкциям следует быть вложенными, если только нет сильных аргументов против этого.
> *Бьерн Страуструп*

Пространства имен могут объявляться внутри других пространств имен. Это позволяет реализовывать многоуровневые пространства имен, а также отделять код, выражающий общие концепции, от специфичного кода, решающего более узкие задачи. Нижеследующие фрагменты иллюстрируют создание вложенных пространств имен.

```
namespace Outer {
    class X { // Класс в пространстве имен Outer
        // ...
    };

    namespace Inner {
        class Y { // Класс в пространстве имен Outer::Inner
            // ...
        };
    }
}
```

При использовании объектов из пространств имен необходимо указывать все квалификаторы с возможным образованием многоуровневых имен, например:

```
void SomeFunction() {
    Outer::X obj1; // Правильно
    Outer::Y obj2; // ОШИБКА: в Outer нет Y
    Inner::Y obj3; // ОШИБКА: нет глобального имени Inner
    Outer::Inner::Y obj4; // Правильно
}
```

При записи using-объявлений и using-директив также применяются стандартные правила разрешения области видимости, причем `using`-директива, обеспечивающая доступность имен из вложенного пространства имен, не делает доступными имена из охватывающего пространства имен:

```
using namespace Outer::Inner;
void AnotherFunction() {
    X obj1; // Ошибка: X не находится в области видимости
            //         пространства имен Inner
    Outer::X obj2; // Правильно
    Y obj3; // Правильно
}
```

Пространства имен как механизм управления версиями

Пространства имен могут использоваться в качестве инструмента управления версиями [Stroustrup, 1994]. Предположим, что существует некоторый библиотечный компонент, содержащий в числе прочих объявлений и определений определение некоторого класса. Пусть заголовочный файл, предоставляющий интерфейс к библиотечным средствам, выглядит следующим образом:

```
// Файл lib.h
namespace Library { // Пространство имен, ориентированное на пользователя
                    // библиотеки
    namespace LibImplementation { // Пространство имен, содержащее
                                  // элементы внутренней реализации
        class InnerClass { // Некоторый класс, используемый разработчиком
                           // библиотеки, а не пользователем
            // ...
        };
    }
    class LibClass { // Библиотечный класс
        LibImplementation::InnerClass *ptr; // Закрытая часть
                                            // определения в связи с
                                            // деталями реализации
    public:
        // Открытая функциональность
        virtual void f1();
        virtual void f2();
        // ...
    };
}
```

Теперь рассмотрим возможный фрагмент клиентского кода, содержащий определение класса, построенного на основе библиотечного класса `LibClass`:

```
// Файл client.h
#include "lib.h"
using namespace Library;
namespace Client {
    class ClientClass : public LibClass {
    public:
        void f1();
        void f2();
        // ...
    };
}
```

Таким образом, клиентский код использует определения, содержащиеся в некотором внешнем коде. Развитие библиотечного кода может предполагать модификацию существующих классов (например, добавление новых членов класса или изменение реализации существующих методов класса). Причем даже такая, казалось бы, незначительная модификация, как изменение порядка следования виртуальных функций, может потребовать редактирование и перекомпиляцию клиентского кода. Это связано с тем, что изменение порядка следования виртуальных функций влечет изменение порядка адресов в таблице виртуальных функций, т. е., фактически, изменяется размещение объекта в памяти. Страуструп замечает по этому поводу: "Я мог бы посоветовать пользователям не наследовать подобным библиотечным классам таким образом, но ведь, несмотря на все предубеждения, они все равно будут поступать по-своему и жаловаться на необходимость перекомпиляции" [Stroustrup, 1994, разд. 17.4.4].

Вообще говоря, следует учитывать, что внешний код может быть реализован в виде статической библиотеки (файл с расширением lib) или в виде динамически подключаемой библиотеки DLL. В первом случае использование обновленного внешнего компонента невозможно, по меньшей мере, без перекомпоновки клиентского кода, поскольку связывание объектного кода клиентской программы и статической библиотеки происходит на этапе компоновки. Однако вопрос обязательности перекомпиляции исходных текстов остается открытым.

Второй случай предполагает размещение библиотечного кода в составе DLL-компонентов, которые могут совместно использоваться разными программами. При этом нередко возникают ситуации, когда более старым приложениям приходится работать с обновленными версиями DLL-компонентов, что иногда приводит к некорректной работе таких приложений с новыми версиями библиотек. Таким образом, проблема совместимости версий является довольно острой.

Абстрагируясь от особенностей размещения библиотечного кода, можно рассматривать два вида совместимости старого и модифицированного компонентов: совместимость на уровне исходного кода и совместимость на уровне двоичного кода.

Совместимость на уровне исходного кода предполагает, что приложения, которые зависели от прошлой версии, продолжают корректно работать с новой версией *после* перекомпиляции. Совместимость на уровне двоичного кода предполагает, что приложения, зависящие от прошлой версии, корректно работают с новой версией *без* перекомпиляции. Приведенный пример показывает, что в общем случае C++ не поддерживает совместимость на уровне двоичного кода.

В связи с этим возникает вопрос, можно ли организовать определения библиотечного кода таким образом, чтобы было позволено отложить ревизию и перекомпиляцию клиентского кода на более поздний (или более подходящий) момент.

Возможный вариант реализации библиотечного кода, содержащего "старую" и модифицированную версии класса `LibClass`, выглядит следующим образом:

```
// Файл release1.h
namespace Release1 { // Пространство имен, содержащее прежнюю версию
                     // библиотеки
    namespace LibImplementation { // Пространство имен, содержащее
                                  // элементы внутренней реализации
        class InnerClass { // Некоторый класс, используемый разработчиком
                           // библиотеки, а не пользователем
            // ...
        };
    }
    class LibClass { // Библиотечный класс
                     // (старая версия, нужная для совместимости
                     //  с существующим клиентским кодом)
        LibImplementation::InnerClass *ptr; // Закрытая часть
                                            // определения в связи с
                                            // деталями реализации
    public:
        // Открытая функциональность
        virtual void f1();
        virtual void f2();
        // ...
    };
}

// Файл release2.h
namespace Release2 { // Пространство имен, содержащее прежнюю версию
                     // библиотеки
    namespace LibImplementation { // Пространство имен, содержащее
                                  // элементы внутренней реализации
        class InnerClass { // Некоторый класс, используемый разработчиком
                           // библиотеки, а не пользователем
            // ...
        };
    }
    class LibClass { // Библиотечный класс
                     // (новая версия)
```

```
        LibImplementation::InnerClass *ptr; // Закрытая часть
                                            // определения в связи с
                                            // деталями реализации
    public:
        // В новой версии изменен порядок виртуальных функций
        // и появилась еще одна функция
        virtual void f2();
        virtual void f1();
        virtual void f3();
        // ...
    };
}
```

Для облегчения перехода на новую версию можно создать заголовочный файл, содержащий определение псевдонима для пространств имен:

```
// Файл lib.h
#include "release1.h"
#include "release2.h"

namespace Library = Release1; // Теперь Library — псевдоним для
                              // пространства имен Release1
```

Таким образом, все зависимости от версии локализованы в заголовочном файле lib.h. После того, как осуществлено приведение клиентского кода в соответствие с новой версией библиотеки, в заголовочном файле lib.h достаточно изменить псевдоним пространства имен:

```
namespace Library = Release2; // Теперь Library — псевдоним для
                              // пространства имен Release2
```

Подобный подход предоставляет инструмент разрешения конфликтов версий на уровне исходного кода, но вряд ли может служить универсальным средством, способствующим совместимости версий на уровне двоичного кода (см. также [Meyers, 1998, Item 34]).

Проблемы управления версиями не исчерпываются локализацией версий в рамках пространств имен. В *гл. 11 рассматриваются* аспекты управления версиями в связи с особенностями полиморфного поведения объектов базового библиотечного класса и производного клиентского класса, а также в связи с использованием компонентов DLL. В *гл. 12 вкратце анализируются* подходы к управлению версиями, используемые в компонентном программировании.

Стандартное пространство имен. Пространства имен и код, использующий библиотеки C

Элементы стандартной библиотеки C++ объявлены в пространстве имен `std` (которое называется стандартным пространством имен). Для того чтобы воспользоваться объявлениями из стандартного пространства имен, необходимо применить using-объявление или using-директиву. Так, в большинстве примеров, рассмотренных в книге, используются определения классов и функций в связи с реализацией средств потокового вывода C++:

```
#include <iostream> // Средства стандартного ввода-вывода
                    // библиотеки языка C++
using namespace std;
// ...
```

Напомним, что одна из задач, решавшихся в ходе проектирования языка C++, заключалась в поддержании пользовательской базы языка C. Необходимость обеспечения совместимости с кодом, не использующим пространств имен, потребовала некоторых специфических подходов.

Так, если в ходе проектирования программы возникает необходимость использования средств библиотек языка C (не C++), то можно воспользоваться заголовочными файлами, имена которых сконструированы из имен заголовочных файлов библиотек языка C добавлением литеры "c" в начале имени. Например, для функций из стандартной библиотеки ввода-вывода языка C это выглядит следующим образом:

```
#include <cstdio> // Средства стандартного ввода-вывода
                  // библиотеки языка C
using namespace std;
// ...
```

Обратите внимание на то, что в директиве препроцессора использовано имя файла без расширения. Имя файла без расширения является официальным стандартом и предназначено для использования совместно с концепцией пространств имен.

Для того чтобы обеспечить совместимость кода, использующего концепцию пространств имен, и кода на C/C++, не использующего пространств имен, разработчики стандартной библиотеки проделали следующее. Заголовочные файлы с расширением h обеспечивают неявное подключение стандартного пространства имен:

```
// Файл stdio.h
namespace std {
    // Все необходимые объявления
}
using namespace std; // Подключение пространства имен std
```

или

```
// Файл stdio.h
#include <cstdio>  // БЕЗ РАСШИРЕНИЯ
using namespace std; // Подключение пространства имен std
// Файл cstdio
namespace std {
    // Все необходимые объявления
}
```

Аналогичным образом подготовлены и другие стандартные заголовочные файлы.

Реализация заголовочных файлов подобным образом позволяет обеспечить работу с программами, не использующими концепцию пространств имен, без переделки.

Проблемы пространств имен в C++

Для того чтобы содержание этого и последующего разделов было понятнее, рассмотрим простой пример. Пусть имеются следующие файлы, содержащие текст программы на C++, которая может рассматриваться как "продвинутая версия" известной программы "Hello, World!" (листинг 8.1). Я не очень-то люблю начинать с нее курс программирования, однако отчего же не вернуться к этой популярной программе в ходе обсуждения?

Листинг 8.1. Пространства имен C++

```
/*
 * HelloDate.h
 */

#ifndef _HELLODATE_H_
#define _HELLODATE_H_

// Определяем пространство имен
namespace MyNamespace {

    // Класс HelloDate в пространстве имен MyNamespace
    class HelloDate {
```

```
    public:
        // Содержит единственный метод:
        // печатает приветствие, текущую дату и время
        void SayHello();
    };
}
#endif

/*
 * HelloDate.cpp
 *
 * Определение класса, содержащегося в модуле HelloDate
 */

#include <iostream>  // Для ввода-вывода
using namespace std; // подключение стандартного
                     // пространства имен

#include <time.h>    // Для работы с датой и временем

#include "HelloDate.h" // Интерфейсный файл модуля

using MyNamespace::HelloDate; // using-объявление

// Определение метода SayHello
void HelloDate::SayHello() {
    time_t now;

    time( &now ); // Получить текущее время

    cout << "Hello! It's: ";

    // Отобразить текущее время
    // (используются функции, объявленные в time.h)
    cout << asctime( localtime( &now ) ) << endl;
}
```

Предположим, что данный модуль откомпилирован в статическую библиотеку, и код библиотечного модуля содержится в файле HelloDate.lib. Интерфейс к этой библиотеке обеспечивается заголовочным файлом HelloDate.h, который и должен поставляться потребителю библиотеки вместе с кодом библиотечного модуля.

Текст простого консольного приложения, использующего библиотечный код, может выглядеть, например, следующим образом:

```
                              // В клиентском коде нужно:
#include "HelloDate.h"        // 1. Указать местонахождение
                              // объявлений библиотеки
```

```
using namespace MyNamespace; // 2. Указать используемое
                             // пространство имен
void main() {
    HelloDate hd;
    hd.SayHello();
}
```

Графически процесс обработки многомодульной программы, использующей пространства имен, представлен на рис. 8.2.

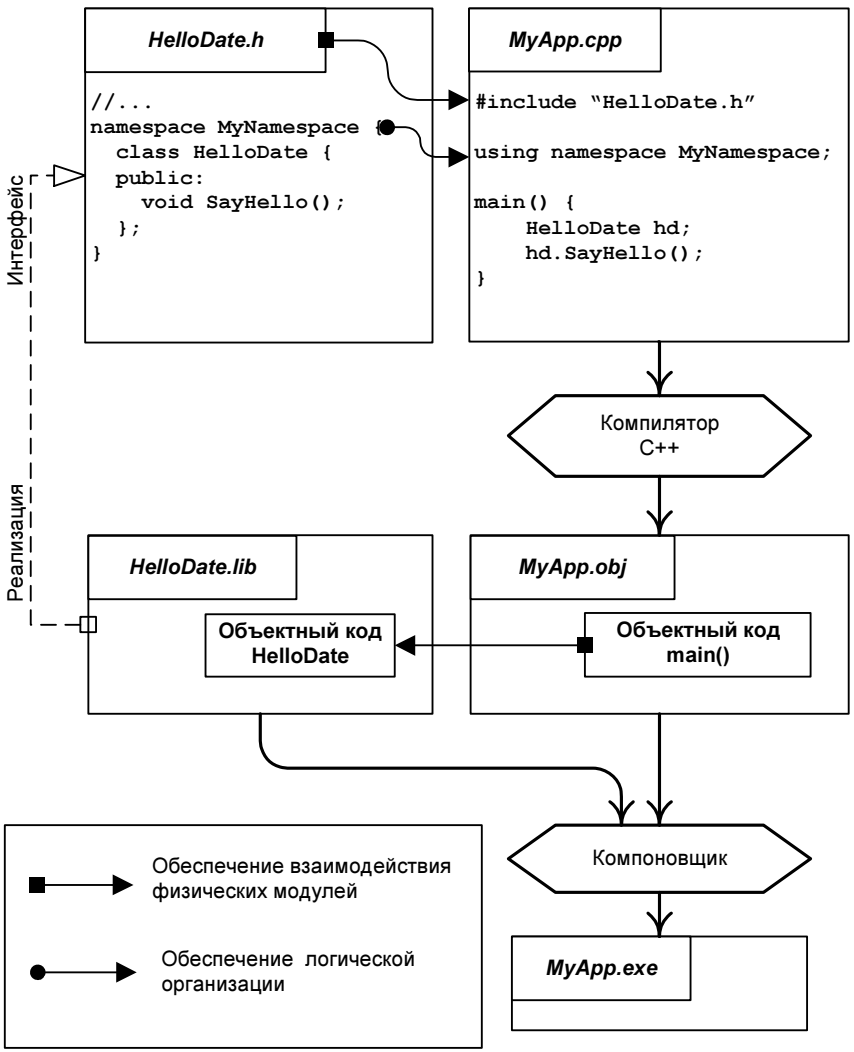


Рис. 8.2. Пространства имен и обработка программы в C++

Одна из проблем реализации пространств имен в C++ связана со следующим. Несмотря на то, что пространства имен предоставляют механизм отражения *логического* группирования программных объектов, пользователю недостаточно знать только имя пространства имен для того, чтобы воспользоваться каким-либо компонентом библиотеки. Необходимо иметь точное представление о том, в каком заголовочном файле (или файлах — ведь пространство имен может включать объекты из разных модулей) содержатся необходимые объявления. Таким образом, пространства имен C++ не обеспечивают полноценного разделения физической и логической структуры проекта. Открытым остается вопрос разрешения конфликтов имен самих пространств имен, а также проблема обеспечения совместимости версий на уровне двоичного кода *(см. также упражнения 6—7 в разд. 8.5)*.

В других языках программирования механизмы реализации концепции пространств имен несколько отличаются от модели C++. Далее в этой главе рассматриваются средства описания взаимосвязей физических и логических модулей в языках Java и C#.

8.3. Пространства имен в Java

В языке Java программные компоненты связываются в единую библиотеку посредством так называемых пакетов (packages). Опуская некоторые детали, отметим, что пакеты Java однозначно проецируются на структуру каталогов жесткого диска. При этом кодовые файлы (так называемые байт-коды Java), относящиеся к одному пакету, размещаются в одном каталоге. Подобный подход позволяет полностью отделить модульность на уровне физического размещения файлов от модульности на уровне логического группирования программных объектов.

Именование пакетов

Пакет является основным элементом механизма реализации иерархических пространств имен в Java. Работоспособность реализации пространств имен в Java в большой степени связана с обеспечением уникальности имен пакетов. Подход, используемый Java для обеспечения уникальности имен пакетов, наглядно представлен на рис. 8.3.

Имя пакета составляется из обращенного доменного имени компьютера, где хранятся классы библиотеки, и имени каталога на диске. Таким образом, уникальность имен пакетов гарантируется уникальностью доменных имен. Ориентированность Java на разработку программного обеспечения для Internet объясняет увязку механизма реализации модульности программных проектов с организацией сетевого взаимодействия компьютеров и программ.

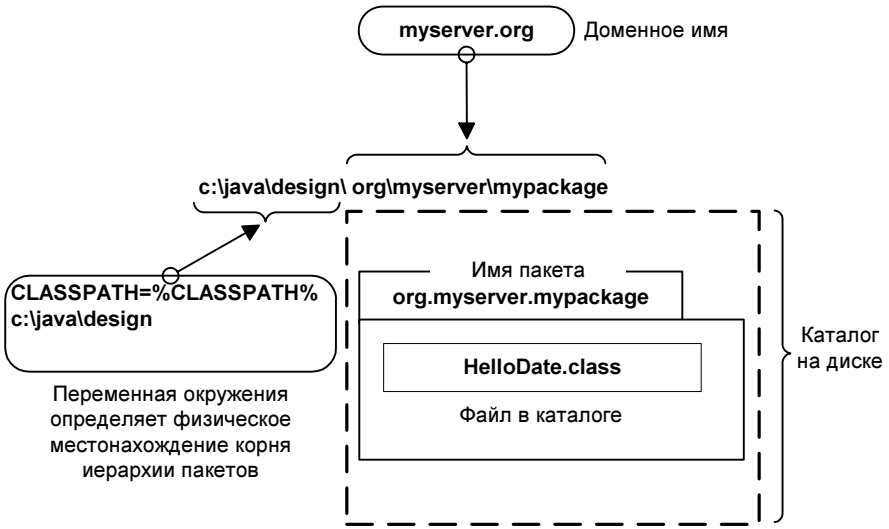


Рис. 8.3. Соответствие пакета Java и каталога на диске

В конечном счете файлы, составляющие пакет, размещаются на диске некоторого компьютера. Полное имя каталога, содержащего классы пакета, составляется из двух частей. Первая часть представляет собой путь к корневому каталогу иерархии пакетов, записанному в переменной окружения `CLASSPATH` (обычно таких путей бывает несколько). Собственно имя пакета образует завершающую часть полного имени каталога. Отметим, что информация, записанная в переменной окружения `CLASSPATH`, используется транслятором и виртуальной машиной Java при исполнении полученного в результате трансляции байт-кода.

Наиболее краткую и довольно точную характеристику пакетов Java находим у Эккеля: "имя пакета — путь и директория" [Eckel, 2000-2]. Все классы из библиотеки Java хранятся в пакетах, образующих довольно разветвленную иерархию. Для доступа к определениям классов в Java-программах используется директива `import`, например:

```
import java.util.Hashtable;
    // Импортировать класс Hashtable из пакета java.util

import java.util.*;
    // Импортировать все классы из пакета java.util
```

Таким образом, пользователю пакета (разработчику клиентского кода) не нужно вдаваться в детали формирования пакета и в особенности физической организации файлов, составляющих пакет. Для того чтобы воспользоваться

классами пакета, в клиентской программе необходимо указать только имя пакета.

Создание и использование пакета

Рассмотрим пример реализации на языке Java программы, подобной программе C++, представленной в листинге 8.1 *предыдущего раздела*. Структура взаимодействия модулей Java изображена на рис. 8.4.

Текст всех файлов, используемых в решении задачи, приводится в листинге 8.2.

Листинг 8.2. Особенности реализации пространств имен в языке Java

```
// Файл HelloDate.java
package org.myserver.mypackage; // Определяем наименование пакета
    // NB! Пакет должен быть размещен в каталоге
    //     %CLASSPATH%\org\myserver\mypackage

import java.util.Date;

// Класс, содержащий единственный метод,
// печатающий приветствие, дату и текущее время
public class HelloDate {
    public void SayHello() {
        System.out.print( "Hello, it's: " );
        System.out.println( new Date() );
    }
}

// Файл MyApp.java
import org.myserver.mypackage.*; // Используем все определения
                                 // из указанного пакета

// Класс,содержащий метод main (точку входа в приложение)
public class MyApp {
    public static void main( String[] args ) {

        // Создает объект класса HelloDate
        HelloDate hd = new HelloDate();

        // Вызываем метод SayHello
        hd.SayHello();
    }
}
```

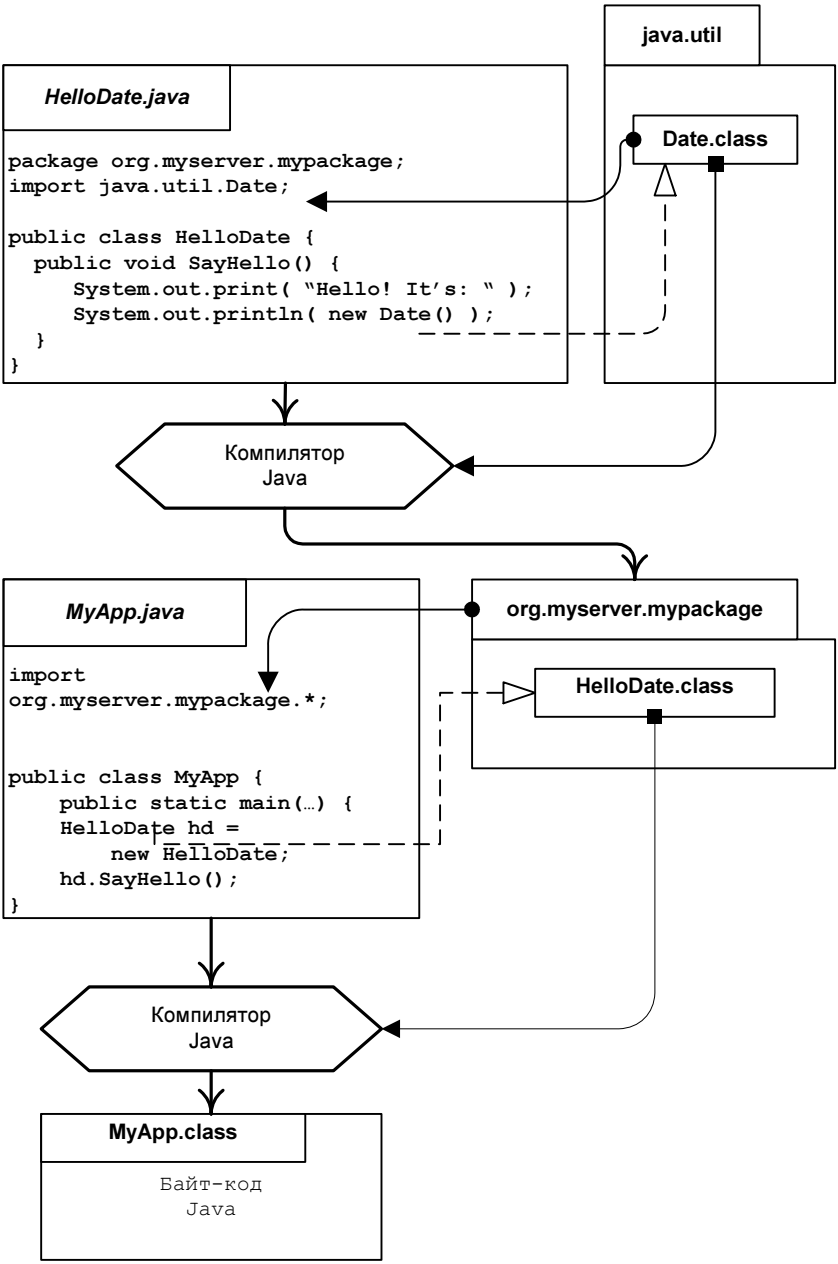


Рис. 8.4. Пространства имен и модульность в языке Java

Обратите внимание, что при определении класса `HelloDate` (реализующего функциональность, аналогичную одноименному классу из проекта на C++) используется класс `Date` из пакета библиотеки Java java.util. Благодаря директиве `package`, в результате компиляции Java-файла HelloDate.java получается пакет `org.myserver.mypackage`. В примере, иллюстрируемом на рис. 8.4, пакет содержит всего один класс `HelloDate.class`. В общем случае пакет является контейнером для набора классов, которые могут быть упакованы в архив JAR (Java ARchive) специальной утилитой.

Программа MyApp.java моделирует клиентское приложение, использующее класс из пакета `org.myserver.mypackage`. Директива `import` обеспечивает возможность использования класса `HelloDate`, байт-код которого содержится в указанном пакете. Однозначное соответствие между именем пакета и именем каталога на диске позволяет в исходном тексте использовать только имена пакетов. Пакеты при этом являются инструментом логической организации классов Java. Все явные и неявные ассоциации, отражающие взаимодействие физических модулей, относятся к этапу компиляции и системного администрирования.

Отметим дополнительно, что результатом работы компилятора Java является байт-код, требующий для своего исполнения виртуальной машины Java (JVM). О подробностях работы с пакетами Java см., например, в [Naughton, Schildt, 1999], [Eckel, 2000-2].

Примечание

Для того чтобы проект, представленный в листинге 8.2, успешно компилировался и обрабатывался Java-машиной, не забудьте о необходимости правильного задания значения переменной окружения `CLASSPATH`, которая должна содержать путь к каталогу `\org\myserver\mypackage` на вашем компьютере. Обратите внимание, что в каталоге проекта на компакт-диске содержатся batch-файлы, обеспечивающие установку значения переменной окружения и вызов компилятора Java и интерпретатора байт-кода. Для работы с примером ознакомьтесь с содержимым файла readme.txt в корневом каталоге проекта (см. также описание содержимого компакт-диска в *приложении 3*).

Возможность конструирования многоуровневых имен пакетов позволяет реализовать иерархию пакетов. При этом на физическом уровне иерархия пакетов соответствует системе каталогов на диске. В связи с этим имена пакетов следует тщательно продумывать. С одной стороны, это обусловлено проектными соображениями, поскольку нельзя переименовать имя пакета, не переименовывая каталоги, в которых хранятся классы пакета. С другой стороны, пакет является основной формой распространения кода Java. Таким образом, изменение имени пакета повлечет за собой необходимость изменения и перекомпиляции всего клиентского кода, использующего данный пакет.

Единая концепция размещения и именования пакетов, используемая в Java, позволяет успешно и в едином стиле объединять код, написанный разными разработчиками.

Права доступа в связи с модульной организацией

Пакет Java является своего рода контейнером для классов и интерфейсов, входящих в состав пакета. Во многих случаях помещение программных объектов в один пакет означает, что они имеют некоторую общность в связи с реализуемой функциональностью или используемыми ресурсами. Отражением этого факта является реализация в Java концепции доступа по умолчанию. Если для элементов класса отсутствует явная спецификация доступа (закрытый, защищенный или открытый), то действует правило, в соответствии с которым этот элемент доступен в классах, входящих в состав того же самого пакета. За пределами пакета эти элементы не видны. Доступность членов класса извне полностью регулируется явными спецификаторами доступа (по существу, точно так же, как и в языке C++).

Доступ по умолчанию является некоторой аналогией дружественного доступа в C++ (и в то же время — альтернативой ему!). Доступ по умолчанию не требует никакой явной спецификации, но при этом накладывает дополнительные ограничения доступности членов класса только в пределах модуля. Такое решение имеет безусловную внутреннюю логику. Если классы, не находящиеся в отношении наследования, должны быть "приоткрыты" друг другу, кажется естественным помещать их в один пакет. Таким образом, доступ по умолчанию не просто предоставляет механизм, упрощающий взаимодействие классов, входящих в один пакет, но стимулирует программиста к тщательному продумыванию модульной структуры приложения.

8.4. Пространства имен в C#. Введение в архитектуру приложения .NET

В платформе Microsoft .NET реализация пространств имен главным образом определяется особенностями организации среды исполнения программ .NET — Common Language Runtime (CLR). Поэтому пространства имен C# — это, по существу, пространства имен CLR. Так же как и в Java, в основе реализации модульности в платформе .NET лежит иерархическая модель организации пространств имен, упрощающая использование существующих типов и создание на их основе новых типов. При этом пространства имен CLR не требуют обязательной привязки к структуре файлов и каталогов на диске.

В языке C# синтаксис определения пространств имен сходен с синтаксисом C++. Однако элементами пространств имен являются не объявления, а определения программных объектов.

Здесь следует отметить, что CLR работает не с отдельными модулями, а с так называемыми сборками (assemblies). Сборки являются единицами логического группирования модулей, а также наименьшей единицей повторного использования, распространения, безопасности и отслеживания версий кода. Каждая сборка содержит информацию обо всех модулях, входящих в состав сборки, а также о внешних сборках, содержащих ресурсы, используемые данной сборкой. Подобная самоописываемость сборок существенно упрощает инсталляцию программ .NET, их распространение, а также взаимодействие компонентов .NET между собой.

Для того чтобы воспользоваться определением (класса) из пространства имен, внешнего по отношению к анализируемому коду (фактически, это означает — воспользоваться определением из внешней сборки), требуется выполнить следующие действия:

1. Использовать using-директиву в исходном тексте C# для подключения пространства имен.
2. Сделать сборку, содержащую указанное пространство имен, известной компилятору (во многих случаях имя пространства имен совпадает с именем сборки, хотя это и не обязательно). Как результат, в метаданные сборки заносится информация о внешней сборке.

Таким образом, аналогично Java, для использования того или иного компонента библиотеки, требуется знать пространство имен, содержащее нужный тип. Директива для подключения пространства имен выглядит в C# следующим образом:

```
using System.Collections; // Позволяет использовать типы
                          // из пространства имен
                          // System.Collections
```

Продолжим рассмотрение примера программы, печатающей информацию о текущих дате и времени. Рис. 8.5 иллюстрирует особенности взаимодействия модулей при обработке C#-программы, аналогичной двум предшествующим примерам на C++ и Java.

Листинг 8.3 содержит все определения, необходимые для реализации проекта подобной структуры.

Листинг 8.3. Модульность и пространства имен в C#

```
// Файл HelloDate.cs
// (компилируется в составе сборки HelloDate.dll)
using System;
```

```
namespace MyNamespace // Определяем пространство имен
{
    // Класс, содержащий единственный метод,
    // печатающий приветствие, дату и текущее время
    public class HelloDate
    {
        public void SayHello()
        {
            Console.Write( "Hello! It's: " );
            Console.WriteLine( DateTime.Now.ToString() );
        }
    }
}

// Файл MyApp.cs
// Пример пользовательской программы,
// использующей сборку HelloDate.dll
using System; // Используем пространство имен System
using MyNamespace; // Используем пространство имен MyNamespace
                   // (из сборки HelloDate.dll)

class MyApp
{
    static void Main( string[] args )
    {
        HelloDate hd = new HelloDate();
        hd.SayHello();
    }
}
```

Аналогично подходу, используемому в Java, взаимосвязи модулей в связи с логической организацией (namespace, using) отделены от взаимосвязей модулей в связи с физической организацией. Информация о сборке указывается посредством задания одного из ключей компилятора, в исходном тексте ссылок на сборку нет. При использовании интегрированной среды проектирования программисту обычно не приходится вручную определять ключи компилятора: ссылка на внешнюю сборку обеспечивается специальной командой среды. При этом модель, используемая в C#, по своей гибкости сопоставима с моделью C++, поскольку, с одной стороны, каждая сборка может содержать несколько пространств имен, с другой стороны, одно пространство имен может быть рассредоточено по нескольким сборкам.

В рассмотренном примере сборка HelloDate.dll является частной, или закрытой. Сборка такого типа нужна, если она используется одним приложением,

поэтому может размещаться в каталоге установки соответствующего приложения. При проектировании многокомпонентных приложений возникает необходимость создавать сборки, совместно используемые многими приложе-

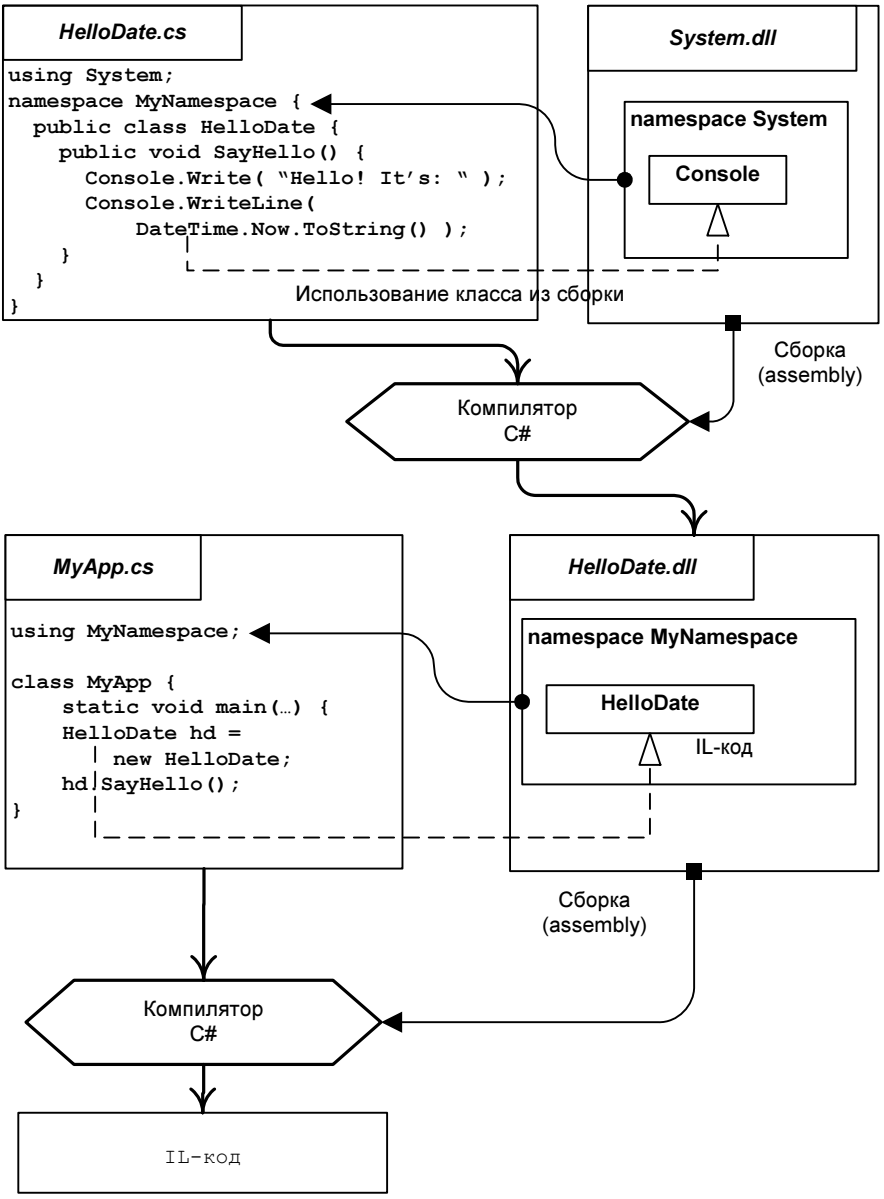


Рис. 8.5. Пространства имен и модульность в C#

ниями. Такие сборки называются общими сборками, или сборками с глобальным развертыванием. Для избежания случайных конфликтов имен сборок и самих пространств имен, сборки с глобальным развертыванием должны иметь уникальное имя. Уникальность и криптографическая стойкость имен общих сборок обеспечивается специальным механизмом строгого именования сборок. Общие сборки помещаются в специальный каталог (глобальный кэш сборок), автоматически проверяемый CLR при обнаружении ссылки на сборку. *Информацию об элементах архитектуры приложения .NET см. в разд. 12.5.*

Замечание

Для отражения того факта, что в сборку, как правило, объединяются взаимодействующие классы, по аналогии с доступом в пределах пакета Java, CLR поддерживает доступ в пределах сборки. В языке C# для этой цели служит спецификатор `internal`.

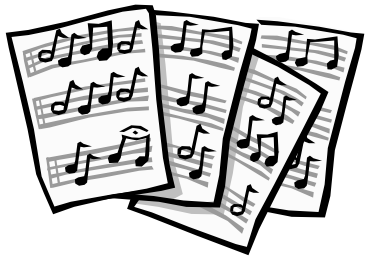
Подводя итог, необходимо отметить, что, несмотря на видимое сходство организации иерархии пространств имен в Java и в Microsoft .NET Framework, механизмы реализации взаимодействия объектов пространств имен и модулей имеют существенные отличия. Это затрагивает и отделение модулей, используемых внутри конкретного проекта, от модулей, нацеленных на совместное использование, и обеспечение уникальности именования модулей, и механизмы управления версиями при развертывании совместно используемого кода.

8.5. Вопросы и упражнения

1. Какие области видимости программных объектов в языке C++ Вы можете перечислить? Какие особенности пространства имен как области видимости Вы можете назвать?
2. Каким образом в C++ обеспечивается работа с исходными текстами на C или C++, написанными без использования концепции пространств имен?
3. В чем заключается отличие using-объявления от using-директивы в C++? Что имеет приоритет при одновременном использовании using-объявления и using-директивы для одноименных программных объектов? Как Вы считаете, что следует, как правило, использовать в программах для обеспечения доступности имен программных объектов: using-объявления или using-директивы? Обоснуйте свое мнение. Приведите примеры исключений из этого правила.
4. Почему глобальные переменные не имеют приоритета над переменными, ставшими известными в блоке благодаря using-объявлению или using-директиве?

5. Напишите законченную версию Garland Application *(см. гл. 2)*, организованную в соответствии с рис. 2.15. Обоснуйте Ваши проектные решения в связи с логической и физической структурой программного проекта.
6. Обсудите с коллегами мнение автора этой книги о проблемах реализации пространств имен в языке C++ *(см. разд. 8.2)*.
7. Разработайте тестовую программу, которая могла бы служить иллюстрацией материала, рассматриваемого в *подразделе "Пространства имен как механизм управления версиями" разд. 8.2*. Реализуйте пример библиотечного кода, размещаемого в статической библиотеке, и напишите тестовое клиентское приложение. Обсудите проблемы, возникающие при изменении библиотечного кода. Удается ли обеспечить совместимость клиента и библиотеки на уровне двоичного кода?

ГЛАВА 9



Введение в обобщенное программирование

Для начала обсуждения материала этой главы нам придется вернуться к примеру класса `QueueInt`, реализующего абстракцию стандартной очереди (см. примеры в листингах 3.3, 3.4). Одним из недостатков имеющейся реализации является то, что этот класс не представляет по-настоящему абстрактную модель очереди, поскольку позволяет хранить в ней только целочисленные данные. Имея в виду, что очередь представляет собой форму организации данных разных типов, а не способ работы с данными конкретного типа, было бы неплохо иметь возможность определить класс очереди без привязки к конкретному типу хранящихся в ней данных. Модель программирования, допускающую подобную реализацию, обычно называют обобщенным программированием. Основная область применения обобщенного программирования — реализация алгоритмов, выражаемых без привязки к конкретным типам обрабатываемых данных.

В этой главе мы рассмотрим механизмы реализации обобщенного программирования в языках C и C++.

9.1. Обобщенное программирование без использования шаблонных функций

Основным механизмом, обеспечивающим обобщенное программирование в C++, являются шаблонные функции, классы и их специализации. Однако это не единственно возможный подход к реализации обобщенных алгоритмов. Так, от стандартной библиотеки языка C стандартная библиотека C++ унаследовала обобщенные реализации быстрой сортировки `qsort()` и двоичного поиска `bsearch()`. Они могут использоваться для работы с массивами, тип которых не имеет конструкторов копирования и деструкторов, определенных пользователем. Рассмотрим этот подход на примере реализации алгоритма

сортировки простыми обменами (который выбран для иллюстрации из-за его компактности и логической простоты).

Определение функции сортировки простыми обменами для сортировки массива целых чисел

Напомним реализацию функции сортировки массива целых чисел:

```
void BubbleSortInt( int *array, int n ) {

    for( int i = 1; i < n; i++ )
        for( int j = n-1; j>=i; j-- )
            if( array[ j ] < array[ j-1 ] ) {

                int copy = array[ j ];
                array[ j ] = array[ j-1 ];
                array[ j-1 ] = copy;
            }
}
```

Эта простейшая реализация, вероятно, не требует никаких пояснений. Напомним, что в ходе сортировки простыми обменами на каждом `i`-м проходе последовательно сравниваются пары соседних элементов массива, начиная с пары, составленной из элементов с индексами `n-1` и `n-2`, и заканчивая парой элементов с индексами `i` и `i-1`. При этом `i` пробегает значения от `1` до `n-1` включительно.

Если порядок следования элементов рассматриваемой пары нарушен (в данной версии массив `array` сортируется по неубыванию), то элементы меняются местами. Не позднее чем за `n-1` просмотров массива, он оказывается упорядоченным.

Определение универсальной функции сортировки простыми обменами

Чтобы иметь возможность воспользоваться этим алгоритмом для сортировки массивов, состоящих из данных других типов, нужно определить две вещи:

- тип функции сравнения элементов пользовательского типа;
- функцию сортировки, использующую эту функцию сравнения.

Возможная реализация версии функции сортировки, адаптируемой к обработке последовательностей данных разных типов, представлена в листинге 9.1.

Листинг 9.1. Обобщенная функция сортировки в стиле языка C

```
// Определение типа функции, сравнивающей два значения
// пользовательского типа
typedef int (*CompareFunctionType)( const void* pArg1,
                                    const void* pArg2 );
    // Функция должна возвращать:
    //    отрицательное значение,
    //           если значение *pArg1 меньше значения *pArg2;
    //    положительное значение,
    //           если значение *pArg1 больше значения *pArg2;
    //    0,     если значения *pArg1 и *pArg2 равны.

// Обобщенная реализация функции сортировки простыми обменами
void BubbleSortGeneric(
    void *begin, // Адрес начала сортируемого массива
                 // (любой указатель может быть присвоен
                 //  указателю на void)
    int n,       // Число элементов сортируемого массива
    int elemSize,// Размер одного элемента в байтах
    CompareFunctionType compare)
                 // Указатель на функцию сравнения
{
    // Преобразование указателя на начало массива
    // к типу char*
    // NB! Необходимо, т. к. для указателя на void
    //     не разрешены арифметические операции
    char *base = static_cast <char *> ( begin );

    for( int i = 1; i < n; i++ ) {
        for( int j = n-1; j>=i; j-- ) {

            char *basej = base+j*elemSize;           // base[ j ]
            char *basejmin1 = base+( j-1 )*elemSize; // base[ j-1 ]

            if ( compare( basej, basejmin1 ) < 0 ) {

                // Меняем местами base[ j ] и base[ j-1 ]
                // (переставляем байты)
                for( int curByte = 0;
                         curByte < elemSize;
                         curByte++ ) {

                    char byteCopy = basej[ curByte ];
                    basej[ curByte ] = basejmin1[ curByte ];
```

```
                    basejmin1[ curByte ] = byteCopy;
                }
            }
        }
    }
}
```

Схема, иллюстрирующая управление данными при использовании функции `BubbleSortGeneric()`, приведена на рис. 9.1.

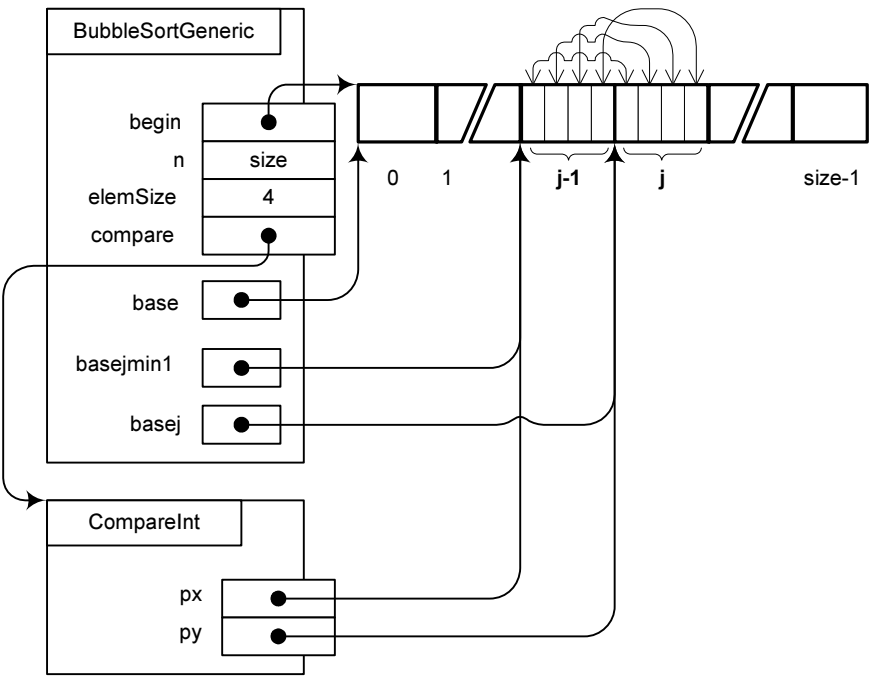


Рис. 9.1. Обобщенное программирование средствами языка C (на примере функции сортировки простыми обменами)

Для того чтобы воспользоваться этой функцией для сортировки массива данных произвольного типа, необходимо определить функцию, обеспечивающую реализацию семантики сравнения двух значений этого типа и имеющую интерфейс, определяемый типом `CompareFunctionType` (обязательно с указанными типами аргументов). Например, для сортировки целых чисел нужно написать следующий код:

```
int CompareInt( const void* px, const void *py ) {
   return *( static_cast <const int*>( px ) ) -
```

```
        *( static_cast <const int*>( py ) );
}

void SortInt( int *a, int length ) {

   BubbleSortGeneric( a, length, sizeof( int ), CompareInt );
}
```

Недостаток такого стиля обобщенного программирования (единственно возможного при программировании на языке C) заключается в отсутствии какой-либо информации о типе сортируемых элементов. Преобразование любого указателя к указателю на `void` по определению успешно, поэтому нет никакой гарантии, что на всех этапах используются одни и те же типы. Функция сравнения, требуемая для работы `BubbleSortGeneric()`, не такая уж наглядная, поскольку обязана оперировать указателями на `void`. Таким образом, из объявления функции не очевидно, что она обеспечивает реализацию семантики сравнения для данных конкретного пользовательского типа.

Другим недостатком является то, что реализация функции `BubbleSortGeneric()` в стиле языка C требует аккуратной работы с отдельными байтами сортируемых данных, и поэтому менее наглядна и, следовательно, существенно сложнее для восприятия, чем специализированная версия `BubbleSortInt()`.

9.2. Шаблонные функции

Шаблоны (templates) являются средством языка, обеспечивающим поддержку обобщенного программирования, основанного на использовании типов в качестве параметров. Использование шаблонных функций позволяет получить реализации, которые нагляднее обобщенных функций в стиле C, безопасны по отношению к типам, и, как правило, имеют более простой интерфейс. Отметим, что использование шаблонов в качестве механизма реализации обобщенного программирования не является решением, присущим исключительно программированию на языке C++. Шаблоны поддерживаются и в ряде других языков программирования (например, в языках Ada и Eiffel).

Определение шаблонной функции

Для того чтобы определить шаблонную функцию, нужно воспользоваться специальным предложением языка C++, предваряющим это определение. В простейшем случае такое предложение выглядит следующим образом:

```
template <class T>
```

Идентификатор `T` исполняет в этой конструкции роль параметра шаблона. Слово `class` указывает на основную область применения шаблонов — опре-

деление и использование параметризуемых типов классов, однако шаблонными могут быть объявлены и отдельные функции. Шаблон может иметь несколько параметров.

Вариант шаблонной реализации функции `BubbleSort()` представлен в листинге 9.2.

Листинг 9.2. Шаблонная функция сортировки простыми обменами

```
template <class T>
void BubbleSort( T *array, int n ) {

    for( int i = 1; i < n; i++ )
        for( int j = n-1; j>=i; j-- )
            if( array[ j ] < array[ j-1 ] ) {

                T copy = array[ j ];
                array[ j ] = array[ j-1 ];
                array[ j-1 ] = copy;
            }
}
```

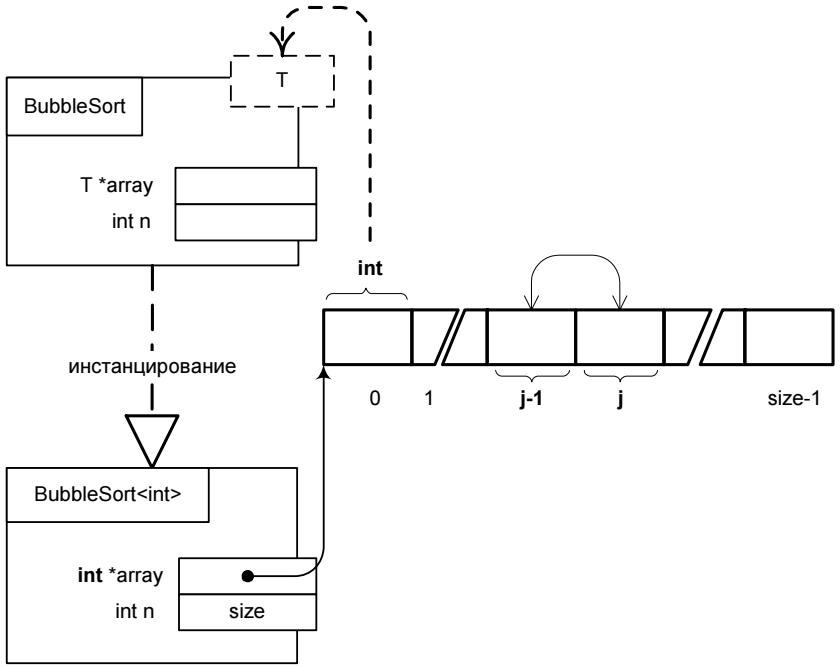


Рис. 9.2. Использование шаблонной функции сортировки

Эта версия незначительно отличается от специализированной версии для сортировки массива целых чисел (рис. 9.2). Кроме того, в ней не используется указатель на функцию, осуществляющую сравнение. Вместо этого пользователю предлагается перегрузить операцию отношения < для работы с данными нужного пользователю типа. Разумеется, для встроенных типов этого делать не нужно, поскольку семантика сравнения данных встроенных типов известна. Таким образом, для сортировки данных целого типа эта функция может быть использована точно так же, как и функция `BubbleSortInt()`:

```
void SortInt( int *a, int length ) {

    BubbleSort( a, length );

}
```

Использование шаблонной функции. Инстанцирование

Для того чтобы воспользоваться шаблонной функцией для произвольного пользовательского типа, следует определить подходящую семантику операции сравнения, например:

```
struct SportResult { // Представление результата
                     // спортивного соревнования
   int points; // Набранные очки
   int wins;   // Одержанные победы
};

int operator < ( SportResult& arg1, SportResult& arg2 ) {

    // Если набранные очки совпадают, то сравниваем победы
    if ( arg1.points == arg2.points ) return arg1.wins < arg2.wins;

    // Основная ветвь: сравниваем набранные очки
    return arg1.points < arg2.points;
}

void SortExample( SportResult *standings, int length ) {

    BubbleSort( standings, length );

}
```

Процесс генерирования определения функции на основе информации об аргументе шаблона называется инстанцированием шаблона. Компилятор выводит типы параметров шаблона на основе типов аргументов функции при вызове. После того как тип параметра шаблона станет известен, на основании

шаблонного определения генерируется специализированная функция. Версия шаблона для конкретного аргумента называется специализацией шаблона. В последнем примере такой специализацией является функция с интерфейсом `BubbleSort( SportResult*, int )`, код которой автоматически генерируется транслятором.

Таким образом, шаблоны обеспечивают возможность генерации кода на основе параметризованных определений (поэтому иногда шаблоны называют реализацией параметрического полиморфизма, т. е. полиморфизма времени компиляции). В этом кроется и недостаток шаблонных определений. Обращение к шаблонным функциям с разными типами аргументов приведет к появлению множества почти идентичных функций, код которых, разумеется, будет занимать место в памяти (чего не происходит при использовании обобщенных функций в стиле C — там работает единственный экземпляр). Риск расточительного расходования памяти является платой за те удобства, которые предоставляют шаблоны.

9.3. Шаблонные классы — основной инструмент параметрического полиморфизма

Применение шаблонов для написания функций, реализующих стандартные алгоритмы, естественным образом распространяется и на случай определений классов. Рассмотрим возможность такого определения на примере модернизации класса `QueueInt`, поддерживающего работу с данными разных типов.

Определение и использование шаблонного класса

Вариант определения шаблонного класса `Queue` представлен в листинге 9.3. Обратите внимание, что в данной версии те члены класса, которые в прежних версиях объявлялись закрытыми, теперь определены со спецификатором `protected`. Это пригодится нам в дальнейшем.

Листинг 9.3. Определение шаблонного класса `Queue`

```
/*
 * Queue.h
 *
 * Определение класса для работы с очередью
 * Шаблонная версия
 */
#ifndef _Queue_h
#define _Queue_h
```

```
// Типы исключений, генерируемых методами класса Queue
class BadSizeException {};
class PutFullException {};
class GetEmptyException {};

// Определение шаблонного класса для представления очереди
template <class T>
class Queue {        // Стандартная очередь:
                     // "первым пришел — первым обслужен"
protected:
    enum {
        EMPTY = -1,   // Признак пустоты очереди
        MAXSIZE = 100 // Предельный размер очереди
    };

    T *body;                // Массив для хранения данных
    int size;               // Предельный размер
                            // инициализированной очереди
    int length;             // Текущая длина очереди

    int last;               // Индекс последнего занесенного
                            // элемента очереди
    int next;               // Индекс извлекаемого
                            // элемента очереди

public:
    // Конструктор обеспечивает создание очереди заданного размера
    // NB! Конструктор может генерировать исключения
    //     BadSizeException и bad_alloc
    Queue( int actualSize = MAXSIZE )
        throw( BadSizeException, bad_alloc );

    // Занесение элемента в очередь
    // NB! Метод может генерировать исключение PutFullException
    void Enqueue( const T& ) throw ( PutFullException );

    // Извлечение элемента из очереди
    // NB! Метод может генерировать исключение GetEmptyException
    T Dequeue() throw ( GetEmptyException );

    // Печать текущего содержимого очереди
    void Print();
};

// Конструктор
template <class T>
```

```
Queue <T>::Queue( int customSize )
    throw( BadSizeException, bad_alloc ) {

    if( customSize < 1 || customSize > MAXSIZE )
        throw BadSizeException();

    body = new T[ customSize ]; // Может генерировать исключение
                                // bad_alloc

    size = customSize;

    last = EMPTY;
    next = 0;

    length = 0;
}

// Добавление элемента в очередь
template <class T>
void Queue<T>::Enqueue( const T& item ) {

    if( length == size ) throw PutFullException();

    last = ( last+1 ) % size;
    length++;

    body[ last ] = item;
}

// Извлечение элемента из очереди
template <class T>
T Queue<T>::Dequeue() {

    if( length == 0 ) throw GetEmptyException();

    T copy = body[ next ];

    next = ( next+1 ) % size;
    length--;

    return copy;
}

// Печать содержимого очереди
template <class T>
void Queue<T>::Print() {

    if( length == 0 ) {
        cout << "Queue is empty." << endl;
```

```
        return;
    }

    int count;
    int ix;
    for( ix = next % size, count = 1;
         count<=length;
         ix = ( ix+1 ) % size, count++ ) {

        cout << body[ ix ] << " ";
    }

    cout << endl;
}

#endif
```

В отличие от обычных классов, реализация методов шаблонного класса обязательно должна располагаться в том же файле, где находится определение класса. Вероятно, это связано с тем, что шаблонный тип является не совсем настоящим типом, а служит своего рода схемой, на основе которой компилятор генерирует конкретный класс и реализацию его методов. Поэтому определение шаблонного класса и его методов может считаться, по существу, одним неразделяемым объявлением.

Тип для правильного инстанцирования шаблона указывается при каждом использовании имени шаблонного типа, например (в случае создания очереди из целых чисел):

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

#include "Queue.h"

void main()
{
    int size;

    cout << "Enter queue size:";

    cin >> size;

    cout << "Creating queue: size=" << size << "... ";
    cout << endl;
    Queue<int> *queue;
    try {
        queue = new Queue<int>( size );
    }
```

```
    catch( BadSizeException ) {
       cout << "Incorrect size." << endl;
       exit( 1 );
    }
    catch( bad_alloc ) {
       cout << "Not enough memory." << endl;
       exit( 1 );
    }
    cout << "OK." << endl;
    // ...
}
```

Графически процесс инстанцирования шаблонного класса представлен на рис. 9.3.

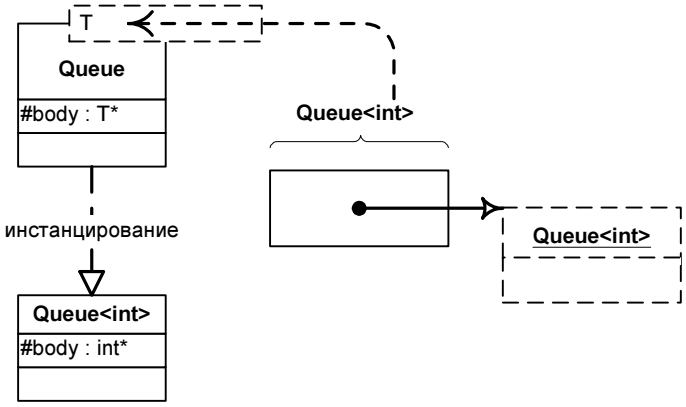


Рис. 9.3. Определение и инстанцирование шаблонного класса

Читателю предлагается самостоятельно изучить детали реализации и использования шаблонного класса `Queue`. В качестве отправной точки может быть использована программа на компакт-диске (см. проект, относящийся к листингу 9.3).

Параметры шаблонов

В качестве параметров шаблонов могут использоваться параметры трех видов:

- параметры-типы (как в рассмотренных выше примерах);
- параметры встроенных типов;
- параметры-шаблоны [Stroustrup, 2000, разд. B.13.3 русского перевода].

Параметр шаблона, не являющийся типом (например, параметр встроенного типа), является константой, т. е. его нельзя изменять в теле шаблона. Аргумент параметра целого типа тоже должен быть константой. Использование параметра целого типа иллюстрирует следующий фрагмент определения шаблонного класса `Queue` с параметризованным максимальным размером очереди:

```
template <class T, int SIZE_LIMIT>
class Queue
{
    enum { EMPTY = -1, MAXSIZE = SIZE_LIMIT };

    T   *body;
// ...
}
```

Для инстанцирования шаблона нужно указать все необходимые аргументы:

```
Queue<int, 200> queue2;
// ...
```

9.4. Контейнеры и итераторы

Одним из очевидных применений шаблонных классов является проектирование так называемых контейнерных (вмещающих) классов (типа `Queue`), обеспечивающих различные варианты хранения наборов объектов (стеки, очереди, hash-таблицы, векторы и т. д.). Основные операции над контейнером — это добавление и извлечение элемента из него. Некоторые контейнеры поддерживают последовательный просмотр хранящихся в них элементов. Страуструп выделяет два традиционных типа контейнеров: специализированные и стандартные. Рассмотрим последовательно оба этих типа.

Проектирование специализированных контейнеров

Специализированные контейнеры хорошо соответствуют известному принципу, который применительно к структурному программированию звучит так: "одна хорошо спроектированная функция решает одну задачу". При таком стиле конструирования контейнеров можно ясно выразить особенности их представления и цели их использования. При этом не рассматривается вопрос совместимости алгоритмов работы с контейнерами разных типов между собой. Таким образом, во главу угла ставится не унификация использования различных контейнеров, а удобство работы с конкретными контейнерами.

Определение специализированного контейнерного класса

Для примера сравним два вида специализированных контейнерных классов, воплощающих абстракции вектора и стека. Для контейнера-вектора добавление и извлечение элементов может быть реализовано через операцию индексирования `[]`, а для контейнера-стека традиционное управление данными, хранящимися в стеке, обеспечивается парой операций `push/pop`. Для соответствующих классов можно выбрать подходящее представление, не заботясь о других видах контейнеров:

```
template <class T>
class Vector {
public:
    explicit Vector( int size );
    T& operator[]( int index );
    // ...
};
```

```
template <class T>
class Stack {
public:
    explicit Stack( int size );
    void push( const T& item );
    T& pop();
    // ...
};
```

Таким образом, интерфейсы специализированных контейнеров различны.

Итерируемые контейнеры

Обычное использование многих видов контейнеров — это проход по контейнеру, или итерация. Набор действий, соответствующих итерации, может отличаться в зависимости от типа контейнера: для массива итерация предполагает увеличение (или уменьшение) на 1 индекса массива, а для списка — переход по значению указателя, связывающего элементы списка. Несмотря на это, пользователю контейнеров было бы желательно иметь возможность написать код для перебора элементов контейнера независимо от его типа. Таким образом, требуется не что иное, как определить общий интерфейс для осуществления итераций по контейнеру.

Решение может выглядеть следующим образом. Сначала определим интерфейс итератора (в C++ это абстрактный класс, содержащий только чисто виртуальные методы):

```
template <class T>
class Iterator {
```

```
public:
    virtual T* first() = 0;
            // Получить адрес первого элемента
    virtual T* next() = 0;
            // Получить адрес следующего элемента
};
```

Предположим, что определение шаблонного класса `Queue` у нас имеется (см. листинг 9.3).

Построим класс, обеспечивающий представление очереди, допускающей итерации, посредством наследования от шаблонного класса `Queue` и абстрактного шаблонного класса `Iterator` (листинг 9.4).

Листинг 9.4. Определение очереди, допускающей итерации

```
template <class T>
class IteratedQueue : public Queue<T>, public  Iterator<T> {
    int counter; // Счетчик итерируемых элементов
    int index;   // Индекс итерируемого элемента

public:
    IteratedQueue( int actualSize = MAXSIZE ) :
        Queue<T>( actualSize ) {

        counter = 0;
    }

    // Реализация получения адреса первого элемента
    T* first() {

        if( length > 0 ) {
            counter = 1;
            index = Queue<T>::next; // Индекс первого извлекаемого
                                    // элемента очереди
            return &body[ index ];  // Адрес этого элемента
        }

        return 0; // Возвращает 0, если нет элементов в очереди
    }

    // Реализация получения адреса очередного элемента
    T* next() {

        counter++;
        if( counter <= length ) {
            index = ( index+1 ) % size; // Индекс очередного элемента
```

```
            return &body[ index ];      // Адрес этого элемента
        }
        return 0; // Возвращает 0, если больше нет элементов
    }
};
```

Теперь ясен смысл определения членов класса `Queue` со спецификатором доступа `protected`. Это позволяет при реализации методов класса `IteratedQueue` иметь доступ к необходимым членам базового класса.

Использование итератора специализированного контейнера

Теперь мы можем написать программу для просмотра содержимого *любого* контейнера, поддерживающего интерфейс `Iterator`. Например, можно написать функцию, осуществляющую поиск элемента с заданным значением:

```
// Поиск элемента в итерируемом контейнере
template <class T>
const T* FindItem( Iterator<T> *itCont, // Адрес контейнера,
                                        // поддерживающего интерфейс
                                        // Iterator<T>
                   const T& item        // Элемент для поиска
                 ) {

    const T *ptr = itCont->first(); // Получить адрес первого
                                    // элемента контейнера

    while( ptr!= 0 ) { // Пока есть элементы...

        if( *ptr == item ) return ptr; // Если это искомый элемент -
                                       // завершаем работу

        ptr = itCont->next(); // Переходим к следующему элементу
    }

    return 0; // Возвращает 0, если не нашли
}
```

Данная функция корректно работает с контейнерами, содержащими данные встроенных типов без всяких дополнительных условий (за исключением, разумеется, того условия, что контейнер должен поддерживать интерфейс `Iterator`). Если контейнер содержит данные некоторого пользовательского типа, не поддерживаемого языком напрямую, для правильной работы обобщенного алгоритма `FindItem()` должна быть определена семантика сравнения

двух элементов пользовательского типа, т. е. должна быть определена перегруженная операция отношения ==.

В программе тестирования класса IteratedQueue, представленной на компакт-диске, в качестве примера обобщенного алгоритма реализована также обобщенная версия функции печати содержимого любого контейнера, поддерживающего интерфейс Iterator:

```
// Печать содержимого итерируемого контейнера
// NB! Для пользовательского типа должна быть определена
//     операция вывода <<
template <class T>
void GenericPrint( Iterator<T> *itCont ) {

    cout << "GenericPrint:" << endl;
    T *ptr = itCont->first();// Получить адрес первого
                            // элемента контейнера

    while( ptr!= 0 ) { // Пока есть элементы...
        cout << "\t" << *ptr << endl; // Печатаем значение очередного
                                      // элемента
        ptr = itCont->next();// Переходим к следующему элементу
    }
    cout << "End of GenericPrint" << endl;
}
```

К числу достоинств специализированных контейнеров относится возможность разработки контейнеров независимо друг от друга с последующим "вписыванием их в общие рамки" посредством конструирования классов, поддерживающих общий интерфейс итератора. При необходимости создания универсальных служб это достоинство оборачивается недостатком: контейнеры не имеют ничего общего, и объекты в контейнерах также не имеют ничего общего. Другой недостаток связан с затратами на вызов виртуальных функций типа first() и next(). В целом подобная модель отличается большой гибкостью и хорошо адаптируется к конкретным задачам.

Проектирование стандартных контейнеров

Стандартная библиотека шаблонов (Standard Template Library — STL) использует другой подход к проектированию контейнеров.

Разработчики STL исходили из предположения, что техника обобщенного программирования предполагает не только возможность определения и использования контейнерных типов (которые являются шаблонными классами или специализациями шаблонных классов), но и возможность работать с дан-

ными, хранящимися в контейнере, с использованием одних и тех же алгоритмов (сортировки содержимого контейнера, копирования содержимого одного контейнера в другой контейнер, поиска среди элементов контейнера, применения к элементам контейнера некоторой общей функции и т. д.).

Обобщенное представление стандартного контейнера и итератора контейнера

В основе механизмов, обеспечивающих подобные возможности, лежит обобщенное представление контейнера (рис. 9.4).

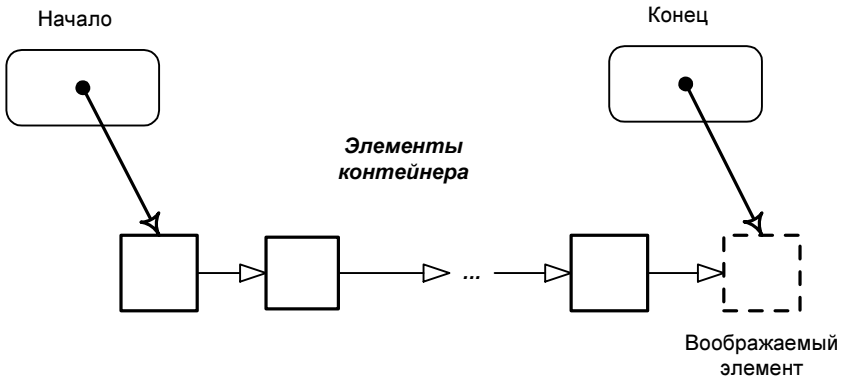


Рис. 9.4. Обобщенное представление контейнера

Началом контейнера в этом представлении считается первый элемент контейнера. Концом контейнера удобно считать несуществующий элемент, следующий за последним элементом (one-past-the-last-element). Шаг итерации состоит в переходе от одного элемента контейнера к другому.

Таким образом, для работы с элементами контейнера необходимо иметь возможность ссылаться на элементы контейнера. Для этого, как было показано в предыдущем разделе, необходимо определить итератор контейнера.

Итератор ссылается (указывает) на элемент и предоставляет операцию, заставляющую его (итератор) ссылаться на следующий элемент. Например, для массива C++ итератор образуют две операции: получение адреса элемента массива (`(&array[i]` или `array+i`) и переход к следующему элементу (инкремент).

Поскольку итератор абстрагирует понятие указателя на элемент последовательности, требуется реализация следующих операций над итератором:

- получение доступа к элементу контейнера через итератор (используются операции `*` и `->`);

- изменение итератора таким образом, чтобы он ссылался на следующий элемент (используется операция ++);
- сравнение итераторов (используются операции == и !=).

Если нужно обеспечить совместимость класса с контейнерами стандартной библиотеки, разработчик контейнера должен определить такие операции для итератора, которые поддерживаются контейнерным классом. Например, для того чтобы обеспечить возможность прямой итерации по контейнеру (см. рис. 9.4), разработчик должен определить два метода:

- метод для получения итератора, указывающего на первый элемент последовательности элементов контейнера (этот метод в STL-контейнерах называется `begin()`);
- метод для получения итератора, указывающего на элемент, следующий за последним (этот метод в STL-контейнерах называется `end()`).

Для итерации в обратном порядке нужно определить соответственно методы `rbegin()` (итератор, указывающий на первый элемент в обратной последовательности) и `rend()` (итератор, указывающий на элемент, следующий за последним, в обратной последовательности).

Определение контейнерного класса в стиле STL

Рассмотрим возможность переработки реализации класса `Queue` в стиле STL-контейнеров. В STL-версии класса `Queue` обеспечим поддержку двух прямых итераторов — итератора для чтения (`const_iterator`) и итератора для записи (`iterator`). Имена и организация классов-итераторов совпадают с именами и организацией аналогичных классов стандартной библиотеки, что должно облегчить изучение классов, реализованных в составе библиотеки STL.

В листинге 9.5 представлено определение класса `STLQueue` и пример программы тестирования возможностей обобщенной реализации очереди в стиле определений библиотеки STL. В частности, следует обратить внимание на использование стандартных алгоритмов, успешно работающих как с классами библиотеки STL, так и с только что спроектированным классом `STLQueue`. Отметим, что комментарии к данному тексту в основном предназначены для прояснения тех элементов решения, которые связаны с проектированием стандартных контейнеров. Элементы реализации, смысл которых совпадает с прежними реализациями, здесь не комментируются. При этом наиболее существенные изменения по сравнению с прежними версиями выделены полужирным начертанием.

Примечание

Несмотря на то, что текст, представленный в листинге 9.5, является, вероятно, самым длинным в книге, следует заметить, что класс `STLQueue` иллюстрирует

реализацию только основных элементов определения STL-контейнера и не может считаться полной законченной версией контейнерного класса. Для обеспечения полной совместимости класса с библиотекой STL определение класса `STLQueue` следует доработать. Читатель может самостоятельно изучить определения классов библиотеки STL и сделать необходимую модернизацию.

Листинг 9.5. Проектирование контейнера, совместимого с STL

```
/*
 * STLQueue.h
 *
 * Определение класса для работы с очередью
 * Шаблонная версия, проектируемая в стиле STL
 */
#ifndef _STLQueue_H
#define _STLQueue_H

#include <iostream>
using namespace std;

/*
 * Определения классов исключений в связи с организацией работы очереди
 */
class QueueException {
public:
    virtual const char* what() {
        return "Undefined queue exception.";
    }
};

class InitializationException : public QueueException {
public:
    const char* what() {
        return "Queue initialization exception.";
    }
};

class ProcessingException : public QueueException {
public:
    const char* what() {
        return "Queue processing exception.";
    }
};
```

```
class BadSizeException : public InitializationException {
public:
    const char* what() {
        return "Incorrect size error.";
    }
};

class PutFullException : public ProcessingException {
public:
    const char* what() {
        return "Put item error. Queue is full.";
    }
};

class GetEmptyException : public ProcessingException {
public:
    const char* what() {
        return "Get item error. Queue is empty.";
    }
};

/*
 * Определение контейнерного класса
 */
template <class T>
class STLQueue { // Контейнер-очередь:
                 // класс, проектируемый в стиле STL-контейнера
public:
    typedef T value_type; // Тип элементов контейнера

private:
    enum { EMPTY = -1, MAXSIZE = 100 };

    T   *body;
    int size;  // Увеличенный на единицу размер очереди

    int last;
    int next;

    int length;

public:
    STLQueue( int actualSize = MAXSIZE )
        throw( BadSizeException, bad_alloc );

    void Enqueue( const T& ) throw( PutFullException );
    T Dequeuet() throw ( GetEmptyException );
```

```
void Print();

/*
 * Объявления и определения в связи с итератором для чтения
 */

class const_iterator; // Объявление итератора для чтения
friend class const_iterator; // Предоставление итератору
                             // дружественного доступа к
                             // представлению класса

// Определение класса-итератора для чтения
class const_iterator {

    const value_type* ptr; // Указатель на элемент контейнера
    const STLQueue<T>* q;  // Указатель на контейнер
public:
    // Конструктор по умолчанию
    const_iterator() {}

    // Конструктор: инициализация итератора
    // значением адреса элемента
    const_iterator(
        const STLQueue<T>* _q, // Адрес контейнера
        const value_type* _ptr // Адрес элемента
                  ) : ptr( _ptr ),
                       q ( _q ) {}

    // Конструктор: инициализация итератора
    // значением другого итератора
    const_iterator(
        const const_iterator& _it
                  ) : ptr( _it.ptr ),
                       q( _it.q ) {}

    // Операция разыменования итератора
    const value_type& operator*() const {
         return *ptr;
    }

    // Операция доступа к элементу контейнера
    // через итератор
    const value_type* operator->() const {
         return ptr;
    }
```

```
    // Продвижение по контейнеру
    // (префиксная форма операции инкремента)
    const_iterator& operator++() {
        ++ptr; // Увеличить адрес очередного элемента

        if( ptr >= q->body + q->size ) {
             // Если вышли за пределы body, установить
             // prt на начало массива body
             // (см. схему организации хранения элементов очереди
             //  на рис. 3.1)
             ptr = q->body;
        }
        return *this;
    }

    // Операции сравнения итераторов
    bool operator==( const_iterator& it2 ) const {
        return ptr == it2.ptr;
    }

    bool operator!=( const_iterator& it2 ) const {
        return ( !( *this == it2 ) );
    }
};

// Получение итератора для чтения, ссылающегося на
// начало контейнера
// NB! Объявление функции константной позволяет отличить
//     ее от перегруженной версии iterator begin() (см. ниже)
const_iterator begin() const {

    if( length == 0 ) return const_iterator();

    return const_iterator( this, &body[ next ] );
}

// Получение итератора для чтения, ссылающегося на
// конец контейнера
// NB! Объявление функции константной позволяет отличить
//     ее от перегруженной версии iterator end() (см. ниже)
const_iterator end() const {

     if( length == 0 ) return const_iterator();

     return const_iterator( this, &body[ ( last+1 ) % size ] );
}
```

```
/*
 * Объявления и определения в связи с итератором для записи
 */

class iterator; // Объявление итератора для записи
friend class iterator; // Предоставление итератору
                       // дружественного доступа к
                       // представлению класса

// Определение класса-итератора для записи
class iterator : public const_iterator {

    value_type* ptr; // Указатель на элемент контейнера
    STLQueue<T>* q;  // Указатель на контейнер
public:
    // Конструктор по умолчанию
    iterator() {}

    // Конструктор: инициализация итератора
    // значением адреса элемента
    iterator(
        STLQueue<T>* _q, // Адрес контейнера
        value_type* _ptr // Адрес элемента
                 ) : ptr( _ptr ),
                      q ( _q ),
                      const_iterator( _q, _ptr ) {}

    // Конструктор: инициализация итератора
    // значением другого итератора
    iterator(
        const const_iterator& _it
                 ) : ptr( _it.ptr ),
                      q( _it.q ),
                      const_iterator( _it.q, _it.ptr ) {}

    // Операция разыменования итератора
    value_type& operator*() const {
         return *ptr;
    }

    // Операция доступа к элементу контейнера
    // через итератор
    value_type* operator->() const {
         return ptr;
    }
```

```
        // Продвижение по контейнеру
        // (префиксная форма операции инкремента)
        iterator& operator++() {
            ++ptr; // Увеличить адрес очередного элемента

            if( ptr >= q->body + q->size ) {
                 // Если вышли за пределы body, установить
                 // prt на начало массива body
                 // (см. схему организации хранения элементов очереди
                 //  на рис. 3.1)
                 ptr = q->body;
            }
            return *this;
        }

        // Операции сравнения итераторов
        bool operator==( iterator& it2 ) const {
            return ptr == it2.ptr;
        }

        bool operator!=( iterator& it2 ) const {
            return ( !( *this == it2 ) );
        }
    };

    // Получение итератора для записи, ссылающегося на
    // начало контейнера
    // NB! Эта функция не константная!
    //     Ср. с перегруженной версией const_iterator begin() (см. выше)
    iterator begin() {

        if( length == 0 ) return iterator();

        return iterator( this, &body[ next ] );
    }

    // Получение итератора для записи, ссылающегося на
    // конец контейнера
    // NB! Эта функция не константная!
    //     Ср. с перегруженной версией const_iterator end() (см. выше)
    iterator end() {

         if( length == 0 ) return iterator();

         return iterator( this, &body[ ( last+1 ) % size ] );
    }
};
```

```
/*
 * Реализация методов класса STLQueue
 */

template <class T>
STLQueue <T>::STLQueue( int customSize )
throw( BadSizeException, bad_alloc ) {

    if( customSize < 1 || customSize > MAXSIZE )
        throw BadSizeException();

    size = customSize+1; // Лишний (фиктивный) элемент необходим
                         // для организации итераций

    body = new T[ size ];

    last = EMPTY;
    next = 0;

    length = 0;
}

template <class T>
void STLQueue<T>::Enqueue( const T& item )
throw( PutFullException ) {

    if( length == size-1 ) throw PutFullException();

    last = ( last+1 ) % size;
    length++;

    body[ last ] = item;
}

template <class T>
T STLQueue <T>::Dequeue() throw ( GetEmptyException ) {

    if( length == 0 ) throw GetEmptyException();

    T copy = body[ next ];

    next = ( next+1 ) % size;
    length--;

    return copy;
}

template <class T>
void STLQueue <T>::Print() {
```

```
    if( length == 0 ) {
        cout << "Queue is empty" << endl;
        return;
    }

    int count;
    int ix;
    for( ix = next, count = 1;
         count <= length;
         ix = ( ix+1 ) % size, count++ ) {

        cout << body[ ix ] << " ";
    }

    cout << endl;
}
#endif

/*
 * STLAlg.h
 *
 * Примеры реализации обобщенных алгоритмов в стиле STL:
 *     1. Копирование контейнеров;
 *     2. Печать содержимого контейнера на консоль
 */
#ifndef _STLAlg_h
#define _STLAlg_h

// Копирование контейнеров:
// реализация, идентичная STL-алгоритму копирования
// (функция снабжена тестовой печатью для наблюдения за итерированием)
template <class In, class Out> void MyCopy(
        In from,    // Итератор, указывающий на начало контейнера-
                    // источника данных
        In too_far, // Итератор, указывающий на конец контейнера-
                    // источника данных
        Out to      // Итератор, указывающий на начало контейнера-
                    // приемника данных
        ) {
    cout << "MyCopy:" << endl;
    while( from != too_far ) {
        *to = *from;
        cout << "\t" << *from << " copied" << endl;
        ++to;
        ++from;
    }
```

```
    cout << "End of MyCopy" << endl;
}

// Печать содержимого контейнера
// (функция снабжена тестовой печатью)
template <class Cont> void PrintContents( Cont& c ) {
    cout << "PrintContents:" << endl;

    Cont::const_iterator from = c.begin();
    Cont::const_iterator too_far = c.end();
    int counter = 0;

    while( from != too_far ) {

        ++counter;
        cout << "\t" << counter << " : " << *from << endl;
        ++from;
    }
    if( counter == 0 ) cout << "\tEmpty" << endl;

    cout << "End of PrintContents" << endl;
}
#endif

/*
 * QueueAlgTest.cpp
 *
 * Программа, иллюстрирующая использование класса STLQueue,
 * алгоритмов из файла STLAlg.h
 * и стандартных средств STL
 */
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

#include "STLQueue.h"
#include "STLAlg.h"

// Вспомогательная функция печати целочисленного массива,
// используемого в ходе тестирования
void PrintArray( int *a, int size ) {
    cout << "Copied array: ";
    for( int i=0; i<size; i++ )
        cout << a[ i ] << " ";
    cout << endl;
}
```

```
// Программа тестирования
int main() {

    int array[ 3 ]; // Создаем целочисленный массив из трех чисел
                    // (не инициализирован)
    vector<int> vect( 4 ); // Создаем стандартный вектор из четырех
                           // элементов целого типа

    // Создаем очередь queue класса STLQueue, имеющую размер
    // по умолчанию, способную хранить целочисленные данные
    cout << "Creating queue: size = 101... ";
    STLQueue<int> *queue;
    try {
        queue = new STLQueue<int>( 101 );
    }
    catch( BadSizeException& e ) {
        cout << endl << e.what() << endl;
        cout << "OK." << endl;
    }
    catch( bad_alloc ) {
        cout << endl << "Memory allocation error" << endl;
        cout << "Fatal error. Application stopped." << endl;
        return -1;
    }

    // Создаем очередь queue1 класса STLQueue, имеющую размер 3,
    // способную хранить целочисленные данные
    cout << "Creating queue: size = 3... ";
    STLQueue<int> *queue1;
    try {
        queue1 = new STLQueue<int>( 3 );
        cout << "OK." << endl;
    }
    catch( bad_alloc ) {
        cout << endl << "Memory allocation error" << endl;
        cout << "Fatal error. Application stopped" << endl;
        return -1;
    }

    // Работаем с очередью queue1 (добавляем и удаляем элементы)
    // NB! Сопровождающими комментариями показывается
    //     текущее состояние очереди
    cout << "queue1.Enqueue( 100 )... ";
    queue1->Enqueue( 100 ); // 100
    cout << "OK." << endl;
```

```
cout << "queue1.Print(): ";
queue1->Print();
cout << "OK." << endl;

cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // Empty
cout << "OK." << endl;

cout << "queue1.Print(): ";
queue1->Print();
cout << "OK." << endl;

cout << "queue1.Enqueue( 200 )... ";
queue1->Enqueue( 200 ); // 200
cout << "OK." << endl;

cout << "queue1.Enqueue( 300 )... ";
queue1->Enqueue( 300 ); // 200 300
cout << "OK." << endl;

// Копируем содержимое очереди в массив array
// (используем функцию из STLAlg.h)
MyCopy( queue1->begin(), queue1->end(), &array[ 0 ] );
PrintArray( array, 2 ); // Контрольная печать содержимого array:
                        // должен содержать значения 200 300

// Заносим в queue1 еще один элемент
cout << "queue1.Enqueue( 400 )... ";
queue1->Enqueue( 400 ); // 200 300 400
cout << "OK." << endl;

// Печатаем содержимое контейнера queue1
// (используем функцию из STLAlg.h)
PrintContents( *queue1 ); // Должны быть напечатаны значения
                          // 200 300 400

// Контрольная печать встроенной функцией-членом класса STLQueue
// (содержательно результаты этой и предыдущей печати должны
//  совпадать)
cout << "queue1.Print(): ";
queue1->Print(); // 200 300 400
cout << "OK." << endl;

// Выполняем попытку занесения элемента в переполненную очередь
try {
    queue1->Enqueue( 500 ); // Генерируется PutFullException
}
```

```
catch( PutFullException& e ) {
    cout << e.what() << endl;
    cout << "OK." << endl;
}

// Извлекаем элементы из очереди queue1
cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 300 400
cout << "OK." << endl;

cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 400
cout << "OK." << endl;

cout << "queue1.Print(): ";
queue1->Print(); // 400
cout << "OK." << endl;

// Заносим элемент в очередь queue1
cout << "queue1.Enqueue( 600 )... ";
queue1->Enqueue( 600 ); // 400 600
cout << "OK." << endl;

cout << "queue1.Print(): ";
queue1->Print(); // 400 600
cout << "OK." << endl;

cout << "queue1.Enqueue( 600 )... ";
queue1->Enqueue( 600 ); // 400 600 600
cout << "OK." << endl;

cout << "queue1.Print(): ";
queue1->Print(); // 400 600 600
cout << "OK." << endl;

// Копируем содержимое очереди в массив array
// (используем функцию copy из библиотеки STL)
copy( queue1->begin(), queue1->end(), &array[ 0 ] );
PrintArray( array, 3 ); // Контрольная печать содержимого array:
                        // должен содержать значения 400 600 600

// Сравниваем содержимое очереди queue1 и массива array
// (используем функцию equal из библиотеки STL)
if ( equal( queue1->begin(), queue1->end(), &array[ 0 ] ) )
    cout << "Queue and array are now equal" << endl;
```

```
// Подсчитываем число вхождений элемента 600 в очередь queue1
// (используем функцию count из библиотеки STL)
int counted = count( queue1->begin(), queue1->end(), 600 );
cout << "Value 600 counted: " << counted << endl;

// Копируем содержимое очереди в вектор vect
// и сравниваем содержимое двух контейнеров
// (используем функции copy и equal из библиотеки STL)
copy( queue1->begin(), queue1->end(), vect.begin() );
if ( equal( queue1->begin(), queue1->end(), vect.begin() ) )
    cout << "Queue and vector are now equal" << endl;

// Изменяем содержимое вектора
vect[ 3 ] = 1000;

// Печатаем содержимое контейнера queue1
// (используем функцию из STLAlg.h)
PrintContents( vect ); // Должны быть напечатаны значения
                       // 400 600 600 1000

// Извлекаем элементы из очереди queue1
cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 600 600
cout << "OK." << endl;

cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 600
cout << "OK." << endl;

// Заносим элементы в очередь queue1
cout << "queue1.Enqueue( 800 )... ";
queue1->Enqueue( 800 ); // 600 800
cout << "OK." << endl;

cout << "queue1.Enqueue( 900 )... ";
queue1->Enqueue( 900 ); // 600 800 900
cout << "OK." << endl;

cout << "queue1.Print(): ";
queue1->Print(); // 600 800 900
cout << "OK." << endl;

// Осуществляем поиск элемента со значением 600 в очереди
// (используем функцию find библиотеки STL)
// NB! Элемент должен быть обнаружен
STLQueue<int>::iterator found; // Итератор для представления
                               // результата поиска
```

```
found = find( queue1->begin(), queue1->end(), 600 );
cout << "Value requested: 600 => Value found: " << *found << endl;

// Осуществляем поиск элемента со значением 800 в очереди
// (используем функцию find библиотеки STL)
// NB! Элемент должен быть обнаружен
found = find( queue1->begin(), queue1->end(), 800 );
cout << "Value requested: 800 => Value found: " << *found << endl;

// Осуществляем поиск элемента со значением 900 в очереди
// (используем функцию find библиотеки STL)
// NB! Элемент должен быть обнаружен
found = find( queue1->begin(), queue1->end(), 900 );
cout << "Value requested: 900 => Value found: " << *found << endl;

// Осуществляем поиск элемента со значением 1000 в очереди
// (используем функцию find библиотеки STL)
// NB! Ожидаемый результат: элемент не обнаружен
found = find( queue1->begin(), queue1->end(), 1000 );
if( found == queue1->end() )
    cout << "Value requested: 1000 => Not found: " << endl;

// Извлекаем элементы из очереди
cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 800 900
cout << "OK." << endl;

cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 900
cout << "OK." << endl;

cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // Empty
cout << "OK." << endl;

// Выполняем попытку извлечения элемента из пустой очереди
try {
    queue1->Dequeue(); // Генерируется GetEmptyException
}
catch( GetEmptyException& e ) {
    cout << e.what() << endl;
    cout << "OK." << endl;
}

// Печатаем содержимое очереди
// (убеждаемся, что очередь действительно пуста)
cout << "queue1.Print(): ";
```

```
    queue1->Print(); // Empty
    cout << "OK." << endl;

    // Разрушаем объект queue1
    cout << "Destroying queue... ";
    delete queue1;
    cout << "OK." << endl;

    // Успешное завершение тестирующей программы
    return 0;
}
```

Обратите внимание на некоторые изменения реализации членов класса `STLQueue` по сравнению с классом `Queue` в листинге 9.3. Во-первых, спецификатор `protected` вновь заменен спецификатором `private`. Во-вторых, из-за особенностей представления итераторов при размещении массива `body` в памяти приходится зарезервировать один "лишний" элемент. Если этого не сделать, то в том случае, когда очередь будет заполнена, итератор, возвращаемый функцией `begin()`, будет совпадать с итератором, возвращаемым функцией `end()`. Впрочем, эти изменения никак не затрагивают внешний интерфейс класса в той его части, которая совпадает с прежними реализациями.

Обобщенные функции для обработки стандартных контейнеров

Наряду с решением задачи разработки STL-контейнера, листинг 9.5 иллюстрирует также основные принципы написания обобщенных функций, способных обрабатывать итерируемые контейнеры (см. содержимое файла STLAlg.h). Примерами таких функций являются функция копирования контейнеров (которая практически идентична соответствующей функции стандартной библиотеки) и функция печати содержимого контейнера на консоль.

В качестве параметра шаблона при написании таких функций обычно выступают или типы контейнера (как в случае функции `PrintContents()`), или типы итераторов контейнера (как в случае функции `MyCopy()`). Условием возможности обработки контейнера такими функциями является поддержка контейнером необходимых итераторов и обеспечение реализации методов `begin()` и `end()`, возвращающих значение итератора.

Примерами использования функций из файла STLAlg.h служат отдельные фрагменты программы из файла QueueAlgTest.cpp (см. листинг 9.5). Для наглядности функции `PrintContents()` и `MyCopy()` снабжены контрольной печатью, которая, конечно, не нужна в библиотечных реализациях.

Стандартная библиотека шаблонов (STL)

В действительности, для написания программ, работающих с контейнерами в стиле STL, вообще нет необходимости в написании функции `MyCopy()`, хотя она, разумеется, интересна в качестве объекта изучения. Стандартная библиотека уже содержит подобную функцию, которая называется `copy()`. Использование этой функции также иллюстрирует тестирующая программа листинга 9.5.

Реализовав контейнерный класс в соответствии с соглашениями стандартной библиотеки (он имеет методы `begin()` и `end()`) и обеспечив семантику необходимых для обработки контейнера итераторов (так, контейнер `STLQueue` поддерживает итераторы `const_iterator` и `iterator`), мы имеем возможность использовать для работы с данным контейнером функции из стандартной библиотеки. Ниже представлен фрагмент кода, иллюстрирующего использование функций копирования и некоторых других функций из стандартной библиотеки:

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

#include "STLQueue.h"
//...

void main() {

    int array[ 3 ];
    STLQueue<int> queue( 3 );

    queue.Enqueue( 100 );
    queue.Enqueue( 200 );
    queue.Enqueue( 200 );

    // Скопировать содержимое очереди в массив
    // (используем функцию copy)
    copy( queue.begin(), queue.end(), &array[ 0 ] );

    // Сравнить содержимое очереди и массива
    // (используем функцию equal)
    if ( equal( queue.begin(), queue.end(), &array[ 0 ] ) ) {
        cout << "Both containers seem to be the same" << endl;
    }
```

```
    // Подсчитать число элементов, равных 200
    // (используем функцию count)
    int counted = count( queue.begin(), queue.end(), 200 );
    cout << "Value 200 counted: " << counted << endl;

    // Найти элемент, равный 200
    // (используем функцию find)
    STLQueue<int>::iterator found;
    found = find( queue.begin(), queue.end(), 200 );
    cout << "Value found: " << *found << endl;

    // ...
}
```

Подробную информацию о структуре и содержании стандартной библиотеки читатель может почерпнуть из различных источников (например, [Austern, 1999], [Stroustrup, 2000], [Eckel, Allison, 2003] и др.) В этом разделе мы ограничимся перечислением некоторых часто используемых контейнеров и алгоритмов стандартной библиотеки шаблонов (табл. 9.1). При изучении, освоении и правильном использовании этих средств студентам будут полезны изложенные выше проектные принципы, положенные в основу STL.

Таблица 9.1. *Контейнеры и алгоритмы стандартной библиотеки шаблонов (STL)*

Заголовочный файл	Классы и функции	Описание
Последовательные контейнеры		
<vector>	`vector`	Одномерный массив из элементов типа T с возможностью динамического расширения занимаемого пространства
<list>	`list`	Двусвязный список из элементов типа T
<deque>	`deque`	Двусторонняя очередь из элементов типа T
<queue>	`queue`	Очередь FIFO из элементов типа T
<stack>	`stack`	Стек (очередь LIFO) из элементов типа T
Ассоциативные контейнеры		
<map>	`map`	Ассоциативный массив (отображение) из элементов типа T
<set>	`set`	Набор (множество) из элементов типа T

Таблица 9.1 *(окончание)*

Заголовочный файл	Классы и функции	Описание
Алгоритмы		
<algorithm>	`for_each ()`	Выполнение задаваемой операции для каждого элемента последовательности
	`find()`	Поиск первого вхождения значения в последовательность
	`count()`	Подсчет числа вхождений элемента в последовательность
	`equal()`	Возвращает `true`, если последовательности содержат одинаковые элементы, иначе — `false`
	`copy()`	Копирование последовательности в прямом порядке
	`fill()`	Заполнение всех элементов заданным значением
	`remove()`	Удаление элементов с заданным значением
	`reverse()`	Изменение порядка следования элементов на обратный

9.5. Контейнеры, построенные на основе однокоренной иерархии классов (модели Java, C#)

Итак, в C++ использование контейнерных классов, или классов-коллекций, для хранения данных разных типов обеспечивается определением контейнерных классов как шаблонных (параметризуемых) типов. Такой способ определения контейнерных классов позволяет осуществить реализацию контейнера с возможностью проверки типов во время компиляции.

В языках, построенных с использованием однокоренной иерархии классов, обычно используется другая модель конструирования контейнеров. Контейнер определяется как объект, способный хранить объекты (чаще — ссылки на объекты) одного универсального типа (суперкласса `Object`). По существу, контейнер может хранить что угодно. Такая модель реализована, например, в библиотеке Java и в библиотеке классов .NET (следовательно, может использоваться при программировании на языке C#).

Добавление элемента в контейнер осуществляется очень просто, поскольку любой тип может быть преобразован к типу `Object`. Наряду с простотой использования, основное достоинство такого механизма заключается в том, что разные контейнеры пользуются одним и тем же представлением класса (т. е. нет избыточности кода, появляющейся при инстанцировании шаблонов).

Недостаток такого подхода связан с тем, что при добавлении объекта в контейнер теряется информация о статическом типе объекта, поэтому, когда вы извлекаете элемент из контейнера, выдается ссылка на `Object`, а не на действительный динамический тип объекта. Для получения требуемого типа приходится воспользоваться явным понижающим (downcast) преобразованием типа (рис. 9.5).

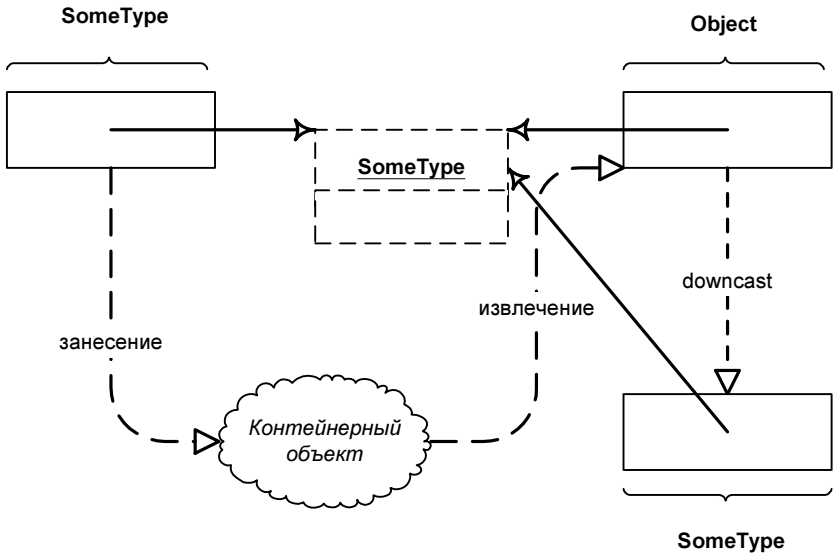


Рис. 9.5. Контейнеры, построенные на основе однокоренной иерархии классов

Как было рассмотрено выше *(см. гл. 5)*, понижающее преобразование не безопасно с точки зрения соответствия типов и нуждается в проверке времени выполнения. Поскольку и программы Java, и программы .NET исполняются в управляемой (контролируемой) среде исполнения, такая проверка осуществляется автоматически средой исполнения (соответственно, виртуальной машиной Java и средой CLR). В случае приведения к неверному типу генерируется исключение, которое может быть перехвачено клиентским кодом: в случае Java — это исключение `ClassCastException`, в случае .Net CLR — `System.InheritClassException`.

Наличие однокоренной иерархии классов делает возможным помещение в один контейнер объектов разных типов (поскольку все они суть `Object`). Здесь следует руководствоваться обычным житейским правилом: не все то, что можно сделать, нужно делать.

9.6. Иерархии шаблонных классов

С точки зрения организации взаимодействия объектов, модель обобщенного программирования предполагает возможность определения различных видов отношений между шаблонными классами и построения иерархий классов, по аналогии с иерархиями "обычных" абстрактных и конкретных классов. В связи с реализацией разнообразных контейнерных классов рассмотрим постановку задачи проектирования классов, обеспечивающих представление и обработку матриц (однородных таблиц данных). Ниже приводится пример целочисленной матрицы, состоящей из трех строк и четырех столбцов:

```
20   14    -7    9
16   -2    99    1
 4    0    -2    73
```

Графическим аналогом матрицы может служить обыкновенная таблица данных (рис. 9.6).

	0	1	2	3
0	20	14	-7	9
1	16	-2	99	1
2	4	0	-2	73

Рис. 9.6. Целочисленная матрица как таблица данных

Организация размещения данных в памяти

С точки зрения абстракций, поддерживаемых языком программирования, матрица — это двумерный массив. Исходя из необходимости обеспечения работы с матрицами произвольных размеров, рассмотрим задачу размещения двумерного массива в динамической памяти (heap). Основными альтернативами являются две возможности. Вариантом размещения, нацеленным на обеспечение наиболее эффективной обработки данных, является такое использование памяти, при котором матрица занимает непрерывный участок памяти (рис. 9.7). В соседних участках памяти при этом, как правило, распо-

лагаются строки матрицы, хотя отдельные языки могут быть ориентированы и на размещение по столбцам: например, такая модель лежит в основе представления двумерных массивов в языке FORTRAN.

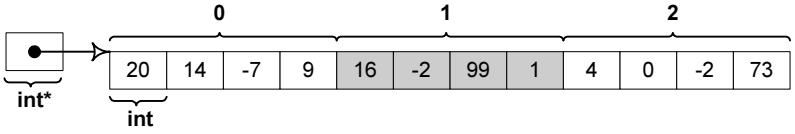


Рис. 9.7. Вариант размещения двумерного массива в непрерывном участке памяти (rectangular array)

При использовании подобной модели организации хранения массивов (rectangular arrays) требуется наличие в динамической памяти относительно больших участков динамической памяти (размер которых определяется произведением числа строк и числа столбцов матрицы).

Альтернативой является размещение отдельных строк матрицы в виде одномерных массивов, расположенных в разных участках динамической памяти, и управление этими массивами с использованием массива указателей, хранящих адреса одномерных массивов-строк (рис. 9.8).

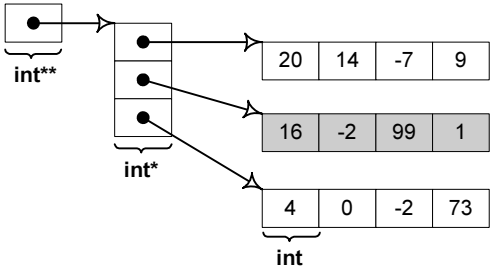


Рис. 9.8. Вариант размещения двумерного массива с использованием массива указателей (jagged array)

В этом случае время обращения к отдельному элементу матрицы может увеличиваться, однако при размещении матрицы не требуется обязательного наличия в динамической памяти непрерывных свободных сегментов большого размера.

Обычно язык программирования явно поддерживает ту или иную модель размещения двумерных массивов. Например, FORTRAN работает с массивами, размещаемыми в непрерывных "прямоугольных" участках памяти, а Java использует ссылочную модель (jagged array). Иногда такие массивы называют "ступенчатыми" [Gunnerson, 2000, перевод на русский язык: Е. Матвеев].

Существуют решения, когда язык программирования предоставляет специфический синтаксис для каждой из моделей размещения. Например, в языке C# многомерный массив является "прямоугольным" (rectangular array), если определяется следующим образом:

```
int [,] matrix = new int[ 3, 4 ]; // Определение массива
matrix[ 1, 2 ] = 99; // Обращение к отдельному элементу
```

Массив C# является "ступенчатым", если он определяется следующим образом:

```
int[][] matrix = new int[ 3 ][]; // Определение массива ссылок
for( int ixStr = 0; ixStr < 3; isStr++ ) { // Размещение отдельных строк
    matrix[ 0 ] = new int[ 4 ];
}
matrix[ 1 ][ 2 ] = 99; // Обращение к отдельному элементу
```

В общем случае, при реализации "ступенчатых" массивов размеры отдельных массивов-строк не обязательно совпадают.

Постановка задачи

Вернемся теперь к постановке задачи проектирования класса C++. Было бы неплохо предоставить программисту удобный механизм, который, с одной стороны, позволял бы при создании матричного объекта указать необходимый способ размещения содержимого матрицы в памяти, а с другой стороны, обеспечивал при работе с матричными объектами единый интерфейс, не зависящий от способа размещения. При таком подходе необходимость устанавливать зависимость кода от способа размещения данных в памяти наступает только в момент создания объекта.

Разумеется, не нужно напоминать о том, что матричный объект должен быть способен хранить и обрабатывать данные разных типов. Таким образом, проектируемые классы, очевидно, должны быть обобщенными.

Разработка классов*

Классы, обеспечивающие решение поставленной задачи, разделены на две группы. Первая группа классов представляет абстракции двумерных контейнеров в связи с операциями размещения матричных объектов в памяти и обеспечением доступа к этой памяти. При этом процесс создания матричного объекта предполагает не только размещение в памяти самого объекта, но и распределение динамической памяти для хранения элементов контейнера по

* Благодарю Алексея Рытенкова за помощь в подготовке и тестировании этого примера.

прямоугольному или ступенчатому "сценарию". На вершине иерархии — абстрактный класс `Base2DCont`, определяющий общий интерфейс, который должны поддерживать оба вида контейнеров (и прямоугольный, и ступенчатый).

```
template <class T>
class Base2DCont {
public:
    virtual ~Base2DCont<T>(){} // Виртуальный деструктор

    // Объявление метода, обеспечивающего создание матрицы с
    // использованием того или иного метода размещения в памяти
    // ("виртуальный конструктор")
    virtual Base2DCont<T>* create( const int sizeRows,
                                   const int sizeCols ) const = 0;

    // Объявление метода для извлечения элемента матрицы
    virtual T& getEl( const int ixRow, const int ixCol ) const = 0;

    // Объявление метода для занесения элемента

    virtual void setEl( const int ixRow,const int ixCol,const T& el )=0;

    // Объявление метода для получения значения числа строк матрицы
    virtual int getSizeRows() const = 0;

    // Объявление метода для получения значения числа столбцов матрицы
    virtual int getSizeCols() const = 0;
};
```

Обратите внимание на функцию `create()`, исполняющую роль "виртуального конструктора". Наличие этой функции позволяет при создании матричного объекта абстрагировать от того, каким образом распределяется и используется память, в которой хранятся элементы матрицы.

Примечание

Для упрощения текстов программ и уменьшения объема рассматриваемого кода в этом разделе мы абстрагируемся от обработки ошибок, которые могут возникнуть при некорректной работе с матричными объектами, и просто используем "заглушки" в виде проверок утверждений. Читатель может самостоятельно дополнить классы необходимыми определениями, позволяющими реализовать полноценную модель обработки ошибок.

Классы `Rect2DCont` и `Jagg2DCont` реализуют абстрактные методы класса `Base2DCont`, обеспечивая управление памятью с выбранной стратегией размещения элементов:

```
#include <assert.h>

// Прямоугольный 2D-контейнер
template <class T>
class Rect2DCont : public Base2DCont<T> {
public:
    // Конструктор создания прямоугольного 2D-контейнера
    Rect2DCont( const int sizeRows = 2, const int sizeCols = 2 ) {
        this->sizeRows = sizeRows;
        this->sizeCols = sizeCols;
        body = new T[ sizeRows*sizeCols ];
    }

    // Деструктор
    virtual ~Rect2DCont<T>() {
        delete [] body;
        body = 0;
    }

    // Функция создания объекта прямоугольного 2D-контейнера,
    // формирующая указатель на Base2DCont
    virtual Base2DCont<T>* create( const int sizeRows,
                                   const int sizeCols ) const {
        return new Rect2DCont( sizeRows, sizeCols );
    }

    // Реализация функции извлечения элемента
    virtual T& getEl( const int ixRow, const int ixCol ) const {

        assert( ( ixRow>=0 )&&( ixCol>=0 ) &&
                ( ixRow<=sizeRows )&&( ixCol<=sizeCols ) );
        return body[ ixRow + sizeRows*ixCol ];
    }

    // Реализация функции занесения элемента
    virtual void setEl( const int ixRow, const int ixCol, const T& el ) {
        assert( ( ixRow>=0 )&&( ixCol>=0 ) &&
                ( ixRow<=sizeRows )&&( ixCol<=sizeCols ) );
        body[ ixRow + sizeRows*ixCol ] = el;
    }

    // Получение размеров матрицы
    virtual int getSizeRows() const { return sizeRows; }
    virtual int getSizeCols() const { return sizeCols; }
```

```
private:
    T* body; // Адрес начала массива элементов, хранящихся в непрерывном
             // участке памяти ("прямоугольного" массива)
    int sizeRows; // Число строк матрицы
    int sizeCols; // Число столбцов матрицы
};

// Ступенчатый 2D-контейнер
template <class T>
class Jagg2DCont : public Base2DCont<T> {
public:
    // Конструктор создания ступенчатого 2D-контейнера
    Jagg2DCont( const int sizeRows = 2, const int sizeCols = 2 ) {
        this->sizeRows = sizeRows;
        this->sizeCols = sizeCols;
        body = new T*[ sizeRows ];
        for( int i = 0; i < sizeRows; i++ )
           body[ i ] = new T[ sizeCols ];
    }

    // Деструктор
    virtual ~Jagg2DCont<T>() {
        for( int i = 0; i < sizeRows; i++ )
            delete[] body[ i ];
         delete[] body;
         body = 0;
    }

    // Функция создания объекта ступенчатого 2D-контейнера,
    // формирующая указатель на Base2DCont
    virtual Base2DCont<T>* create( const int sizeRows,
                                   const int sizeCols ) const {
        return new Jagg2DCont( sizeRows, sizeCols );
    }

    // Реализация функции извлечения элемента
    virtual T& getEl( const int ixRow, const int ixCol ) const {
        assert( ( ixRow >=0 )&&( ixCol>=0 ) &&
                ( ixRow<=sizeRows )&&( ixCol<=sizeCols ) );
        return body[ ixRow ][ ixCol ];
    }

    // Реализация функции занесения элемента
    virtual void setEl( const int ixRow, const int ixCol, const T& el ) {
        assert( ( ixRow>=0 )&&( ixCol>=0 ) &&
                ( ixRow<=sizeRows )&&( ixCol<=sizeCols ) );
```

```
        body[ ixRow ][ ixCol ] = el;
    }

    // Получение размеров матрицы
    virtual int getSizeRows() const { return sizeRows; }
    virtual int getSizeCols() const { return sizeCols; }
private:
    T** body; // Адрес начала "ступенчатого" массива
    int sizeRows; // Число строк матрицы
    int sizeCols; // Число столбцов матрицы
};
```

Спроектированные классы полностью инкапсулируют функциональность в связи с управлением памятью, занимаемой элементами массива. Таким образом, для создания матричного объекта нужно знать, какой вид контейнера необходим для хранения элементов матрицы. В результате мы приходим к структуре определений классов, представленной на рис. 9.9.

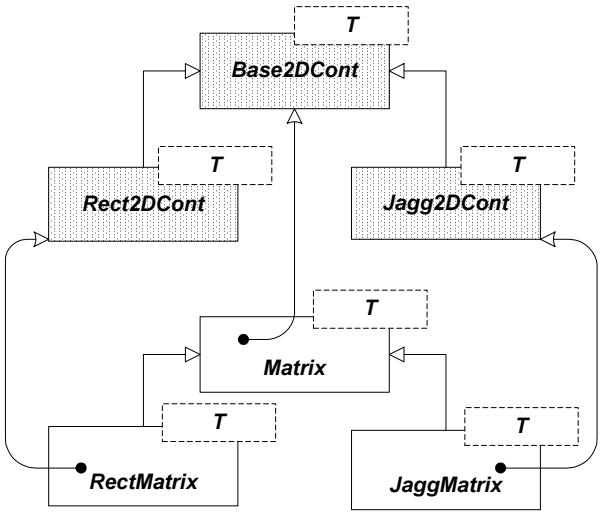


Рис. 9.9. Классы в связи с представлением прямоугольных и ступенчатых матриц

Основная часть функций, необходимых для работы с матрицами, реализуется в классе `Matrix`. Это — центральный класс иерархии. Объект класса `Matrix` хранит указатель на контейнер `Base2DCont`, который может быть связан как с объектом класса `Rect2DCont`, так и с объектом класса `Jagg2DCont`. Соответствующую привязку указателя должны выполнять конструкторы специализированных версий классов `RectMatrix` и `JaggMatrix`. Таким образом, мы приходим к определениям, представленным в листинге 9.6.

Листинг 9.6. Классы в связи с поддержкой матричных вычислений

```
/*
 * Файл Base2DCont.h
 */
#ifndef _Base2DCont_h_
#define _Base2DCont_h_

// Определение класса Base2DCont...

#endif

/*
 * Файл Rect2DCont.h
 */
#ifndef _Rect2DCont_h_
#define _Base2DCont_h_

#include "Base2DCont.h"
// Определение класса Rect2DCont...

#endif

/*
 *Файл Jagg2DCont.h
 */
#ifndef _Jagg2DCont_h_
#define _Jagg2DCont_h_

#include "Base2DCont.h"
// Определение класса Jagg2DCont...

#endif

/*
 *Файл Matrix.h
 */
#ifndef _Matrix_h_
#define _Matrix_h_

#include "Base2DCont.h"
#include "Jagg2DCont.h"
#include "Rect2DCont.h"

#include <assert.h>

// Матрица с возможностью выбора распределителя памяти
template <class T>
```

```
class Matrix {
public:
    // Конструктор (принимает адрес 2D-контейнера в качестве аргумента):
    //     вызывает виртуальную функцию create()
    Matrix( Base2DCont<T>* pCont );
    // Конструктор копирования:
    //     1. вызывает виртуальную функцию create()
    //     2. копирует матрицу
    Matrix( const Matrix<T>& matr );
    // Деструктор (обеспечивает удаление объекта 2D-контейнера)
    ~Matrix();

    // Основные операции над матрицами
    Matrix& operator= ( const Matrix& matr );
    Matrix& operator+= ( const Matrix& matr );
    Matrix  operator+ ( const Matrix& matr ) const;

    // Другие операции (здесь не реализованы, см. упр. 12)...
    /*
    Matrix& operator-= ( const Matrix& matr );
    Matrix  operator- ( const Matrix& matr ) const;
    Matrix<T>& operator*= ( T val );
    Matrix<T>  operator* ( T val ) const;
    Matrix<T>& operator/= ( T val );
    Matrix<T>  operator/ ( T val ) const;
    Matrix<T>  operator* ( const Matrix<T>& matr ) const;
    // ...
    */

    // Функции извлечения и занесения элементов
    T& getEl( const int ixRow, const int ixCol ) const;
    void setEl( const int ixRow, const int ixCol, const T& el );

    // Получение размеров матрицы
    int getSizeRows() const;
    int getSizeCols() const;

    // Получение указателя на 2D-контейнер
    Base2DCont<T>* getCont() const;

  private:
    Base2DCont<T>* pCont; // Указатель на 2D-контейнер
                          // (распределитель памяти)
};
```

```
// Реализация функций-членов класса Matrix
template <class T>
Matrix<T>::Matrix( Base2DCont<T>* pCont ) {
    this->pCont =
        pCont->create( pCont->getSizeRows(), pCont->getSizeCols() );
}

template <class T>
Matrix<T>::Matrix( const Matrix<T>& matr ) {
    pCont = matr.pCont->create( matr.getSizeRows(), matr.getSizeCols() );
    for( int i = 0; i < getSizeRows(); i++ ) {
        for( int j = 0; j < getSizeCols(); j++ ) {
            setEl( i, j, matr.getEl( i,j ) );
        }
    }
}

template <class T>
Matrix<T>::~Matrix() {
    delete pCont;
}

template <class T>
Matrix<T>& Matrix<T>::operator=( const Matrix<T>& matr ) {
    if( pCont ) delete pCont;
    pCont =
       matr.pCont->create( matr.getSizeRows(), matr.getSizeCols() );

     for( int i = 0; i < getSizeRows(); i++ ) {
        for( int j = 0; j < getSizeCols(); j++ ) {
            setEl( i, j, matr.getEl( i,j ) );
        }
    }

    return *this;
}

template <class T>
Matrix<T>& Matrix<T>::operator+= ( const Matrix<T>& matr ) {
    assert( ( getSizeRows() == matr.getSizeRows() ) &&
            ( getSizeCols() == matr.getSizeCols() ) );
    for( int i = 0; i < getSizeRows(); i++ ) {
        for( int j = 0; j < getSizeCols(); j++ ) {
            setEl( i, j, getEl( i,j ) + matr.getEl( i,j ) );
        }
    }
```

```
    return *this;
}

template <class T>
Matrix<T> Matrix<T>::operator+( const Matrix<T>& matr ) const {
    Matrix<T> res( *this );
    res += matr;
    return res;
}

template <class T>
T& Matrix<T>::getEl( const int ixRow, const int ixCol ) const {
    return pCont->getEl( ixRow, ixCol );
}

template <class T>
void Matrix<T>::setEl( const int ixRow, const int ixCol, const T& el ) {
    pCont->setEl( ixRow, ixCol, el );
}

template <class T>
int Matrix<T>::getSizeRows()const {
    return pCont->getSizeRows();
}

template <class T>
int Matrix<T>::getSizeCols()const {
    return pCont->getSizeCols();
}

template <class T>
Base2DCont<T>* Matrix<T>::getCont() const {
    return pCont;
}

// Определение специализированных версий матричных классов

// Прямоугольная матрица
template <class T>
class RectMatrix : public Matrix<T> {
public:
    // Требуется реализовать только правильную инициализацию
    // подобъекта Matrix
    RectMatrix( int sizeRows = 2, int sizeCols = 2 ) :
       Matrix<T>( new Rect2DCont<T>( sizeRows, sizeCols ) ) {}

    RectMatrix( const Matrix<T>& matr ) : Matrix<T>( matr ) {}
};
```

```
// Ступенчатая матрица
template <class T>
class JaggMatrix : public Matrix<T> {
public:
    // Требуется реализовать только правильную инициализацию
    // подобъекта Matrix
    JaggMatrix( int sizeRows = 2, int sizeCols = 2 ) :
        Matrix<T>( new Jagg2DCont<T>( sizeRows, sizeCols ) ) {}

    JaggMatrix( const Matrix<T>& matr ) : Matrix<T>( matr ) {}
};
#endif
```

В классе `Matrix` мы реализовали только часть полезной функциональности в связи с обработкой матриц, остальные операции и функции оставляем читателю для самостоятельной работы. На компакт-диске содержится небольшая программа тестирования спроектированных классов. В книге приведем фрагмент тестовой функции для работы с прямоугольными матрицами, содержащими целочисленные данные:

```
#include "Matrix.h"

#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

// Печать матрицы на консоль
template <class T>
void print( const Matrix<T> &mat ) {
    for( int i = 0; i < mat.getSizeRows(); i++ ) {
        for( int j = 0; j < mat.getSizeCols(); j++ ) {
            cout << mat.getEl( i, j ) << " ";
        }
        cout << endl;
    }
}

// Заполнение матрицы начальными значениями случайным образом
void randomIntMatrix( Matrix<int> &mat ) {
    for( int i = 0; i < mat.getSizeRows(); i++ ) {
        for( int j = 0; j < mat.getSizeCols(); j++ ) {
            mat.setEl( i, j, rand() % 10 );
        }
    }
}
```

```
// Тестовая функция
void RectMatriceTest() {
    srand( (unsigned) time( NULL ) );

    RectMatrix<int> rectMat1( 2, 3 );
    randomIntMatrix( rectMat1 );
    cout << "rectMat1:" << endl;
    print( rectMat1 );

    RectMatrix<int> rectMat2( 2, 3 );
    randomIntMatrix( rectMat2 );
    cout << "rectMat2:" << endl;
    print( rectMat2 );

    RectMatrix<int> rectMat3( 2, 3 );
    rectMat3 = rectMat1 + rectMat2;
    cout << "rectMat3 = rectMat1+rectMat2:" << endl;
    print( rectMat3 );
}
```

Графически процесс создания объекта класса `RectMatrix` иллюстрируется на рис. 9.10.

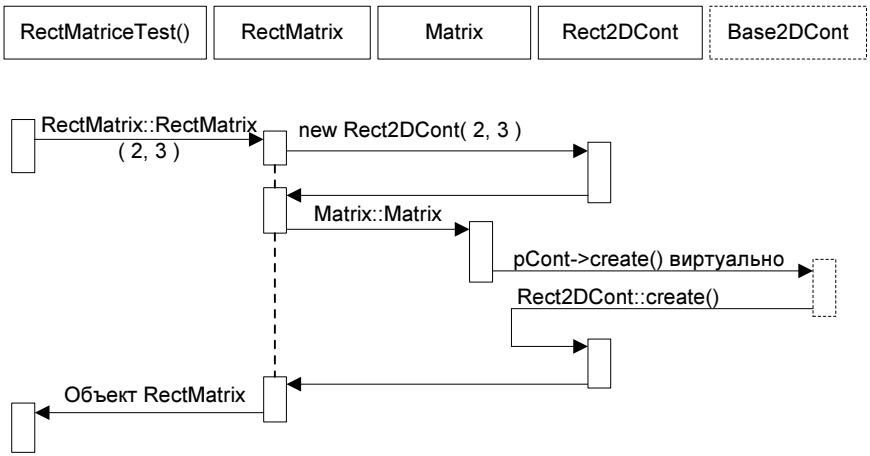


Рис. 9.10. Процесс создания матричного объекта

Обратите внимание, как виртуальный вызов функции `create()`, выполняемый в процессе создания подобъекта класса `Matrix`, обеспечивает связывание с переменной `pCont` объекта класса `Rect2DCont`. В конечном счете это и обеспечивает привязку к матричному объекту требуемого распределителя. На уровне абстракции клиентской программы все внутренние механизмы в связи с

распределением, использованием и освобождением памяти скрыты, чего, собственно, мы и стремились достичь.

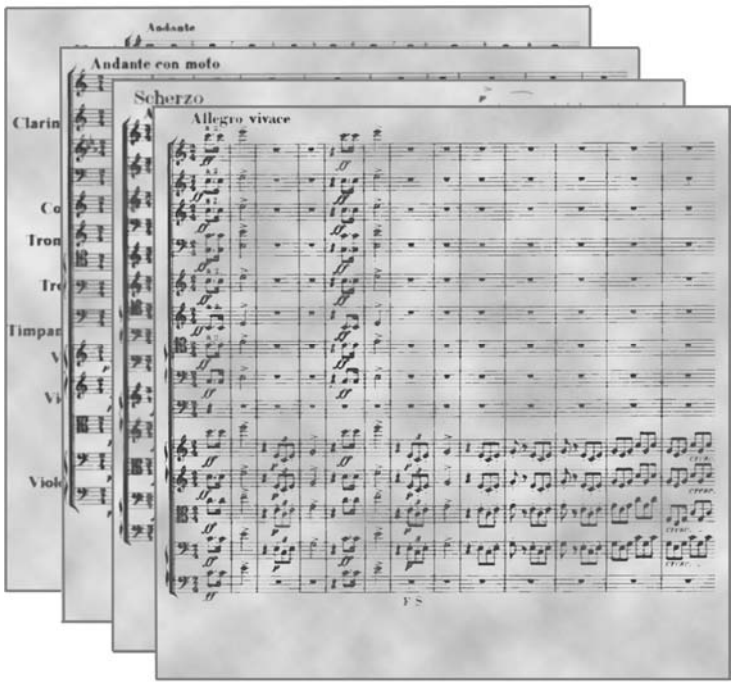
9.7. Вопросы и упражнения

1. Сформулируйте преимущества и недостатки моделей обобщенного структурного императивного программирования с использованием указателей на `void` (модель C) и с использованием шаблонных функций (модель C++).
2. Напишите реализацию алгоритма двоичного поиска на языке C. Напишите тестирующую программу для наборов данных различных типов. Сравните работу вашей функции с функцией `bsearch()` из стандартной библиотеки C. Напишите реализацию того же алгоритма с помощью шаблонной функции. Протестируйте вашу функцию для тех же наборов данных, что и в предыдущей программе. Сравните размер получившихся исполняемых кодов двух программ.
3. Объясните смысл понятия "параметрический полиморфизм" и его отличие от полиморфизма в связи с наследованием. Могут ли эти две формы полиморфизма использоваться в рамках одной и той же иерархии (шаблонных) классов?
4. Почему использование шаблонных функций и классов может приводить к увеличению кода программы? Можно ли с этим бороться?
5. Реализуйте специализированный контейнер, воплощающий абстракцию контролируемого массива, т. е. такого массива, при работе с которым обеспечивается контроль выхода за границы массива. Каким образом лучше всего информировать программиста о возникновении ошибочной ситуации?
6. Разработайте контейнер-контролируемый массив в стиле стандартной библиотеки C++. Сравните возможности Вашей реализации и возможности, обеспечиваемые шаблонным классом `vector` из STL. В чем Вы видите достоинства и недостатки Вашей реализации и реализации стандартной библиотеки?
7. Напишите реализации обобщенных алгоритмов, эквивалентных функциям-алгоритмам STL: `find()`, `count()`, `equal()`, `fill()`, `remove()`, `reverse()` (см. [Stroustrup, 2000, гл. 18]). Протестируйте эти функции на примере контейнера из предыдущего упражнения и на примерах контейнеров STL. Какие функции-алгоритмы, не представленные в STL, Вы включили бы в стандартную библиотеку и почему?
8. Для некоторой иерархии классов напишите реализацию контейнера C++, построенного по принципу контейнера на базе однокорневой иерархии классов. Почему в стандартной библиотеке C++ нет подобных контейнеров?

9. С использованием STL напишите программу, которая читает пары (key, value) (ключ, значение) и выводит сумму всех `value`, соответствующих каждому `key`. Укажите, какие требования должны предъявляться к типам `key` и `value`. Решите ту же задачу, не используя средства STL.
10. Предположим, что имеется набор функций одинаковой сигнатуры, выполняющих действия над общими данными. Функции из набора вызываются последовательно, т. е. результатом работы одной функции является формирование состояния данных, которое становится исходным состоянием для другой функции. Напишите программный модуль, обеспечивающий обобщенную реализацию подобного конвейера для разных наборов функций, работающих, возможно, с разными данными.
11. Реализуйте модификацию программы из предыдущего упражнения в предположении, что функции конвейера не имеют прямого доступа к данным, а используют некоторый интерфейсный класс.
12. Доработайте определение класса `Matrix` из *разд. 9.6*. Проанализируйте получившееся решение с точки зрения совместимости с библиотекой STL.

ЧАСТЬ IV

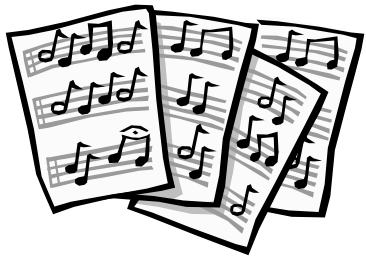
Развитие моделей



Глава 10. Организация вычислительного процесса в управляемых средах

Глава 11. Развитие объектно-ориентированной модели управления типами

Глава 12. Введение в компонентное программирование



ГЛАВА 10

Организация вычислительного процесса в управляемых средах

Развитие объектно-ориентированных языков, различия оснований проектирования разных языков обуславливают необходимость рассмотрения некоторых моделей, концепций и механизмов проектирования, реализация которых либо существенно отличается от реализации C++, либо обеспечивает поддержку на уровне языка тех средств, которые не имеют синтаксического выражения в языке C++. В данной главе рассматриваются некоторые из таких средств, а также приводится ряд дополнительных сведений, необходимых для понимания излагаемого материала.

10.1. Управляемый и неуправляемый код

No silver bullet.
Фред Брукс[1]

В начальном курсе программирования при рассмотрении этапов обработки компьютерной программы студенты знакомятся с двумя типами языков — компилируемыми и интерпретируемыми. Напомним, что компилятор — это разновидность транслятора, который преобразует программу целиком в исполняемый процессорный код (native code). Большинство универсальных языков прикладного программирования являются компилируемыми из-за соображений эффективности.

---

[1] Brooks F.P. No Silver Bullet — essence and accidents of software engineering. Статья опубликована в нескольких источниках, самым доступным в настоящее время, вероятно, является книга [Brooks, 1995], изданная в России в 2000 году.

Интерпретатор — это программа, которая транслирует и исполняет заданную программу пошаговым образом. Ряд языков программирования содержат конструкции, позволяющие изменять существующие подпрограммы или создавать новые динамически, в процессе выполнения (например, Lisp) [Одинцов, 2002]. Для таких языков интерпретация обычно предпочтительнее компиляции. Другой случай применения интерпретаторов связан с кроссплатформенными решениями (Basic, Tcl, Perl, JavaScript). Интерпретатор, по существу, является разновидностью исполнительной системы компьютерной программы, обеспечивающей переносимость кода на уровне исходных текстов. Многие интерпретируемые системы страдают существенно более низкой эффективностью по сравнению с компилируемыми языками.

Рост размеров и сложности программ, увеличение доли повторно используемого (библиотечного) кода, развитие Internet-программирования и распределенного программирования обусловили необходимость создания систем программирования, ориентированных на создание мобильного безопасного устойчивого кода. При этом под мобильностью кода понимается возможность выполнения одной и той же программы на разных вычислительных платформах. Под безопасностью кода понимается встроенная в язык защита от несанкционированного доступа к ресурсам компьютера, на котором исполняется приложение (в особенности, когда речь идет о сетевых приложениях). Устойчивость кода предполагает возможность обнаружения ошибок на возможно более ранних стадиях проектирования и исключение определенных классов ошибок (например, ошибок управления памятью и необработанных исключительных ситуаций). Обеспечение устойчивости кода предполагает максимально возможный контроль корректности операций, совершаемых в процессе исполнения программы, в частности контроль отсутствия ошибок при преобразовании типов и инициализации объектов, обнаружение переполнения в ходе арифметических вычислений и т. д.

Основной механизм, позволяющий обеспечить получение кода, обладающего такими свойствами, основывается на реализации специальной исполнительной системы, работающей не с кодом процессора, а с некоторым промежуточным кодом. Именно промежуточный код и является выходом компилятора при трансляции исходного текста программы.

Код, разрабатываемый на языке, ориентированном на исполнение в контролирующей среде, сейчас принято называть управляемым, или контролируемым, кодом (managed code). Байт-код Java является примером управляемого кода. В этой терминологии стандартный C/C++ генерирует неуправляемый код (unmanaged code), т. е. код, ориентированный непосредственно на процессор.

Управляемый код в Java

Ярким примером успешного применения в компьютерной индустрии подобной организации обработки программы является технология Java. В основе решения проблем построения мобильного, безопасного и устойчивого программного обеспечения с помощью Java лежит трансляция исходного кода в промежуточное представление (байт-код) и его выполнение специальной исполнительной системой, называемой виртуальной машиной Java (Java Virtual Machine, JVM). В классической версии JVM является интерпретатором байт-кода. В настоящее время в рамках платформы Java 2 построены реализации JVM с использованием технологии компиляции "на лету", или JIT-компиляции (Just-in-time compilation). JIT-компиляция — это преобразование в код процессора отдельных частей байт-кода (обычно соответствующих отдельным методам классов) по мере обращения к ним. JIT-компиляция позволяет существенно снизить издержки эффективности, присущие чистой интерпретации, поскольку однажды скомпилированный код уже не нуждается в повторной компиляции (рис. 10.1).

Поскольку программа Java пишется, по существу, для специальной модели вычислительной машины (этой моделью и является JVM), это означает, что язык обеспечивает защиту основной системы от случайных или преднамеренных вредоносных воздействий посредством ограничения Java-программы средой ее выполнения — такую модель иногда называют "программированием в песочнице" (sandbox programming). В процессе выполнения Java-программы JVM осуществляет множество проверок, гарантирующих безопасность и устойчивость кода (среди них некорректные динамические преобразования типов, ошибочная динамическая инициализация, некорректное значение индекса элемента массива, необработанные исключительные состояния, попытка сетевого приложения (апплета) обращаться к файлам на диске и т. д.) В листинге 10.1 приведен пример простой Java-программы, суммирующей значения целочисленных аргументов командной строки.

Листинг 10.1. Суммирование аргументов командной строки (пример на Java)

```
// Файл SumArgs.java

// Класс-"сумматор"
class ASum {
    int theSum; // Текущая сумма

    // Конструктор: устанавливает theSum в 0
    public ASum() {
        theSum = 0;
    }
```

```
     // Добавление к сумме очередного слагаемого
    public void incSum( int toAdd ) {
        theSum = theSum + toAdd;
    }

    // Просмотр текущего значения
    public int getSum() {
        return theSum;
    }
}

// Основной класс приложения (содержит main)
public class SumArgs {
    public static void main( String args[] ) {

        ASum theSum = new ASum();

        int argsCount; // Счетчик аргументов командной строки

        for( argsCount = 0;
             argsCount < args.length;
             argsCount++ ) {

             int nextArg = Integer.parseInt( args[ argsCount ] );
             System.out.println( "Arg " + argsCount + " parsed: " +
                                 nextArg );

             theSum.incSum( nextArg );

             System.out.println( "Actual sum is " +
                                 String.valueOf( theSum.getSum() ) );
        }

        // Печать результата
        System.out.println( "The final sum is " +
                            String.valueOf( theSum.getSum() ) );
    }
}
```

Компиляция Java-программы в байт-код осуществляется путем вызова компилятора. В режиме командной строки обращение к нему выглядит следующим образом:

```
javac SumArgs.java
```

Для исполнения Java-программы в среде JVM с заданием трех аргументов командной строки, значения которых 10, 20, 30, выполним команду:

```
java SumArgs 10 20 30
```

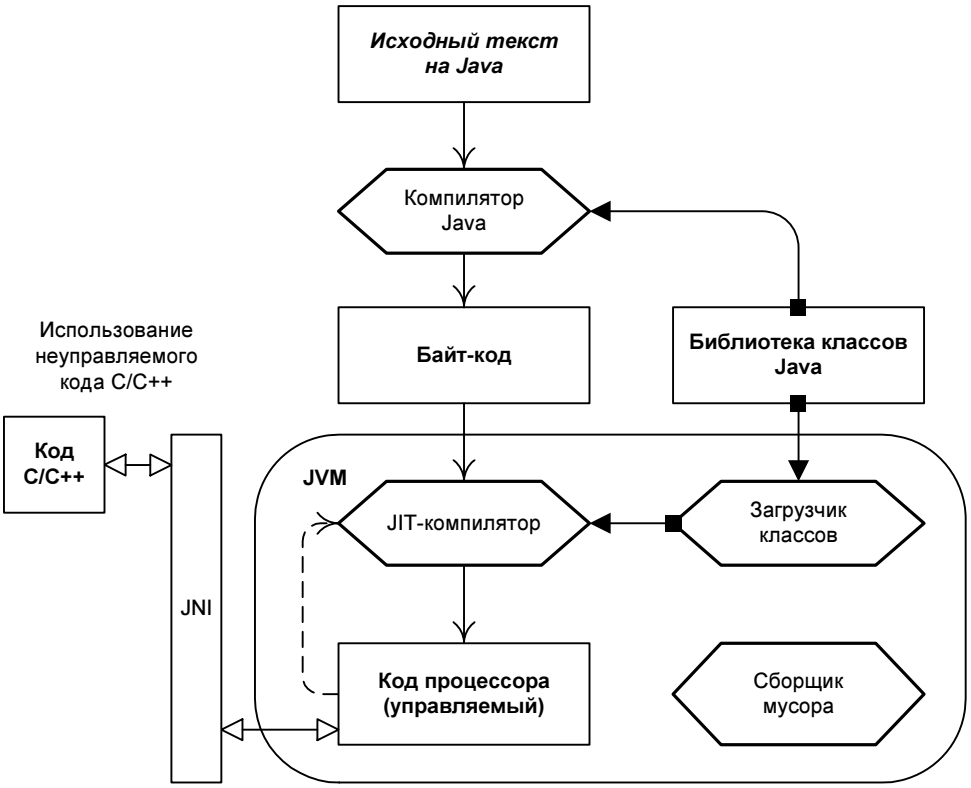


Рис. 10.1. Обработка Java-программы

В результате исполнения программы получим следующие сообщения:

```
Arg 0 parsed: 10
Actual sum is 10
Arg 1 parsed: 20
Actual sum is 20
Arg 2 parsed: 30
Actual sum is 30
The final sum is 30
```

Приведенная в листинге 10.1 программа не содержит контроля успешности получения целого значения в результате разбора строкового аргумента (что, вообще говоря, не очень хорошо). Это упущение программиста не ведет к непредсказуемым последствиям, поскольку необработанные исключения перехватываются JVM. В качестве примера выполним программу, задав некорректное значение аргумента командной строки:

```
java SumArgs 10 qwer
```

В результате исполнения программы получим следующий вывод:

```
Arg 0 parsed: 10
Actual sum is 10
Exception in thread "main" java.lang.NumberFormatException: qwer
        at java.lang.Integer.parseInt(Integer.java:426)
        at java.lang.Integer.parseInt(Integer.java:476)
        at SumArgs.main(SumArgs.java:60)
```

Информация о возникшем исключении содержит трассу стека, что позволяет установить причины допущенной ошибки. Разумеется, программист может встроить в программу собственный обработчик этого исключения.

Взаимодействие Java с неуправляемым кодом

Возможны ситуации, когда программиста не устраивают ограничения, накладываемые управляемым кодом. Речь может идти, например, о соображениях эффективности, об обеспечении охраны авторских прав на оригинальные алгоритмы, об использовании ранее спроектированных компонентов. Среди прочих решений Java предоставляет возможность использовать код, спроектированный на C/C+, посредством специального механизма, называемого Java Native Interface — JNI (интерфейс взаимодействия с родным кодом процессора). Для иллюстрации работы JNI видоизменим решение, представленное в листинге 10.1, таким образом, чтобы метод `incSum()` вызывался как неуправляемый код.

Текст программы на Java изменится незначительно (листинг 10.2).

Листинг 10.2. Использование интерфейса JNI (программа на Java)

```
// Файл SumArgsJNI.java

// Класс-"сумматор"
class ASum {
    int theSum; // Текущая сумма

    // Конструктор: устанавливает theSum в 0
    public ASum() {
        theSum = 0;
    }

     // Добавление к сумме очередного слагаемого
    public native void incSum( int toAdd ); // "native" означает, что
                                            // код этой функции
                                            // импортируется как
                                            // неуправляемый
```

```
    // Просмотр текущего значения
    public int getSum() {
        return theSum;
    }

    // Обеспечивает подключение dll, содержащей код метода
    // incSum, первоначально спроектированный на языке C++
    static {
       System.loadLibrary( "sum_in_c" );
    }
}

// Основной класс приложения (содержит main)
public class SumArgsJNI {
    public static void main( String args[] ) {

        ASum theSum = new ASum();

        int argsCount; // Счетчик аргументов командной строки

        for( argsCount = 0;
             argsCount < args.length;
             argsCount++ ) {

             int nextArg = Integer.parseInt( args[ argsCount ] );
             System.out.println( "Arg " + argsCount + " parsed: " +
                                 nextArg );

             theSum.incSum( nextArg ); // Вызывается родной код
        }

        // Печать результата
        System.out.println( "The final sum is " +
                            String.valueOf( theSum.getSum() ) );
    }
}
```

Из текста видно, что теперь определение функции `incSum()` импортируется из динамически подключаемой библиотеки sum_in_c.dll. От программиста требуется обеспечить доступность этой библиотеки. Механизм JNI предписывает следование определенной процедуре связывания кода на Java с родным кодом процессора (полученным из программы на C/C++). Краткое описание этого механизма поможет нам в дальнейшем при рассмотрении проблем взаимодействия компонентов, разработанных на разных языках программирования. Рис. 10.2 иллюстрирует на примере рассматриваемой программы основные этапы организации взаимодействия таких компонентов с неуправляемым кодом, с использованием интерфейса JNI.

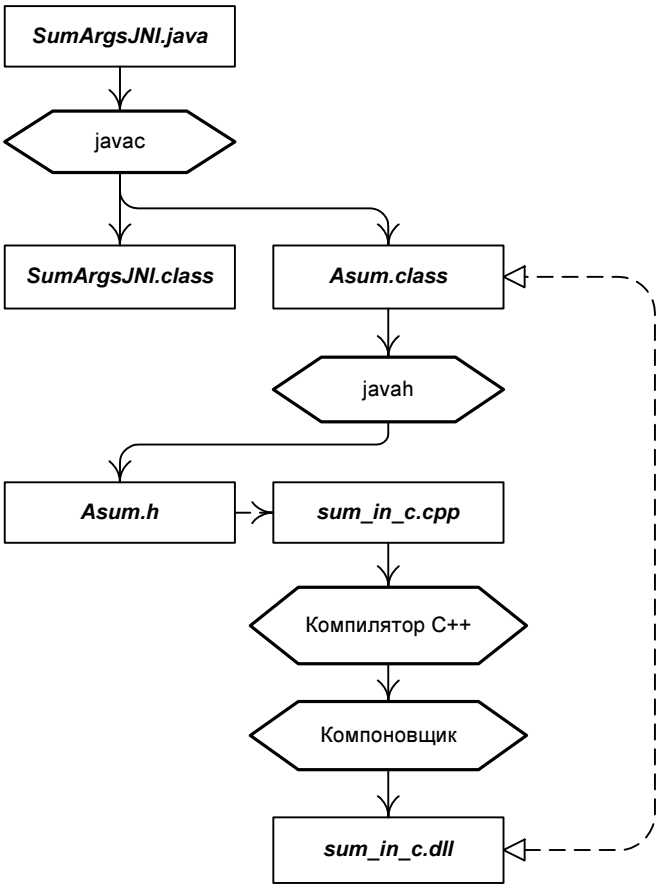


Рис. 10.2. Организация взаимодействия с неуправляемым кодом посредством JNI

В результате компиляции Java-программы мы получаем два файла, содержащие байт-код Java для классов `ASum` и `SumArgsJNI`. Нас сейчас интересует, в первую очередь, класс `ASum`, поскольку именно он содержит импортируемый метод. Специальная программа javah обеспечивает генерацию заголовочного файла C/C++, содержащего среди прочего прототип функции, которая должна быть определена на языке C или C++:

```
javah -jni ASum
```

Ниже приведен результат работы программы javah, в нашем случае — содержимое файла ASum.h (полужирным начертанием выделен интересующий нас прототип):

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class ASum */
```

```
#ifndef _Included_ASum
#define _Included_ASum
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     ASum
 * Method:    incSum
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_ASum_incSum
  (JNIEnv *, jobject, jint);

#ifdef __cplusplus
}
#endif
#endif
```

Как видно из содержимого файла ASum.h, программа javah заменила имя `incSum()` на новое имя, которое и следует использовать при определении этой функции на языке C++. Теперь имеется все необходимое для того, чтобы разработать реализацию метода `incSum()` в виде неуправляемого кода. Результат проектирования представлен в листинге 10.3 (комментарии к тексту поясняют процесс разработки функции, обращающейся к членам класса `ASum`, определенного в файле ASum.java).

Листинг 10.3. Использование интерфейса JNI (код на C++)

```
// Файл sum_in_c.cpp
// Проект: sum_in_c.dll

#include <jni.h> // Для использования интерфейса JNI
#include "ASum.h" // Сгенерирован автоматически программой javah
#include <stdio.h>

// Функция, соответствующая методу incSum класса ASum.java
// NB! Имя функции сконструировано программой javah
//     (см. заголовочный файл "ASum.h"):
//     схема именования (искажения имен) обусловлена
//     требованиями, навязываемыми механизмом JNI
extern "C"
JNIEXPORT void JNICALL Java_ASum_incSum(
        JNIEnv *env, // Указатель на структуру, содержащую данные,
                     // необходимые для работы JNI,
                     // в частности, массив указателей на функции JNI
```

```
        jobject obj, // указатель на объект, на котором
                     // вызывается родной метод
                     // (эквивалентен указателю this в C++)
        jint toAdd   // параметр метода incSum
        ) {

    static int calls = 0; // Счетчик вызовов этой функции

    jclass   cls; // Идентификатор класса Java
    jfieldID fld; // Идентификатор метода класса

    jint     curSum; // Текущая сумма

    cls = env->GetObjectClass( obj ); // Получить идентификатор класса
    fld = env->GetFieldID( cls, "theSum", "I" );
                                      // Получить доступ к
                                      // переменной-члену класса
                                      // "theSum"
    if( fld == 0 ) {
        // Обработка ошибки на случай изменения имен в определении
        // класса ASum.java
        printf( "Error: No field named \"theSum\" in class ASum\n" );
        return;
    }

    // Получить значение переменной "theSum"
    curSum = env->GetIntField( obj, fld );

    // Вычислить новое значение суммы и изменить значение
    // переменной "theSum"
    env->SetIntField( obj, fld, curSum = curSum + toAdd );

    // Печать информационных сообщений
    printf( "Native C function call # % d\n", ++calls );
    printf( "Actual sum is %d\n", curSum );
}
```

Примечание по поводу спецификатора extern "C" см. в разд. 12.2.

Перед компиляцией этой программы следует обеспечить доступность заголовочного файла jni.h и файла jni_md.h, подключаемого из файла jni.h, через задание ключа компилятора `/I` с указанием пути размещения этих файлов на вашем компьютере, например:

```
/Ic:\j2sdk1.4.0\include /Ic:\j2sdk1.4.0\include\win32.
```

После установления сведений о каталогах нужно скомпилировать данную программу в динамически подключаемую библиотеку sum_in_c.dll.

После того как необходимая библиотека получена, ее месторасположение нужно описать в вашем пути к системным библиотекам. Впрочем, в данном случае, вместо того, чтобы редактировать переменные окружения ради одного учебного проекта, вы можете просто скопировать файл sum_in_c.dll в каталог, содержащий байт-код Java-программы SumArgsJNI (т. е. файл SumArgsJNI.class).

Теперь исполним программу, задав аналогичные аргументы командной строки, что и в случае "чистой" Java-программы:

```
java SumArgsJNI 10 20 30
```

Выполнение программы, использующей внешний процессорный код (см. листинг 10.2), приведет к выводу следующих сообщений.

```
Arg 0 parsed: 10
Native C function call #1
Actual sum is 10
Arg 1 parsed: 20
Native C function call #2
Actual sum is 20
Arg 2 parsed: 30
Native C function call #3
Actual sum is 30
The final sum is 30
```

Диагностические сообщения, печатаемые из функции `Java_ASum_incSum()`, подтверждают обращение к функции из библиотеки sum_in_c.dll. Как видно из рассмотренного примера, использование неуправляемого кода в программе на Java не так уж сложно. В то же время, данный механизм не отличается особым изяществом и не распространяется на компоненты, разработанные на других языках программирования.

Ограничения, накладываемые управляемым кодом

Для того чтобы гарантировать безопасность приложения, управляемый код предполагает определенные ограничения на используемые средства программирования. Например, управляемый код не может содержать указателей. Управляемый код не может полагаться на освобождение динамической памяти самим программистом — отсутствие утечек памяти должно быть гарантировано, поэтому системы, работающие с управляемым кодом, снабжаются подсистемой сборки мусора (GC, garbage collection), обеспечивающей автоматическое удаление объектов, размещенных в динамической памяти.

Управляемый код платформы Microsoft .NET

Концепция управляемого кода лежит также в основе платформы Microsoft .NET, спроектированной исходя из идеи не просто кросс-платформенного кода, а кросс-языкового кода. Microsoft .NET предоставляет основу для разработки приложений, использующих компоненты, первоначально написанные на разных языках программирования, хотя, разумеется, наиболее полно концепции, заложенные в основу .NET, воплощаются в языке C#, специально созданном в качестве основного инструмента программирования для .NET Frameworks. Исполнительная система платформы .NET, называемая общеязыковой средой исполнения (Common Language Runtime, CLR), работает с IL-кодом (intermediate language — промежуточный язык), генерируемым компиляторами языков .NET (рис. 10.3).

Каждый CLR-совместимый компилятор наряду с управляемым кодом (IL-code) создает таблицы *метаданных*. Метаданные содержат информацию обо всех программных элементах, определенных в этом модуле (типах, членах типов), а также о программных элементах из других управляемых модулях, на которые имеются ссылки в данном модуле. Метаданные представляют собой важный инструмент обеспечения целостности и надежности управляемого кода. Создаваемые таблицы метаданных, безусловно, ведут к увеличению объектного кода, однако в случае управляемого кода эти затраты оправданы следующими обстоятельствами.

- Метаданные позволяют осуществлять верификацию времени выполнения на безопасность выполняемых кодом операций (вызов методов с корректными наборами аргументов, корректные преобразования типов, отсутствие ошибок обращения к неинициализированной памяти, правильность использования возвращаемых значений и т. п.).
- При компиляции не требуются различные заголовочные и интерфейсные файлы, поскольку вся информация о типах, на которые имеются ссылкы в управляемом коде, находится в таблицах метаданных самого модуля.
- Инструментальные средства интегрированных сред программирования (например, Visual Studio) используют метаданные для получения информации о членах того или иного класса и сигнатурах методов класса. Эти сведения выдаются программисту в виде подсказок, что в конечном счете упрощает и ускоряет проектирование.
- Метаданные используются сборщиком мусора для установления всех зависимостей между объектами и обеспечения правильного освобождения памяти, занятой неиспользуемыми объектами *(см. также разд. 10.3)*.
- Метаданные обеспечивают сериализацию и десериализацию объектов, т. е. сохранение объектов в виде набора байтов с последующим их восста-

новлением при более позднем запуске приложения или при необходимости передать объект в правильном окружении зависимых объектов на удаленный компьютер.

- Метаданные используются механизмом отражения *(см. разд. 11.6)*.

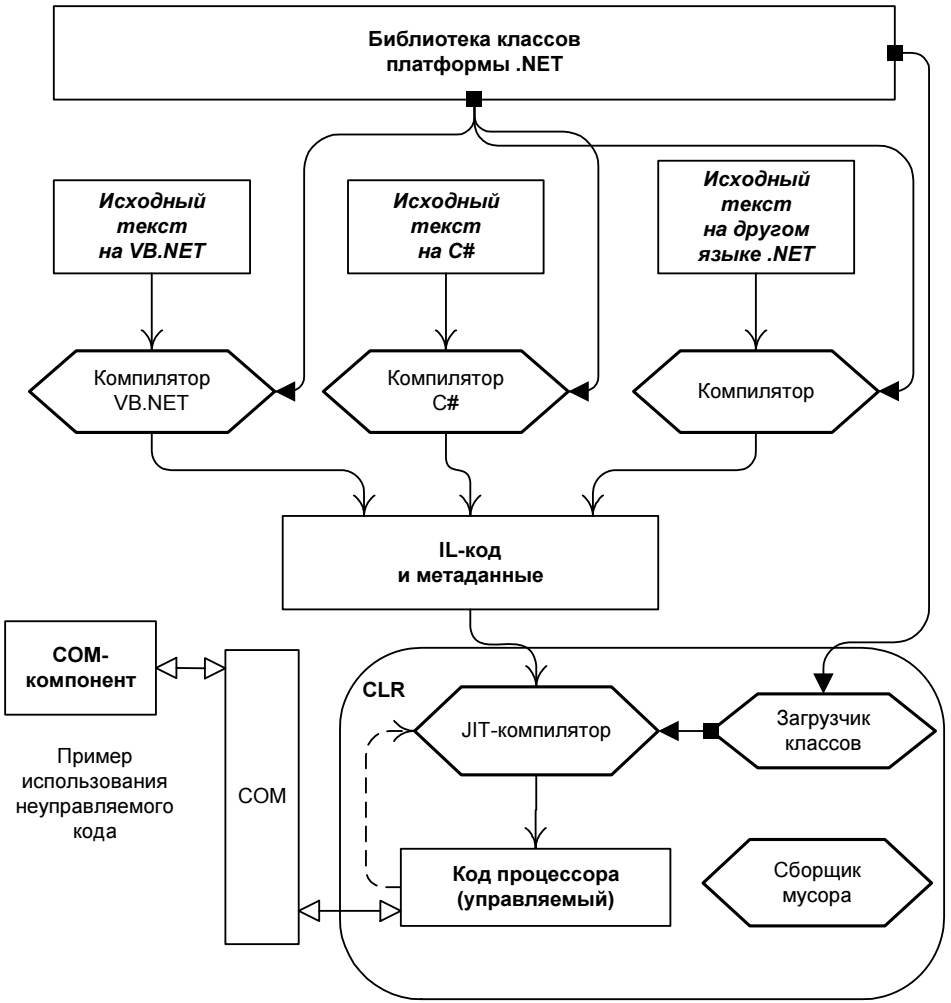


Рис. 10.3. Модель исполнения .NET

Первая реализация CLR была ориентирована на работу под управлением операционных систем Windows. В настоящее время существуют и продолжают разрабатываться реализации CLR и для других вычислительных платформ. Организация взаимодействия компонентов, разработанных на языках, для

которых имеются компиляторы в промежуточный код CLR (IL-код), существенно упрощается, поскольку независимые компоненты распространяются в виде объектных библиотек, содержащих IL-код. При этом языком высокого уровня, в котором наиболее полно воплощены концепции программирования на платформе .NET Frameworks и ее технологические решения, является язык C#. Многие концепции, лежащие в основе языков C# и Java, схожи. Однако C# не следует считать копией Java, использующей синтаксис C++ для выражения сходной с Java семантики. Наряду с возможностями, отсутствовавшими в Java (например, поддержка специализированных атрибутов, встроенные средства для безопасных арифметических вычислений, контролируемых по переполнению), C# наследует ряд средств, привычных для программистов на C++ (гибкая работа с размерными типами, включая возможность передачи экземпляров размерных типов по ссылке, поддержка структур и перечислений, перегрузка операций). Кроме того, разработчики C# реализовали синтаксические конструкции для выражения концепций, реализуемых в других языках, но не имеющих встроенного синтаксиса (события, делегаты, индексаторы). Некоторые из этих концепций рассматриваются в *гл. 11*. Графическую схожесть языков C# и Java иллюстрирует пример решения задачи, *рассмотренной в разд. "Управляемый код в Java"*, на языке C# (листинг 10.4). Обратите внимание на использование встроенных средств языка для контроля переполнения в ходе арифметических вычислений (блок, помеченный спецификатором `checked`).

Листинг 10.4. Суммирование аргументов командной строки (пример на C#)

```
// Файл SumArgs.cs
using System;

// Класс-"сумматор"
class ASum {
    int theSum; // Текущая сумма

    // Конструктор: устанавливает theSum в 0
    public ASum() {
        theSum = 0;
    }

     // Добавление к сумме очередного слагаемого
    public void incSum( int toAdd ) {
        checked {
            theSum = theSum + toAdd;
        }
    }
```

```
    // Просмотр текущего значения
    public int getSum() {
        return theSum;
    }
}

// Основной класс приложения (содержит main)
public class SumArgs {
    public static void Main( String [] args ) {

        ASum theSum = new ASum();

        int argsCount; // Счетчик аргументов командной строки

        for( argsCount = 0;
             argsCount < args.Length;
             argsCount++ ) {

             int nextArg = Int32.Parse( args[ argsCount ] );
             Console.WriteLine( "Arg " + argsCount + " parsed: " +
                                nextArg );
             try {
                 theSum.incSum( nextArg );
             }
             catch( OverflowException ) {
                 Console.WriteLine( "Too large values: overflow" );
                 break;
             }

             Console.WriteLine( "Actual sum is " +
                               ( theSum.getSum() ).ToString() );
        }

        // Печать результата
        Console.WriteLine( "The final sum is " +
                           ( theSum.getSum() ).ToString() );
    }
}
```

В случае если при очередном сложении происходит переполнение, возникает исключение `System.OverflowException`, перехватываемое в теле функции `Main()`. В качестве примера приведем результат вывода программы при следующем формате запуска приложения:

```
SumArgsNET.exe 2 3 2000000000 1000000000
```

Программа выведет следующее:

```
Arg 0 parsed: 2
Actual sum is 2
Arg 1 parsed: 3
Actual sum is 5
Arg 2 parsed: 2000000000
Actual sum is 2000000005
Arg 3 parsed: 1000000000
Too large values: overflow
The final sum is 2000000005
```

Код, безопасный по отношению к типам

Концепция управляемого кода не должна смешиваться с концепцией кода, безопасного по отношению к типам (type-safe code), т. е. кода, совершающего над типами и экземплярами типов только разрешенные операции. Говоря более формально, код считается безопасным по отношению к типам, если выполнены следующие условия (type safety contract):

1. Ссылки на экземпляры типа всегда совместимы с типом, на который они ссылаются.
2. Только правильно определенные и разрешенные операции совершаются над экземплярами этого типа.
3. Используемые сущности строго соответствуют объявлениям, и производятся только те неявные преобразования, которые не могут привести к потере значащих разрядов (примером нарушения этого условия является, скажем, попытка передачи некоторого 64-разрядного целого значения в качестве 32-разрядного целого аргумента).

Код, относительно которого можно в ходе выполнения гарантировать его безопасность по отношению к типам, обычно называют верифицируемым кодом, безопасным по отношению к типам (verifiably type-safe code). Например, компилятор Visual C# .NET всегда порождает управляемый код (т. е. код, требующий CLR для исполнения), но он является верифицируемым кодом, безопасным по отношению к типам, только в том случае, если не содержит блоков, помеченных как unsafe. Unsafe-блоки — это блоки программы, использующие указатели или обращающиеся к внешним неуправляемым компонентам, для которых безопасность типов гарантировать нельзя (например, осуществляют захват внешних по отношению к управляемой среде ресурсов памяти и их освобождение). В действительности, код, не являющийся безопасным по отношению к типам (в том числе неуправляемый код), может и не содержать опасных или некорректных операций, однако гарантировать их отсутствие в ходе выполнения нельзя.

В табл. 10.1 систематизированы основные категории кода, с которыми "имеют дело" управляемые среды [Пышкин, 2004-1].

Таблица 10.1. *Коды, исполняемые в управляемой среде*

Категория кода	Описание	Примеры генераторов кода
Верифицируемый управляемый код, безопасный по отношению к типам (verifiably type-safe managed code)	Управляемый код, созданный в строгом соответствии с условиями кода, безопасного по отношению к типам, для которого можно гарантировать проверку соблюдения этих условий в ходе выполнения	C# Visual Basic .NET Visual J# Java
Неверифицируемый управляемый код, безопасный по отношению к типам (not verifiably type-safe managed code)	Управляемый код, созданный в соответствии с type safety contract, для которого нельзя гарантировать проверку корректности в ходе выполнения. Безопасность такого кода может контролироваться другими средствами, не связанными напрямую с исполнительной системой	C++ C# (программы с unsafe-блоками)
Неверифицируемый неуправляемый код, безопасный по отношению к типам (not verifiably type-safe unmanaged code)	Любой неуправляемый код (например, код из динамически подключаемых библиотек), созданный с максимальным учетом требований безопасности по отношению к типам, но для которого невозможно контролировать соблюдение этих требований в процессе исполнения	C++, не использующий указатели явно
Неуправляемый небезопасный код (unmanaged unsafe code)	Неуправляемый код, созданный без учета соглашений и требований, определяемых управляемой средой исполнения	C/C++

В распределенных средах по умолчанию небезопасный код разрешается исполнять только на локальном компьютере, хранящем этот исполняемый код. Следовательно, код, разрабатываемый для мобильных устройств, должен быть безопасным по отношению к типам.

Вместо резюме

Если управляемый код столь хорош, не должен ли он рано или поздно заменить повсюду использование неуправляемого кода? Иными словами, означает ли это, что преимущества в виде ускорения и упрощения проектирования, получаемые разработчиками и пользователями управляемого кода, приведут

к быстрому вытеснению C++ языками Java и C# и сопутствующими им технологиями? Ответ сегодняшнего дня — нет. В качестве определяющих можно привести следующие факторы.

- Стандартные версии языков Java и C# все же не поддерживают ряд средств, имеющихся в C++ (например, обобщенного программирования с использованием шаблонов).
- Проблема эффективности управляемого кода большей частью разрешается за счет повышения мощности аппаратных средств, в то время как C++ не требует высокопроизводительного оборудования.
- "Управляемые" языки разработки коммерческого программного обеспечения в общем и целом навязывают следование определенному стереотипу проектирования и в основном являются объектно-ориентированными (хотя существуют и версии процедурных, функциональных и логических языков, адаптированные для платформы .NET). Язык C++ поддерживает множество стилей программирования, а не один единственно истинный подход: "Мы знаем из истории массу идеалистов, пытавшихся заставить людей делать что-либо наилучшим образом, и при этом повинных в самых жестоких трагедиях" [Stroustrup, 1998].
- Существует масса областей проектирования, где язык C++ продолжает активно использоваться (разработка встраиваемого программного обеспечения, разработка операционных систем, драйверов и утилит, взаимодействие с ассемблерным кодом, разработка компиляторов и т. д.).
- Человеческий фактор — интерес к языкам C и C++ не ослабевает.

При этом очевидно, что развитие и конкуренция различных моделей программирования ведет, в конечном счете, к развитию проектных идей, систем разработки и в целом — интеллектуального потенциала разработчиков. Ведь в этимологии слова "конкуренция" не только "соревнование", но и "совпадение, согласие, сотрудничество".

10.2. Встроенные, размерные и ссылочные типы

В этом разделе мы рассмотрим основание системы классификации типов, используемой большинством современных объектно-ориентированных языков.

Некоторые типы данных применяются настолько широко, что их поддержка обеспечивается средствами языка. При этом множество этих типов, вообще говоря, не универсально. Несмотря на различия в реализациях, множество концепций, воплощаемых распространенными типами, является ограничен-

ным и конечным. Типы, поддержка которых обеспечивается средствами языка программирования, обычно называются встроенными, или элементарными типами (primitive types). Остальные типы либо определяются пользователем (user-defined types), либо библиотеками типов (library types), подобными .NET Framework Class Library.

Большинство встроенных типов обычно являются размерными типами (value types). Экземпляры (переменные) этих типов обычно размещаются в стеке приложения. Это позволяет автоматически очищать память, занятую экземплярами размерных типов при выходе их за пределы области действия. Переменная размерного типа предоставляет непосредственный доступ к данным. Ниже приведены примеры размерных типов на C#:

```
// Фрагмент текста  на C#
struct Y {
    // ...
}
void ValueTypesExample() {

    char c;   // Переменная встроенного типа
    Y yStr;   // Переменная структурного типа
    string s; // Переменная строкового типа
    // ...
}
```

Под ссылочными типами (reference types) обычно понимаются типы, экземпляры которых хранятся в динамической памяти (и, возможно, подлежат обработке сборщиком мусора — *см. разд. 10.3*). Представляющая экземпляр переменная неявно содержит адрес объекта в динамической памяти, поэтому при обращении к данным ссылочного типа происходит разыменование (dereference) экземпляра. Классы C# (равно как и классы Java) являются ссылочными типами:

```
// Фрагмент текста  на C#
class X {
    // ...
}
void ReferenceTypeExample() {

    X xObj = new X(); // Объектная переменная
    // ...
}
```

В некотором смысле ссылочный тип напоминает адресацию объекта, размещенного в динамической памяти, посредством указателя в языке C++. Одна-

ко это весьма приблизительная аналогия. Дело в том, что в C++ вообще нельзя характеризовать принадлежность типа к ссылочному или размерному, поскольку то, где будет размещен экземпляр типа, зависит от особенностей кода программы:

```
// Фрагмент текста на C++
class X {
public:
    void f();
    // ...
};
void CPPTypesExample() {

    int a;   // Данное целого типа размещается в стеке
             // (тип ведет себя как размерный)
    int *pa = new int[ 1 ];
             // Данное целого типа размещается в
             // динамической памяти
             // (тип ведет себя подобно ссылочному)
    *pa = 1; // Обращение к экземляру данных
             // посредством явного разыменования
    // ...
    X xObj;  // Объект размещается в стеке,
             // (тип ведет себя как размерный)
    xObj.f();// Обращение к полям объекта
             // непосредственно через объектную переменную

    X *pObj = new X;
            // Объект размещается в динамической памяти
            // (тип ведет себя подобно ссылочному)

    (*pObj).f(); // Обращение к полям объекта
                 // посредством явного разыменования
    // ...
}
```

В отличие от "чистых" ссылочных типов, модель связывания с объектами посредством указателя требует явного разыменования указателя при обращении к экземпляру типа в динамической памяти. Кроме этого, ссылка, в отличие от указателя, по определению не может быть неинициализированной (в противном случае транслятор выдаст сообщение об ошибке); значение ссылки нельзя изменить *(см. также информацию о перехвате исключений по ссылке и по значению указателя в разд. 6.2 и в листинге 6.4)*. Таким образом, использование ссылочных типов вместо указателей исключает один из основных источ-

ников ошибок в программах на C/C++ — неинициализированные или неправильно инициализированные указатели и некорректное изменение значения указателя (рис. 10.4). Итак, в неуправляемом коде C/C++ объявление типа не определяет, где будет размещен экземпляр типа — это "решает" код. В управляемом коде разработчик, выбирая представление для типа (например, структурный тип или класс), однозначно указывает, где будут размещаться экземпляры этого типа, а разработчик, использующий тип в своем коде, управлять размещением типа не может. Наряду с использованием сборщика мусора, это исключает ошибки управления памятью в клиентском коде.

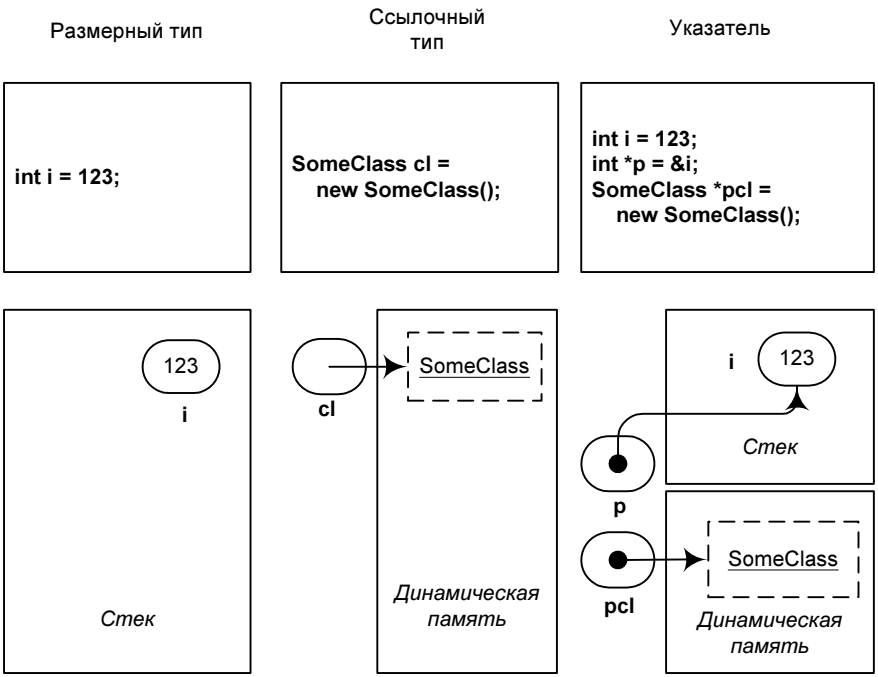


Рис. 10.4. Размерные, ссылочные типы и указатели

В C++ преимущественное использование указателей для связи с объектами, размещенными в динамической памяти, с одной стороны, связано с часто требующейся необходимостью совместимости с кодом на C, с другой стороны, вероятно, историческими причинами. В принципе, ничто не мешает использовать в программах на C++ для этой цели ссылочные переменные:

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

```
class X {
public:
    int a;
};

void main() {

    X& refX = *( new X );
    refX.a = 10;

    cout << refX.a << endl;

    delete &refX;
}
```

Хотя этот код и выглядит немного экзотично, он более безопасен, чем "классический" аналог:

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

class X {
public:
    int a;
};

void main() {

    X* ptrX = new X;
    ptrX->a = 10; // или ( *ptrX ).a = 10;

    cout << ptrX->a << endl;

    delete ptrX;
}
```

Указатель `ptrX` можно объявить и инициализировать (по ошибке) произвольным значением, что невозможно для ссылки `refX`.

Потенциальная небезопасность кода, содержащего указатели, привела к тому, что некоторые объектно-ориентированные языки не поддерживают указателей вовсе (Java), а некоторые, допуская принципиальную возможность использования указателей, ограничивают их применение (Object Pascal, C#).

Объекты и встроенные типы

В ряде случаев может возникать необходимость обрабатывать экземпляры встроенных типов как объекты (таким образом, требуется создать объект в динамической памяти, представляющий экземпляр встроенного типа). Такая

ситуация может возникать в случае необходимости использования контейнера, вмещающего экземпляры встроенных типов. В языках, построенных на основе принципа однокоренной иерархии классов, универсальность контейнеров обеспечивается тем, что контейнеры хранят не сами объекты, а ссылки на них (фактически, ссылки на `Object`). Таким образом, контейнеры обеспечивают обработку только тех объектов, которые размещены в динамической памяти и управляются ссылками.

Примечание

В C++ гибкость контейнеров обеспечивается механизмом параметрического полиморфизма, в основе которого лежит конструирование шаблонных типов с использованием ключевого слова `template`. *О проектировании контейнерных классов в стиле стандартной библиотеки шаблонов см. гл. 9.*

Механизмы реализации представления экземпляров встроенных типов в виде объектов могут различаться в разных языках. Так, в Java для этой цели используются так называемые классы-оболочки (wrappers), например:

```
int value = 32; // Экземпляр встроенного типа
Integer intObj = new Integer( value ); // Объект типа-оболочки
```

Объект класса `Integer` содержит единственное целочисленное значение (рис. 10.5).

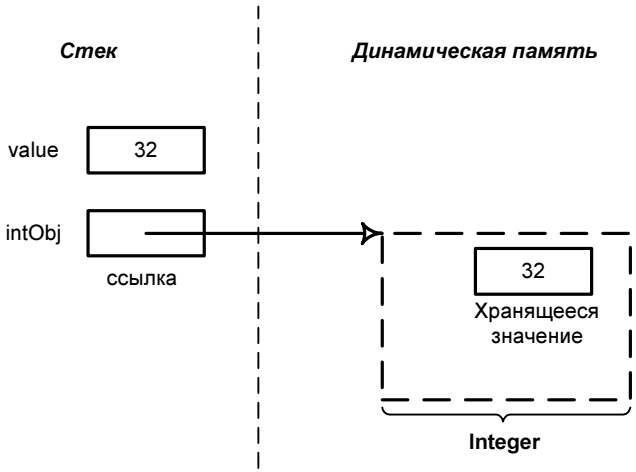


Рис. 10.5. Классы-оболочки в Java

Классы-оболочки обеспечивают также полезную дополнительную функциональность в связи с обработкой данных представляемого типа, например, класс `Integer` содержит, в частности, следующие методы:

- конструктор, позволяющий создать объект класса-оболочки на основе получаемого в качестве аргумента значения типа `int`;
- конструктор, позволяющий создать объект класса-оболочки на основе получаемого в качестве аргумента значения типа `String` (т. е., по существу, исполняющий роль неявного преобразования из `String` в `Integer`);
- метод `intValue()`, возвращающий значение целого числа, хранящегося в объекте класса `Integer`;
- статический метод `ParseInt()`, обеспечивающий преобразование строкового объекта в целое число;
- метод `toBinaryString()`, обеспечивающий преобразование значения, хранящегося в объекте `Integer`, в строку, хранящую двоичное представление этого значения;
- метод `toHexString()`, обеспечивающий преобразование значения, хранящегося в объекте `Integer`, в строку, хранящую двоичное представление этого значения, и др.

Таким образом, в качестве компенсации за неэффективность хранения данных встроенных типов в виде объектов программист получает возможность использовать различные функции обработки этих данных, объявленные в пространстве имен класса-оболочки.

По-другому решена обозначенная проблема управления экземплярами встроенных типов разработчиками обобщенной среды исполнения программ платформы Microsoft .NET. Здесь данные встроенных типов всегда размещаются в стеке, и у программиста нет явной возможности разместить соответствующий объект в динамической памяти (по крайней мере, до тех пор, пока он находится в рамках ограничений, накладываемых управляемым кодом). Вот маленький пример на C#:

```
int value1; // Экземпляр создается в стеке
System.Int32 value2 = new System.Int32(); // Экземпляр создается в стеке!
```

Если же необходимость помещения объекта в динамическую память продиктована особенностями кода, применяется так называемая *упаковка* (boxing) размерного типа. Упаковка осуществляется неявно:

```
int inputValue = 32;
Queue intQueue = new Queue ();
intQueue.Enqueue( inputValue ); // Добавить элемент в очередь
```

Поскольку контейнер-очередь может хранить только ссылки на объекты, размещенные в динамической памяти, происходит создание объекта в динамической памяти, куда копируется значение `inputValue`. Ссылка на создан-

ный объект передается в качестве аргумента метода `Enqueue()`. Обратите внимание, что созданный объект недоступен программисту — нигде нет явной ссылочной переменной, связанной с созданным объектом (рис. 10.6).

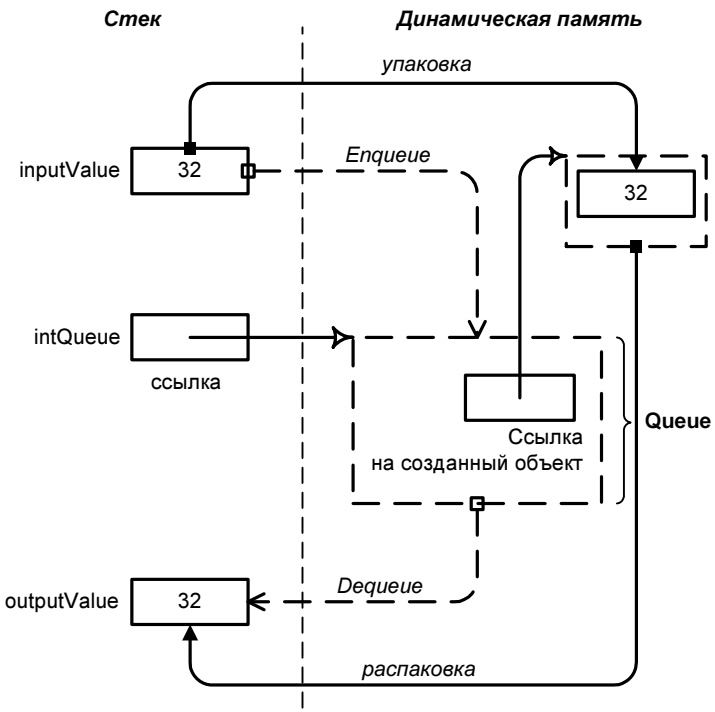


Рис. 10.6. Действие механизма упаковки/распаковки объектов в .NET

В ходе извлечения объекта из контейнера реализуется (неявно) механизм, называемый *распаковкой* объекта (unboxing):

```
int outputValue =intQueue.Dequeue();
```

При распаковке извлекается адрес поля, хранящего целочисленное значение. Затем происходит копирование значения, хранящегося по этому адресу, в переменную `outputValue`.

10.3. Автоматическое удаление объектов (сборка мусора)

В C++ очистка памяти от объектов, размещенных в динамической памяти, осуществляется вручную — посредством использования операции `delete`.

Объекты, размещенные в стеке, удаляются автоматически при выходе объекта из области видимости. Многие ошибки, возникающие в неуправляемом коде, зачастую обусловлены ошибками в связи с освобождением памяти, наиболее распространенными из которых являются два типа ошибок:

- отсутствие обязательного освобождения памяти;
- попытка использовать уже освобожденную память.

Рихтер указывает, что эти два вида ошибок обычно опаснее остальных, поскольку трудно предвидеть их последствия и периодичность появления [Richter, 2002]. По этой причине ручное управление памятью неприемлемо в программировании, ориентированном на исполнение программ в контролируемых средах. Поэтому технологии, ориентированные на контролируемый код, обычно поддерживают автоматическое освобождение памяти, занятой объектами, на которые больше нет ссылок в программе. Процесс автоматического удаления ненужных объектов называется *сборкой мусора* (garbage collection).

Сборка мусора, конечно, не решает всех проблем разрушения объектов, поскольку существуют случаи, когда объект в ходе своей работы занимает какие-то особые ресурсы, которые не могут быть освобождены сборщиком мусора, например:

```
// Фрагмент текста на Java

class Special {
    public Special() {

        // Занимает внешние ресурсы
        // Например, вызывает код на C,
        // использующий malloc()
        // (как вызвать код на C — см. в разделе 10.1)
        // ...
    }
    // ...
    // ??? Необходимо обеспечить освобождение ресурсов,
    // занятых внешним кодом
}
```

Для таких случаев предусматривают специальный механизм завершения, например, в Java существует метод `finalize()`, который разработчик может определить в своем классе. Когда сборщик мусора принимает решение освободить память, занятую объектом, то непосредственно перед тем, как сделать это, он вызывает метод `finalize()`:

```
class Special {
    public Special() {

        // Занимает внешние ресурсы.
        // Например, вызывает код на C,
        // использующий malloc()
        // ...
    }

    protected void finalize {

        // Код, необходимый для корректного завершения
        // взаимодействия с внешними ресурсами.
        // Например, вызывает код на C,
        // использующий free()
        // ...
}
```

Важно понимать, что метод `finalize()` не является аналогом деструктора C++. Здесь следует иметь в виду два обстоятельства.

- В корректной программе на C++ объекты всегда разрушаются, и вызов деструктора гарантирован при выходе объекта из области видимости или при удалении из динамической памяти. В Java эта модель не поддерживается, и деструкторы не используются.
- Точный момент освобождения памяти в ходе сборки мусора неизвестен. Сборщик мусора не всегда удаляет объекты, память освобождается сборщиком мусора тогда, когда в этом имеется потребность (например, программа приближается к точке критического расхода ресурсов). Поэтому если объекту нужны некоторые обязательные завершающие операции, следует написать специальный метод и вызвать его. Метод `finalize()` для этой цели не подходит, т. к. до его вызова дело может не дойти. Явный вызов методов, ответственных за реализацию корректного разрушения инфраструктуры объекта, позволяет сделать процесс завершения более наблюдаемым, однако при этом нельзя обеспечить гарантированное однократное выполнение завершающих действий: программист может забыть вызывать соответствующий метод или вызвать его более одного раза.

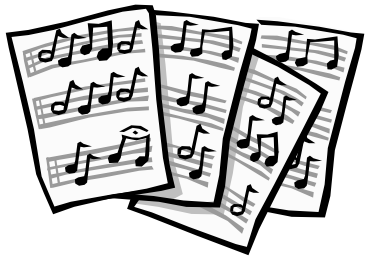
Несмотря на то, что в различных исполняющих системах реализованы разные подходы к сборке мусора, основной принцип неизменен: программист не должен заботиться об освобождении занятой динамической памяти. Таким образом, нет необходимости в специальных средствах языка, отвечающих за обеспечение корректного удаления объектов из памяти (в частности, как следует из изложенного выше, нет необходимости в деструкторах в их традиционном понимании).

Примечание

Оговорка "в традиционном понимании" неслучайна. К сожалению, даже разработчики языков не всегда осмотрительны в выборе используемых терминов. Например, разработчики C# ошибочно назвали деструктором то, что по определению деструктором не является. То, что в C# называется деструктором, не просто замещается методом `Finalize()` (назначение которого аналогично методу `finalize()` в Java). В метод `Finalize()`, автоматически конструируемый на основе так называемого деструктора, встраивается блок `try` на случай исключений, генерируемых в ходе завершения, и вызов завершителя базового класса в блоке `finally`, следовательно, речь не идет просто об "удобном синтаксисе". Рихтер правильно замечает по этому поводу, что разработчики C# допустили ошибку не только в том, что воспользовались синтаксисом, идентичным синтаксису деструктора C++, но и в том, что назвали этот элемент класса C# деструктором [Рихтер, 2002]: "Не позволяйте синтаксису деструктора одурачить вас" — семантика совершенно другая. Причем такая особенность завершения не относится к платформе .NET в целом, речь идет именно об особенностях синтаксиса и семантики конструкции языка C#.

В этой книге мы не будем рассматривать конкретные реализации механизма сборки мусора, оставляя это читателям для самостоятельной работы. Хорошие описания сборщиков мусора читатель может найти в различных источниках (см., например, [Naughton, Schildt, 1999], [Eckel, 2000-2], [Richter, 2002], [Дубовцев, 2004]).

ГЛАВА 11



Развитие объектно-ориентированной модели управления типами

Для полноты рассмотрения основных средств объектно-ориентированного проектирования необходимо принять во внимание два фактора, которые со всей очевидностью могут быть обнаружены в ходе анализа современных объектно-ориентированных языков. Во-первых, реализация основных концепций и механизмов ООП может отличаться от языка к языку. При этом квалифицированный разработчик должен видеть как общие элементы, формирующие основу концептуальной модели, так и особенности, обусловленные философией и проектными принципами, составляющими основание конкретного языка программирования. Во-вторых, некоторые языки могут предоставлять средства, которые, вполне адекватно вписываясь в объектно-ориентированную парадигму, не реализованы в других языках. Как правило, это не означает невозможность реализации подобных механизмов средствами других языков, а просто предоставление изобразительных средств для более наглядного воплощения абстракций предметной области в объектно-ориентированной программе. В данной главе мы сосредоточим основное внимание именно на тех особенностях реализации объектной модели, которые обусловлены указанными факторами.

11.1. Реализация полиморфизма

Обеспечивающее полиморфизм динамическое связывание является одной из основных концепций, лежащих в основе ООП. В *главе 5 были рассмотрены* механизмы реализации полиморфизма в C++. Напомним, что в соответствии с соглашениями, принятыми в C++, возможность полиморфного поведения объектов некоторого класса требует явного разрешения (посредством объявления хотя бы одной виртуальной функции-члена класса). Если определение класса не содержит виртуальных функций, объекты этого класса не поддер-

живают полиморфное поведение, равно как и отсутствует возможность использования динамического приведения типов для указателей или ссылок на эти объекты.

Java: полиморфизм по умолчанию

В языке Java используется иная модель реализации полиморфизма. По умолчанию все методы класса перезаписываемы (т. е., в терминах C++, виртуальны). Сравните пример на C++ листинга 5.8 с примером эквивалентного по функциональности приложения на Java (листинг 11.1).

Листинг 11.1. "Пример с поеданием апельсина" на Java

```
abstract class Fruit {
    public void Eat() {
        System.out.println( "Inside Fruit.Eat()" );

        while( !IsEaten() ) {
            TakePiece();
        }
        FinishEating();
    }
    public abstract boolean IsEaten();
    public abstract void TakePiece();

    public void FinishEating() {
        // По умолчанию никаких действий
    }

    // NB! Внимание: метод finalize() не эквивалентен
    // деструктору C++!
    protected void finalize() {
        System.out.println( "Fruit cleaned up" );
    }
};

class Orange extends Fruit {
    int pieces;

    public Orange() {

        pieces = 8;
        System.out.print( "Got an orange of " );
        System.out.println( pieces + " pieces!" );
    }
```

```
    public Orange( int _pieces ) {

        pieces = _pieces;
        System.out.print( "Got an orange of " );
        System.out.println( pieces + " pieces!" );
    }

    public boolean IsEaten() {
        System.out.println("Inside Orange.IsEaten()");

        return pieces > 0 ? false : true;
    }

    public void TakePiece() {
        System.out.println( "Inside Orange.TakePiece(): " );

        pieces--;
        System.out.println( pieces + " piece(s) left." );
    }

    public void FinishEating() {
          System.out.println( "Whole orange has been eaten" );
    }

    // NB! Внимание: метод finalize() не эквивалентен
    // деструктору C++!
    protected void finalize() {
        System.out.println( "Orange cleaned up" );
    }
};

public class FruitsApp {

    public static void main( String[] args ) {
        Fruit refFruit = new Orange( 4 );
        refFruit.Eat();
    }
}
```

Таким образом, полиморфное поведение объектов Java обеспечивается автоматически. Никаких дополнительных указаний для этого не требуется. Подразумеваемая полиморфность методов Java обеспечивает способность библиотечного кода вызывать методы на экземплярах новых классов без перекомпиляции библиотечного кода.

Наоборот, если разработчик класса хочет быть уверен в неизменности семантики метода класса, он может запретить дальнейшее переопределение класса посредством специального служебного слова `final`, например:

```
class Fruit {
    public final void Eat() { // Нельзя переопределить
                              // в производных классах
        System.out.println( "Inside Fruit.Eat()" );

        while( !IsEaten() ) {
            TakePiece();
        }
        FinishEating();
    }
}
```

В этом случае невозможно перезаписать (override) метод `Eat()` в производном классе.

Таким образом, модель реализации наследования в Java предполагает, что все решения о возможности или невозможности замещения методов класса принимаются при проектировании базового класса: нет возможности скрыть (hide) метод базового класса без замещения, что является обычной ситуацией для C++ *(см. также разд. 4.4)*.

C#: спецификатор *new*, версии классов и виртуальные функции

Модель, используемая в языке C#, на первый взгляд, напоминает модель C++. Аналогично C++, объявление метода виртуальным осуществляется явно (посредством спецификатора `virtual`). В противном случае одноименный метод производного класса скрывает метод базового класса (так же, как и в C++). Однако C# предоставляет и новые средства. Во-первых, виртуальная функция требует явного объявления не только в базовом классе (для этой цели используется спецификатор `virtual`), но и в производном классе (для этого используется спецификатор `override`):

```
class Orange : Fruit {
    // ...

    public override void FinishEating() {
        // ...
    }
    // ...
}
```

Если спецификатор `override` отсутствует, функция скрывает версию одноименного метода базового класса, и компилятор выдает предупреждение, ре-

комендуя явно подтвердить это посредством использования специального служебного слова new:

```
new public void FinishEating() {
    // ...
}
```

Такое объявление означает, что данный метод не участвует в разрешении виртуальных вызовов, т. е. считается новым методом, никак не связанным с одноименным методом базового класса. Впрочем, даже если опустить служебное слово new, реализуется именно такое поведение объектов производных классов. Зачем же разработчики языка придумали новое служебное слово? Для прояснения этого вопроса рассмотрим одну из распространенных проблем, возникающих при использовании стороннего кода.

Одним из недостатков, присущих объектно-ориентированному проектированию, является возникновение неявных и непредусмотренных взаимозависимостей, обусловленных, в частности, использованием в программах наследования [Лекарев, 1997]. Одной из форм таких взаимозависимостей являются версии виртуальных функций, которые могут появиться при внесении изменений в библиотечный код (т. е. код, реализация которого не обязательно доступна клиентскому программисту). Проблему в связи с возникновением версий виртуальных функций называют иногда проблемой хрупкого базового класса (fragile base class). Рассмотрим эту ситуацию на небольшом примере на C#.

Предположим, что в составе некоторой библиотеки имеется класс, обеспечивающий организацию работы с неким внешним ресурсом (листинг 11.2).

Листинг 11.2. Fragile base class problem. Библиотечный компонент

```
using System;
namespace LibraryComponent {

    public class ResourceManager {
        public virtual void GetResource() {
            Console.WriteLine( "Library: resource accessed" );
            // Какие-то действия...
        }
        // ...
    }
}
```

Допустим, что разработчик, использующий библиотечный класс, создал на его основе "улучшенное" определение. Модифицированная версия функции

`GetResource()` содержит, в частности, метод `Check()`, который, например, проверяет достаточность дискового пространства (листинг 11.3).

Листинг 11.3. Fragile base class problem. Клиентский код

```
using System;
using LibraryComponent;

namespace ClientComponent {

    public class CustomResourceManager : ResourceManager {

        public override void GetResource() {

            Console.WriteLine( "Client: accessing resource..." );
            // Какие-то действия, специфичные для
            // клиентского кода...

            Check();

            // Остальные действия
            // (обеспечиваются библиотечным классом)
            base.GetResource();
        }

        protected virtual void Check() {
            Console.WriteLine( "Client: freespace checked" );
            // Проверка наличия места на диске...
        }
    }
    // ...
    public class ClientApp {
        static void Main() {
            ResourceManager rm = new CustomResourceManager();
            rm.GetResource();
        }
    }
}
```

Исполнение клиентской программы порождает ожидаемый вывод:

```
Client: accessing resource...
Client: freespace checked
Library: resource accessed
```

Представим теперь, что по прошествии некоторого времени выпускается новая версия библиотеки, у которой изменена реализация метода `GetResource()` (листинг 11.4).

Листинг 11.4. Fragile base class problem. "Новая версия" библиотеки

```
using System;
namespace LibraryComponent {
    public class ResourceManager {
        public virtual void GetResource() {
            // Сначала — некоторая предварительная проверка
            Check();
            // Затем — еще какие-то действия...
            Console.WriteLine( "Library: resource accessed" );
        }
        protected virtual void Check() {
            Console.WriteLine( "Library: authentication..." );
            // Действия, никак не связанные с клиентской
            // версией Check(), например
            // проверка аутеничности...
        }
        // ...
    }
}
```

Что произойдет при перекомпиляции представленной в листинге 11.3 клиентской программы с новой версией библиотеки? Внимательный читатель может дать ответ на этот вопрос: компилятор C# выдаст предупреждение, содержащее примерно такое сообщение:

```
'ClientComponent.CustomResourceManager.Check()' hides inherited member
'LibraryComponent.ResourceManager.Check()'. To make the current member
override that implementation, add the override keyword. Otherwise add the
new keyword.
```

Возникшую при подобном разрешении виртуальных вызовов коллизию иллюстрирует рис. 11.1.

Таким образом, от программиста требуется явно указать правильную (с точки зрения клиентского разработчика) семантику `Check()`. Если версия `Check()` из производного класса должна считаться новым определением, никак не связанным с методом базового класса, в заголовке функции нужно поместить спецификатор `new` (листинг 11.5).

Листинг 11.5. Fragile base class problem. Новый клиентский код (вариант с *new*)

```
using System;
using LibraryComponent;
```

```
namespace ClientComponent {

    public class CustomResourceManager : ResourceManager {

        public override void GetResource() {

            Console.WriteLine( "Client: accessing resource..." );
            // Какие-то действия, специфичные для
            // клиентского кода...

            Check();

            // Остальные действия
            // (обеспечиваются библиотечным классом)
            base.GetResource();
        }

        new protected virtual void Check() {
            Console.WriteLine( "Client: freespace checked" );
            // Проверка наличия места на диске...
        }
    }
    // ...
    public class ClientApp {
        static void Main() {
            ResourceManager rm = new CustomResourceManager();
            rm.GetResource();
        }
    }
}
```

В этом случае при вызове метода `Check()` из метода `GetResource()` производного класса вызывается версия `Check()` из производного класса, а при вызове метода `Check()` из метода `GetResource()` базового класса вызывается версия `Check()` из базового класса, а значит — вывод программы будет иметь следующий вид:

```
Client: accessing resource...
Client: freespace checked
Library: authentication...
Library: resource accessed
```

Если же версию `Check()` из производного класса следует считать замещением (overriding) "новоявленной" `Check()` из базового класса, то в заголовке функции производного класса нужно поместить спецификатор `override` (листинг 11.6).

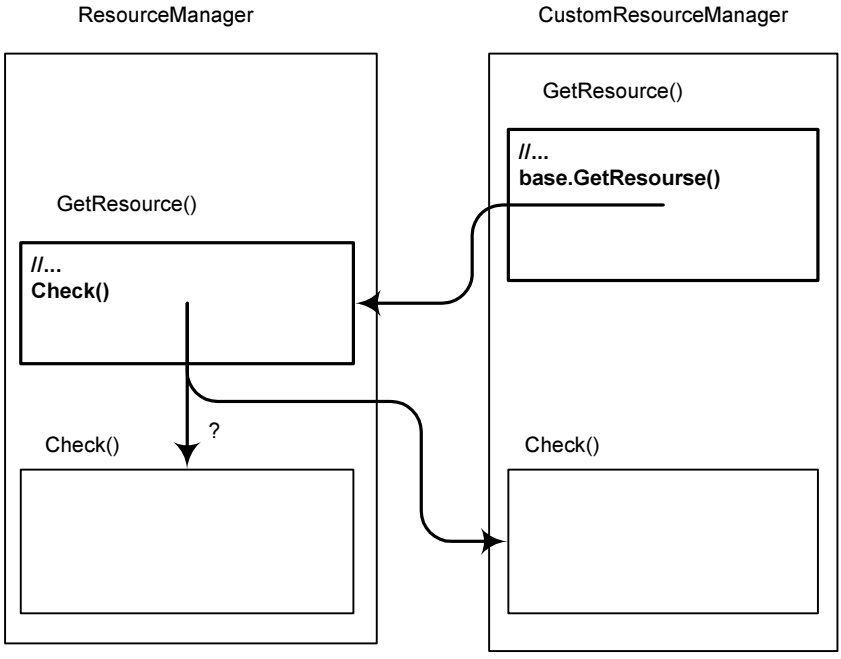


Рис. 11.1. Версии виртуальных функций

Листинг 11.6. Fragile base class problem. Новый клиентский код (вариант с *override*)

```
using System;
using LibraryComponent;

namespace ClientComponent {

    public class CustomResourceManager : ResourceManager {

        public override void GetResource() {

            Console.WriteLine( "Client: accessing resource..." );
            // Какие-то действия, специфичные для
            // клиентского кода...

            Check();

            // Остальные действия
            // (обеспечиваются библиотечным классом)
            base.GetResource();
        }
```

```
        protected override void Check() {
            Console.WriteLine( "Client: freespace checked" );
            // Проверка наличия места на диске...
        }
    }
    // ...
    public class ClientApp {
        static void Main() {
            ResourceManager rm = new CustomResourceManager();
            rm.GetResource();
        }
    }
}
```

Теперь и при вызове метода `Check()` из метода `GetResource()` производного класса, и при вызове метода `Check()` из метода `GetResource()` базового класса всегда вызывается версия `Check()` из производного класса. Это подтверждает анализ вывода, порождаемого программой листинга 11.6:

```
Client: accessing resource...
Client: freespace checked
Client: freespace checked
Library: resource accessed
```

Как трактовалась бы подобная ситуация в "мире" C++ или Java? Метод базового класса является виртуальным (в C++ из-за явного `virtual`, в Java — по умолчанию). Поэтому при реализации полиморфного поведения объекта класса `CustomResourceManager` обращение к методу базового класса `GetResource()` (см. инструкцию `base.GetResource()` в клиентском коде) приведет к вызову версии метода `Check()` из производного класса. Это вовсе не обязательно совпадает с намерениями программиста. Никаких предупреждений не было бы выдано, и код был бы успешно скомпилирован.

В условиях, когда программное обеспечение не обязательно разрабатывается усилиями разработчиков одной компании, полученная во время компиляции информация о возможном изменении семантики полиморфных вызовов является очень ценной. В свою очередь, использованием служебного слова `new` программист явно сообщает читателю программы о своих намерениях и исключает возможные некорректные интерпретации кода.

Впрочем, если следовать принципам, сформулированным Фаулером, код, использующий метод, определяемый со спецификатором `new`, вероятно, является кандидатом для осуществления рефакторинга [Fowler, 1999]. Если версия метода `Check()` в клиентском коде действительно не имеет ничего общего с методом из базового класса, то при наличии доступа к представлению кли-

ентского класса наиболее разумным решением будет переработка определения класса `CustomResourceManager` с переименованием или исключением метода `Check()`. Если класс `CustomResourceManager` является объектом повторного использования, то рефакторингу, возможно, придется подвергнуть не только определение этого класса, но и определения зависимых от него классов.

11.2. Свойства

В соответствии с концепцией сокрытия реализации, довольно распространена практика, когда доступ к переменным-членам класса извне закрыт, а их модификация или считывание записанных в них значений осуществляется посредством специальных функций.

```
// Код C++

class X {
    int internal; // закрытая переменная
public:
    SetInternal( int value ) {
        // Возможно, какая-то обработка значения value...
        // (например, проверка корректности)

        // ... и только затем присваивание
        internal = value;
    }
    int GetInternal() {
        return internal;
    }
    // ...
}
```

Это позволяет предотвратить некорректные модификации переменных-членов класса или ограничить множество возможных модификаций.

Если подобных переменных-членов класса довольно много, то использование такого подхода может приводить к загромождению интерфейса класса множеством пар функций типа *get/set*, а также, в определенных случаях, — к ухудшению показателей эффективности программы. С накладными расходами на вызов функций можно, конечно, бороться за счет объявления подобных функций подставляемыми (inline), однако проблемы увеличения размера кода и усложнения интерфейса класса в целом остаются нерешенными.

Стандартным путем разрешения подобных ситуаций является реализация концепции свойств. Свойство (property) является именованным блоком

команд, доступ к которому извне осуществляется так же, как доступ к обычной переменной-члену класса. Рассмотрим пример реализации концепции свойства в языке C#.

Поддержка свойств на уровне языка (на примере C#)

Примером языка, поддерживающего концепцию свойств посредством специального синтаксиса, является язык C# (рис. 11.2).

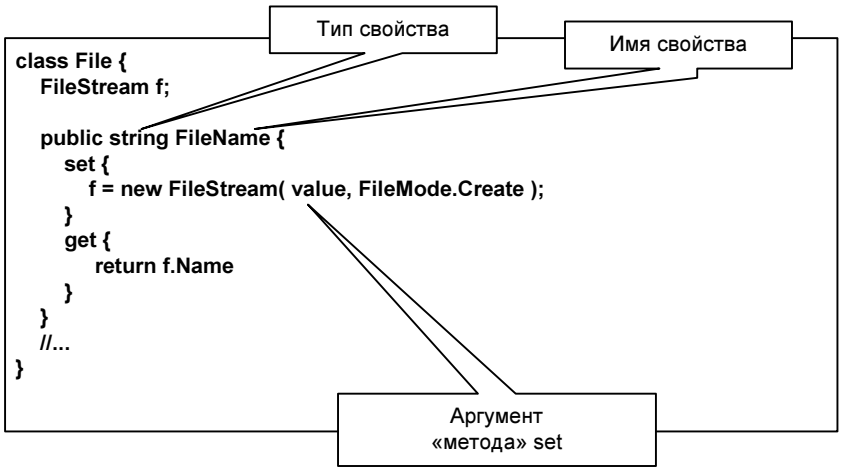


Рис. 11.2. Реализация свойств в языке C#

Для определения свойства в общем случае требуется определить два метода-аксессора, неявно вызываемых при доступе к свойству. Метод-аксессор `set` определяет семантику доступа к свойству по записи, а метод-аксессор `get` — семантику доступа по чтению. Извне доступ к свойству подобен доступу к обычной переменной-члену класса:

```
File fd = new File(); // см. рис. 11.2
fd.FileName = "input.txt"; // вызывается set( "input.txt" )
//...
string s = fd.FileName; // вызывается get()
```

Применение концепции свойств позволяют поддержать на уровне языка следующие возможности:

- разделение интерфейса члена класса и действий, связанных с обращением к члену класса;

- возможность пересмотра реализации доступа к членам класса без изменения интерфейса класса;
- реализация членов класса "только для чтения" (метод `set` не определен);
- реализация членов класса "только для записи" (метод `get` не определен);
- проверка корректности присваиваемого значения в соответствии с контекстом использования;
- отсутствие потерь эффективности, поскольку JIT-компилятор старается реализовать свойства посредством встроенных (inline) функций.

Аналогично методам класса, свойства могут объявляться виртуальными (и даже абстрактными) и участвовать в разрешении виртуальных вызовов.

Особый случай: индексирование

Специальным вариантом свойства является индексатор (indexer) (рис. 11.3).

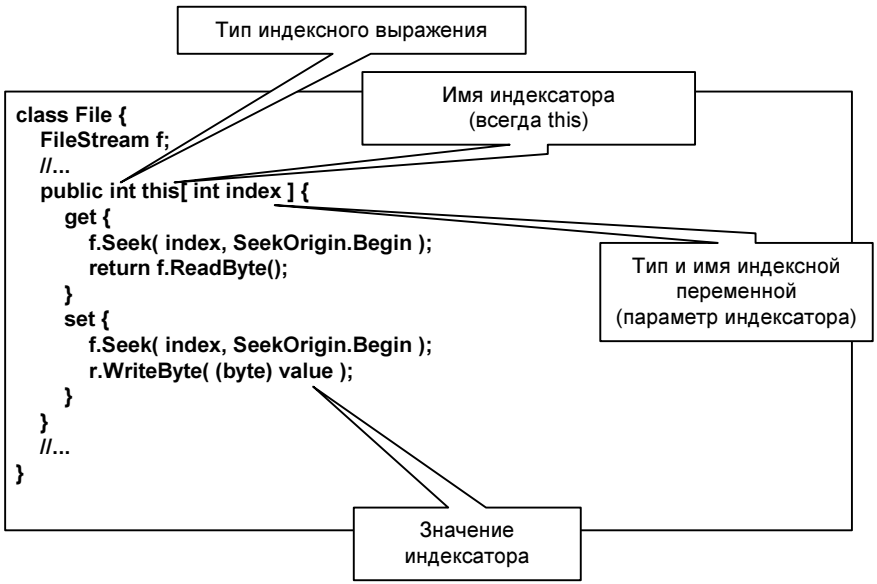


Рис. 11.3. Реализация индексаторов в языке C#

Индексатор воплощает концепцию свойства с параметром и подобен перегруженной операции `[]` в языке C++. Необходимость определить новую семантику индексирования возникает довольно часто при работе с контейнерными классами. Для реализации этого механизма в языке C# предусмотрена возможность такого определения средствами языка. Для приведенного на

рис. 11.3 примера доступа к содержимому файла использование индексатора может выглядеть следующим образом.

```
File buf = new File();
//...
buf[ 0 ] = 'A'; // вызывается f.set( 0, 'A' )
// ...
int x = buf[3]; // вызывается f.get( 3 )
```

Пример, представленный в листинге 11.7, иллюстрирует использование свойства без параметров и свойства с параметром (индексатора).

Листинг 11.7. Свойства с параметром и без параметра

```
using System;
using System.IO;

namespace PropertyTest {
    // Класс, абстрагирующий работу с файлом как с потоком байтов
    // (иллюстративный пример)
    class File {
        FileStream f;
        public File() {
            f = null;
        }

        // Свойство без параметра: имя файла
        public string FileName {
            set {
                // Создается файловый поток
                f = new FileStream( value, FileMode.Create );
            }
            get {
                // Возвращается полное имя созданного файла
                return f.Name;
            }
        }

        // Свойство с параметром (индексатор)
        // index — номер байта в файле
        public int this[ int index ] {
            set {
                // Запись указанного байта в файл
                f.Seek( index, SeekOrigin.Begin );
                f.WriteByte( (byte) value );
            }
```

```
            get {
                // Считывание указанного байта из файла
                f.Seek( index, SeekOrigin.Begin );
                return f.ReadByte();
            }
        }

        // Закрытие файла
        public void Close() {
            if( f != null ) f.Close();
        }
    }

    // Класс приложения
    public class PropertyTestApp {
        static void Main( string[] args ) {
            File buf = new File();

            // Создаем файл, имя которого передано в качестве
            // аргумента командной строки
            buf.FileName = args[ 0 ];

            // Заготовка: массив символов
            char[] arr = new char [] {'A', 'B', 'C' };

            // Печатаем в файл символы из массива arr
            Console.WriteLine( "Writing to file {0}:", buf.FileName );
            int counter = 0;
            foreach( char ch in arr ) {
                Console.WriteLine( "\'{0}\' written", ch );
                buf[ counter ] = ch;
                counter++;
            }

            // Считываем из файла символы
            // и печатаем их на консоль для контроля
            Console.WriteLine( "Reading from file {0}:", buf.FileName );
            for( int byteNo=0; byteNo < counter; byteNo++ ) {
                int next = buf[ byteNo ];
                Console.WriteLine( "\'{0}\' read", (char) next );
            }

            buf.Close();
        }
    }
}
```

Пусть для запуска программы набрана следующая команда:

```
PropertyTest.exe sample.txt
```

Сообщения, печатаемые на консоль в ходе работы приложения, позволяют наблюдать процесс создания на диске файла с именем sample.txt, содержащего символы A, B, C:

```
Writing to file [path]sample.txt:
'A' written
'B' written
'C' written
Reading from file [path]sample.txt:
'A' read
'B' read
'C' read
```

На месте [path] печатается полное имя каталога на диске, в котором создан файл. Обращаем внимание, что типом аргумента индексатора часто является int, однако можно употреблять и другие типы. Разрешается перегрузка индексаторов. Так же как и "обычные" свойства без параметра, индексаторы могут быть виртуальными и, следовательно, замещены в производных классах.

11.3. Функции обратного вызова и делегаты

Примерами стандартных вариантов использования функций обратного вызова могут служить следующие задачи:

- элемент обработчика ошибочных ситуаций, позволяющий учесть при выполнении функции модуля обработки ошибок контекст модуля, инициировавшего вызов обработчика ошибок *(см. гл. 6)*;
- определение семантики сравнения элементов при использовании обобщенных алгоритмов сортировки и поиска *(см. гл. 9)*;
- установление взаимосвязи между функциями WinAPI и клиентским кодом при проектировании Windows-приложений (например, оконные процедуры) и проектирование классов, взаимодействующих друг с другом на основе модели программирования, ориентированной на события (см., например, [Microsoft, 1999], [Лекарев, Давыдов, 2002]).

В C++ функция обратного вызова реализуется за счет косвенного вызова функции посредством использования указателя на функцию. Основной недостаток такого подхода заключается в том, что указатель несет информацию только об адресе функции, а информация о списке параметров функции и типе возвращаемого значения функции доступна только на уровне исходного кода.

При программировании с использованием управляемого кода такой подход неприемлем, поскольку не обеспечивает безопасности типов, иными словами, не позволяет исполнительной системе осуществить контроль соответствия косвенно вызываемой функции сигнатуре той функции, которая подразумевается в точке вызова.

Понятие делегата и реализация в языке C#

Для преодоления этой проблемы, а также для достижения некоторой дополнительной функциональности, разработчики платформы Microsoft .NET ввели понятие делегата, или функции-делегата (delegate).

Использование делегатов иллюстрирует программа на C#, представленная в листинге 11.8.

Листинг 11.8. Делегаты (простой пример на C#)

```
using System;
using System.Windows.Forms; // Требуется для MessageBox

namespace DelegateTest {

// Обработка сообщений
class MessageManager {

    // Делегат: сигнатура метода для отправки сообщения:
    //          должен иметь два параметра типа string;
    //          не имеет возвращаемого значения
    public delegate void MessageFeedback(
        string sender, // Отправитель
        string message // Сообщение
   );

    // Главный метод обработчика:
    // вызывает функцию, передаваемую посредством
    // делегата
    public void ProceedMessage(
        MessageFeedback feedback, // Делегат для функции обратного
                                  // вызова, используемой
                                  // для отправки сообщения

        string sender,  // Отправитель
        string message  // Сообщение
   ) {
```

```
        if( feedback != null )
            // Если делегат определен...
            //... вызвать функцию обратного вызова
            feedback( sender, message );
    }
} // Конец класса MessageManager

// Класс, определяющий и использующий функции,
// связываемые с делегатом
public class DelegateApp {

    // Отправить сообщение на консоль
    static void SendMsgToConsole( string sender, string message ) {
        Console.Write( "Message from " + sender + ": " );
        Console.WriteLine( message );
    }

    // Отправить сообщение в окно сообщений (MessageBox)
    static void SendMsgToMsgBox( string sender, string message ) {
        MessageBox.Show( "Message from " + sender + ": " + message );
    }

    // Основная программа
    static void Main() {

        // Создаем объект-обработчик сообщений
        MessageManager msgMan = new MessageManager();

        // Вызываем обработчик:
        // передаем в качестве делегата создаваемый объект
        // типа MessageManager.MessageFeedback с
        // заданием метода SendMsgToConosole() в качестве
        // функции обратного вызова
        msgMan.ProceedMessage(
            new MessageManager.MessageFeedback( SendMsgToConsole ),
            "Eugene", "Hello!"
       );

        // Снова вызываем обработчик:
        // теперь задаем метод SendMsgToMsgBox()
        // в качестве функции обратного вызова
        msgMan.ProceedMessage(
            new MessageManager.MessageFeedback( SendMsgToMsgBox ),
            "Eugene", "Bye!"
       );
    }
```

```
} // Конец класса DelegateApp
} // Конец пространства имен DelegateTest
```

В классе `MessageManager` определен тип `MessageFeedback`, определяющий сигнатуру функции, которая может быть вызвана посредством обратного вызова. Использование делегата такого типа иллюстрирует метод `ProceedMessage()`, получающий экземпляр делегата в качестве аргумента. Как явствует из функции `Main()`, параметр `feedback` метода `ProceedMessage()` связывается в первом случае с функцией `SendMsgTpConsole()`, осуществляющей печать заданного сообщения на консоль, а во втором случае — с функцией `SendMsgToMsgBox()`, осуществляющей печать сообщения в окне сообщений (рис. 11.4).

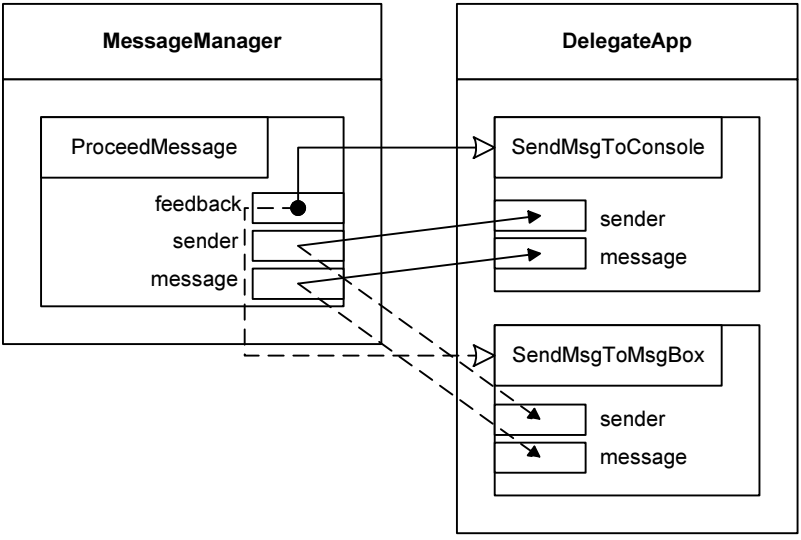


Рис. 11.4. Использование делегатов для реализации механизма обратного вызова функции

В результате работы данной программы сначала в консольное окно будет выведено сообщение:

```
Message from Eugene: Hello!
```

Затем появится окно сообщений, содержащее текст: "Message from Eugene: Bye!".

В ряде случаев посредством обратного вызова может потребоваться вызвать последовательно несколько функций, имеющих одинаковую сигнатуру (рис. 11.5).

Библиотека классов платформы .NET Framework предоставляет класс `Delegate`, обеспечивающий возможности манипулирования цепочками связанных спи-

сков объектов делегатов *(вспомните упражнение 10 из разд. 9.6)*, а язык C# облекает эти возможности в форму специального синтаксиса (листинг 11.9).

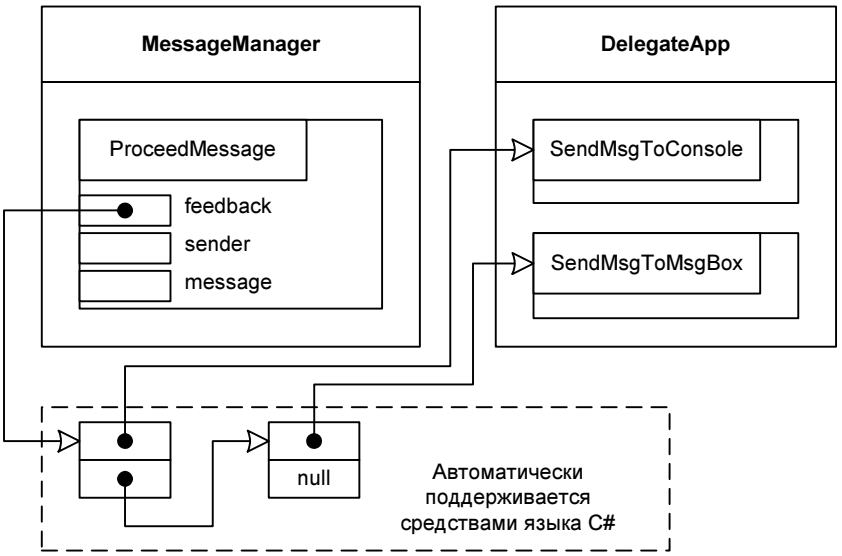


Рис. 11.5. Цепочки делегатов

Листинг 11.9. Цепочка делегатов

```
// Создаем объект-обработчик сообщений
MessageManager msgMan2 = new MessageManager();

// Создаем первый делегат (голова цепочки)
MessageManager.MessageFeedback fbSend =
    new MessageManager.MessageFeedback( SendMsgToConsole );

// Присоединяем еще один делегат
// (теперь цепочка состоит из двух делегатов)
fbSend += new MessageManager.MessageFeedback( SendMsgToMsgBox );

// В ходе обработки сообщений последовательно
// будут вызваны методы
// SendMessageToConsole() и
// SendMessageToMsgBox()
// NB! Необходимый код для последовательной
//     обработки цепочки делегатов генерируется
//     компилятором автоматически
```

```
msgMan2.ProceedMessage(
    fbSend,           // Цепочка делегатов
    "Eugene",         // Отправитель
    "Chain of hello"  // Сообщение
);
```

В данном разделе приведены начальные сведения о делегатах, необходимые для понимания основных принципов работы механизма функций обратного вызова на базе делегатов. За рамками данного рассмотрения остались такие элементы программирования с использованием делегатов, как применение делегатов для вызова экземплярных (нестатических) методов, сравнение делегатов, управление цепочками делегатов, вызов делегатов, точная сигнатура которых на момент компиляции неизвестна, и т. д. Наиболее детально делегаты рассматриваются в книге [Richter, 2002], а также в серии статей Рихтера, написанных им для MSDN Magazine.

11.4. Программирование, ориентированное на события

Механизм управления исполнением кода, ориентированный на события, — это механизм, обеспечивающий взаимные уведомления программных объектов и привязку этих уведомлений к функциям-обработчикам. Большинство программ, обеспечивающих реализацию элементов графического пользовательского интерфейса, основаны на использовании модели программирования, ориентированного на события. В частности, разработка приложений Windows также основана на модели управления действиями, ориентированной на события. Взаимодействие операционной системы, приложения и его компонентов, а также поддержка взаимодействия пользователя и программы осуществляется посредством использования сообщений, каждое из которых является абстракцией события, произошедшего в ходе выполнения кода. Это могут быть сообщения, связанные с действиями пользователя (например, выбор пункта меню, нажатие определенной кнопки и т. д.), и системные сообщения (создание и уничтожение окон, события клавиатуры и мыши и т. п.)

В рамках модели, ориентированной на события, основной задачей программы является обработка получаемых сообщений. Для обеспечения корректной обработки нужно осуществить привязку каждого обрабатываемого сообщения к соответствующей функции. Элементы реализации семантики обработки сообщений связаны с архитектурой приложения. Рассмотрим пример организации обработки сообщений в Windows.

Обработка сообщений приложением Windows

Приложения Windows состоят как минимум из одного процесса (process). Ему выделяется адресное пространство и ресурсы. В рамках одного процесса может быть запущено несколько потоков. Поток (thread) — это минимальная программная единица, для выполнения которой выделяется процессорное время. Процесс имеет как минимум один исполняемый поток, называемый первичным потоком. Для выполнения фоновых задач в многозадачной среде можно создавать вторичные потоки. Приложения, инициирующие несколько потоков, называют многопоточными.

При запуске приложения операционная система создает процесс и начинает выполнение его первичного потока. Информация о первичном потоке передается операционной системе в виде адреса функции. Поэтому все приложения Windows содержат функцию, соответствующую точке входа в приложение. В приложениях Windows, разрабатываемых на C++, точкой входа в приложение является функция `WinMain()`, адрес которой и передается операционной системе в качестве адреса первичного потока.

После выполнения ряда подготовительных действий создается главное окно приложения. Прежде чем окно будет отображено на экране, в операционной системе должны быть зарегистрированы оконные классы — шаблоны, содержащие информацию о свойствах окна (таких как стили окна, форма курсора мыши, значок приложения, цвет фона, имя оконного класса и т. д.). Любое окно Windows принадлежит одному из существующих (зарегистрированных) на данный момент в системе классов, который должен быть определен до того, как окно будет отображено на экране (разумеется, не каждое созданное окно обязательно является отображаемым).

Основной задачей приложения Windows является обработка полученных сообщений. Каждый поток, создающий окно, имеет очередь сообщений — специальная структура данных, в которой операционная система хранит сообщения. Поэтому иногда говорят, что у приложения Windows есть главное окно, имеющее цикл обработки сообщений (с точки зрения кода на C++ цикл обработки сообщений, конечно, содержится внутри функции `WinMain()`). Задача цикла обработки сообщений заключается в извлечении сообщений из очереди сообщений приложения и пересылки их для обработки в соответствующую оконную процедуру. Оконная процедура (являющаяся, по существу, функцией обратного вызова) регистрируется при создании оконного класса (рис. 11.6).

Приложения Windows, спроектированные на основе описанных принципов, основываются на использовании библиотеки функций, предоставляющих низкоуровневый интерфейс для взаимодействия с операционной системой

(в 32-разрядных системах данный интерфейс называют Win32 API — Application Programming Interface, интерфейс прикладного программирования). Win32 API — это не что иное, как комплект функций, структур, сообщений и макросов, с помощью которых можно создавать программы для 32-разрядной платформы Windows. Таким образом, Win32-приложения, использующие Win32 API на уровне архитектуры приложения и реализации основных компонентов, построены на основе процедурной модели программирования.

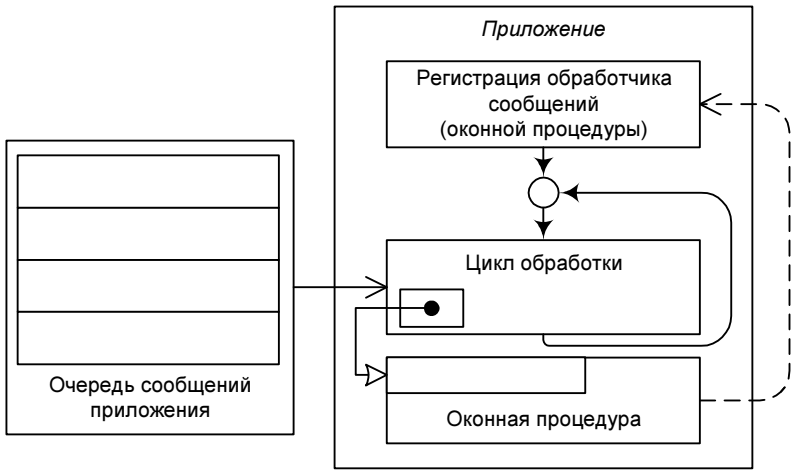


Рис. 11.6. Обработка событий приложением Windows, использующем библиотеку WinAPI

Объектно-ориентированная архитектура приложения

Разработка сложных многокомпонентных приложений Windows на основе использования процедурной парадигмы является довольно трудоемкой и громоздкой задачей. Основной подход, используемый архитекторами программного обеспечения для упрощения и ускорения разработки сложных приложений, заключается в инкапсуляции функций низкоуровневого интерфейса классами, образующими каркас объектно-ориентированной модели приложения. Примерами таких реализаций могут служить библиотека MFC (Microsoft Foundation Classes) от Microsoft и библиотека VCL (Visual Component Library) от Borland.

В основе библиотеки MFC — набор классов C++, инкапсулирующих основные функции Win32 API. Если API является интерфейсом прикладного программирования, то о классах MFC часто говорят, что они формируют среду проектирования приложений (application framework).

Один из основных механизмов объектно-ориентированного программирования, обеспечивающих повторное использование кода, а также отделение интерфейса от реализации, является механизм наследования. Поэтому классы C++, входящие в MFC, организованы в виде иерархии наследования. Наглядное представление иерархии классов MFC можно посмотреть в MSDN, введя в индексной строке запрос "hierarchy chart". На вершине иерархии находится класс `CObject`. Таким образом, библиотека MFC воплощает концепцию однокоренной иерархии классов, используемых для создания базовой архитектуры приложения Windows средствами языка C++. Классу `CObject` непосредственно наследует класс `CCmdTarget`, обеспечивающий представление базовых операций для объектов, способных обрабатывать сообщения Windows.

В основе реализации механизма привязки сообщений (событий) к их обработчикам (реакциям на события) в MFC лежат так называемые карты сообщений (рис. 11.7).

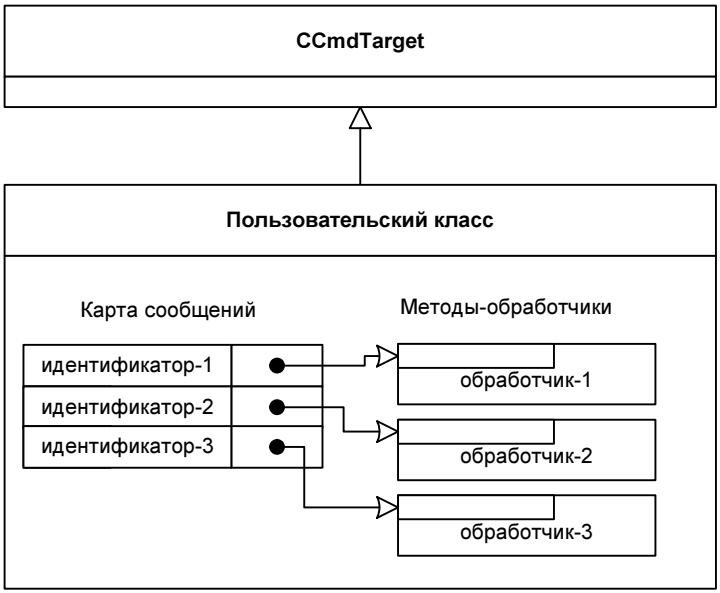


Рис. 11.7. Привязка событий к методам обработчиков в приложении, использующем MFC

В соответствии с [Microsoft, 1999], карта сообщений определяется как объявленная в теле класса таблица, связывающая системные сообщения с функциями-членами данного класса. Она состоит из записей, назначающих функцию-обработчик для каждого идентификатора сообщения. С точки зрения программирования на C++ карта сообщений — не что иное, как набор макросов, заменяющих обращения к средствам, определяемым классом `CCmdTarget`.

Таким образом, для поддержки модели управления действиями, ориентированной на события, разработчики MFC реализовали набор макроопределений, определяющих своего рода метаязык описания событий и их обработчиков (кстати, при создании карт сообщений MFC программистам настоятельно рекомендуется использовать мастера класса или приложения и не редактировать карты вручную). Основной недостаток использования подобного подхода в управляемых средах заключается в невозможности обеспечить контроль корректности типов обработчиков, поскольку единственной информацией об обработчике, помещаемой в карту сообщений, является адрес функции-обработчика.

Библиотека VCL также представляет собой иерархию типов, позволяющих проектировать приложения на более высоком уровне абстракции по сравнению с уровнем, обеспечиваемым Win32 API. Основным языком реализации VCL является язык Object Pascal, поэтому обеспечение модели, ориентированной на события, основывается на использовании встроенных средств языка Object Pascal (поддерживающего и события, и свойства напрямую). Никаких дополнительных механизмов, внешних по отношению к классам VCL (типа карт сообщений), при этом не требуется. Таким образом, механизмы, задействованные в библиотеке VCL, в общем и целом сходны с обсуждаемой далее реализацией концепции событий в языке C#.

Поддержка модели программирования, ориентированной на события, средствами языка

Аналогично Object Pascal, C# обеспечивает представление основных элементов модели, ориентированной на события, средствами языка. Рассмотрим архитектуру простого приложения, использующего данный механизм. Предположим, что в рамках развития некоторой системы управления пользовательскими сообщениями (см. листинги 11.8—11.9) класс, отвечающий за пересылку сообщения (`MessageManager`), информирует заинтересованные объекты о том, что сообщение отправлено. При наступлении события "сообщение отправлено" вызывается обработчик, привязываемый к событию при определении "заинтересованного" класса (в нашем случае в качестве примера выступает класс `WatchSending`). Архитектура приложения представлена на рис. 11.8.

Основные этапы проектирования программы, использующей события, иллюстрирует листинг 11.10. При этом в комментариях пронумерованы те элементы определения класса, которые связаны с использованием механизма, ориентированного на события.

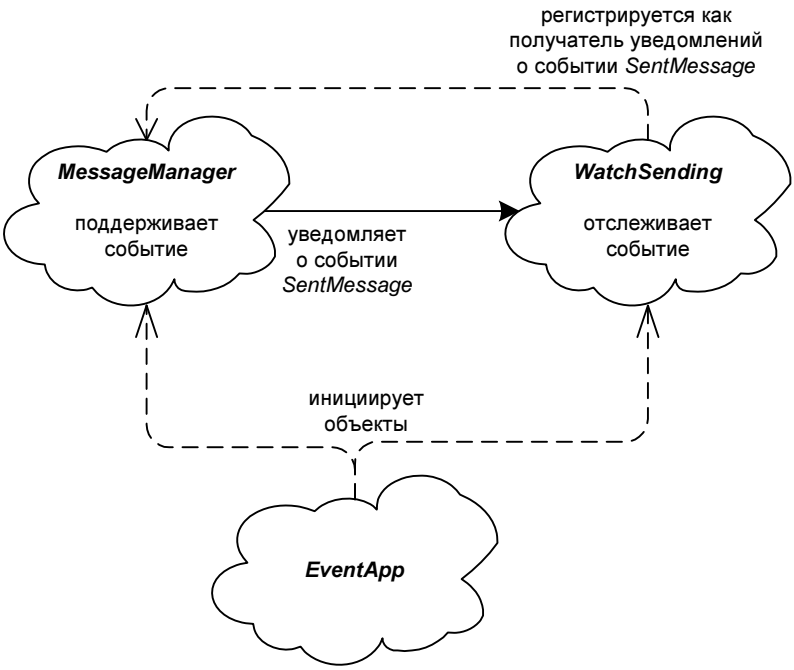


Рис. 11.8. Архитектура простого приложения, использующего обработку событий

Листинг 11.10. Конструирование типов, использующих события

```
using System;
using System.Windows.Forms;

namespace EventTest {
    // Описание события "отправлено сообщение"
    // в соответствии с приемами разработки, рекомендованными
    // Microsoft
    class SentMessageEventArgs : EventArgs { // Имя типа события
        // Сопутствующая информация:
        public readonly string message; // Текст сообщения

        // 1. Конструктор экземпляра события
        public SentMessageEventArgs( string message ) {
            this.message = message;
        }
    }

    // Конструирование типа, поддерживающего событие
    class MessageManager {
```

```
// 2. Делегат, определяющий тип обработчика события
// (соответствующий метод должен быть реализован получателем
//  информации о событии)
public delegate void SentMessageEventHandler (
    object sender, SentMessageEventArgs e
);

// 3. Определение собственно события
// NB! Если некоторый класс содержит член event,
//     это означает, что данный тип обеспечивает
//     извещение других объектов о наступлении события
public event SentMessageEventHandler SentMessage;

// 4. Определение метода, отвечающего за уведомление
// зарегистрированных объектов о наступлении события
// SentMessage
protected virtual void OnSentMessage( SentMessageEventArgs e ) {
    // Если имеются объекты, зарегистрированные для
    // получения уведомлений о событии...
    if( SentMessage != null )
        // ... уведомить их
        // (фактически — вызвать соответствующий обработчик
        //  события)
        SentMessage( this, e );
}

// 5. Определение метода, создающего экземпляр события
// и уведомляющего зарегистрированные объекты
public void NotifySentMessage( string message ) {
    SentMessageEventArgs e =
        new SentMessageEventArgs( message );
    OnSentMessage( e );
}

// Реализация остальной функциональности класса

// Делегат определяет сигнатуру метода, используемого
// для отправления сообщения (см. также листинг 11.8)
public delegate void MessageFeedback(
    string sender, // Отправитель сообщения
    string message // Текст сообщения
);

// Главный метод обработчика:
// вызывает функцию, передаваемую посредством
// делегата
```

```
    public void ProceedMessage(
        MessageFeedback feedback, // Делегат для функции обратного
                                  // вызова, используемой
                                  // для отправки сообщения

        string sender,  // Отправитель
        string message  // Сообщение
    ) {

        if( feedback != null ) {
            feedback( sender, message );

            // 6. Инициирование события SentMessage
            NotifySentMessage( message ); // Послать извещение о
                                          // событии
                                          // зарегистрированным
                                          // объектам
        }
    }
}

// Конструирование типа, отслеживающего событие
class WatchSending {
    // Конструктор.
    // 6. Регистрация для получения уведомлений о событии
    public WatchSending( MessageManager msgMan ) {
        msgMan.SentMessage +=
            new MessageManager.SentMessageEventHandler(
                MessageIsSent );
                // Метод MessageIsSent будет вызван при
                // наступлении события
    }

    // 7. Определение метода, вызываемого при наступлении события
    private void MessageIsSent(
        object sender,
        SentMessageEventArgs e
    ) {
        MessageBox.Show( "The message " + e.message +
                         " has just been sent" );
    }
}

// Тестирующая программа
public class EventApp {
```

```
        // Печать некоторого сообщения на консоль
        static void SendMsgToConsole( string sender, string message ) {
            Console.Write( "Message from " + sender + ": " );
            Console.WriteLine( message );
        }

        // Основная программа
        static void Main() {
            // 8. Создание объекта класса, поддерживающего событие
            MessageManager msgMan = new MessageManager();

            // 9. Создание объекта класса, отслеживающего событие
            WatchSending sw = new WatchSending( msgMan );

           // 10. В ходе работы метода ProceedMessage
           // (определение см. выше) инициируется событие SentMessage
           msgMan.ProceedMessage(
               new MessageManager.MessageFeedback( SendMsgToConsole ),
                                                   "Eugene", "Hello!" );
        }
    }
}
```

На рис. 11.9 иллюстрируется взаимодействие ключевых элементов объектов обозреваемого приложения.

В процессе работы конструктора класса `WatchSending` создается экземпляр делегата `SentMessageEventHandler` и обеспечивается привязка метода `MessageIsSent()` к событию `SentMessage`, определенному в классе `MessageManager`. В терминах более высокоуровневого представления о взаимодействии программных объектов это означает регистрацию объекта класса `WatchSending` как получателя уведомлений о событии `SentMessage`. Фактически, речь идет об определении функции обратного вызова, управление которой передается при возникновении события `MessageIsSent`. В качестве аргументов метод `MessageIsSent()` получает ссылку на объект-источник уведомления (в нашем случае — на объект класса `MessageManager`) и ссылку на объект класса `SentMessageEventArgs`, обеспечивающий представление дополнительной информации, связанной с событием.

В результате работы программы на консоли будет напечатано сообщение

```
Message from Eugene: Hello
```

Сразу после этого на экране появится выводимое функцией `MessageIsSent()` окно сообщений, содержащее текст "The message Hello! has just been sent", что подтверждает вызов метода `MessageIsSent()` при возникновении анализируемого события.

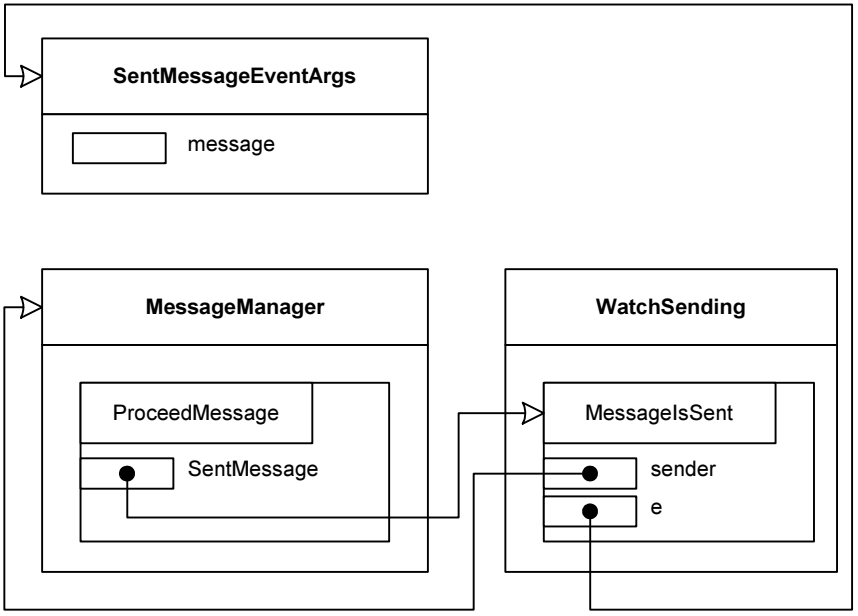


Рис. 11.9. Взаимодействие программных объектов приложения .NET, ориентированного на обработку событий

Примечание

Поддержку на уровне языка абстракций делегатов и событий связывают не только с проектированием приложений, ориентированных на события, но также с поддержкой модели автоматного программирования и паттернов (шаблонов) проектирования, нацеленных на использование этой модели (см., например, [Шамгунов, Корнеев, Шалыто, 2004]).

11.5. Специализированные атрибуты

Одной из новаторских концепций, актуальность которой обусловлена увеличением доли распределенных (distributed) программных систем, является концепция специализированных атрибутов.

Основная идея специализированных атрибутов заключается не только в возможности привязки к элементам метаданных некоторой дополнительной информации, но и в предоставлении механизма для обращения к этой информации в ходе выполнения программы и, следовательно, для динамического изменения поведения программы на основе анализа этой дополнительной информации. Иными словами, атрибуты позволяют реализовать идею обращения к информации времени разработки в ходе выполнения управляемого

кода. При этом процесс обращения к этой информации в ходе работы программы называется рефлексией, или отражением (reflection).

Применение атрибутов, используемых компилятором

Несмотря на то, что здесь, как и в большинстве разделов этой главы, в качестве средства иллюстрации используется язык C#, мы, следуя духу названия книги, постараемся абстрагироваться от описания конкретных реализаций идеи атрибутов, а сосредоточимся на самой концепции специализированных атрибутов. Итак, в первую очередь, специализированные атрибуты предоставляют способ связать некоторую дополнительную информацию с тем или иным элементом программы или данных. Классическим примером, иллюстрирующим полезность специализированных атрибутов, является пример развития некоторой библиотеки, в ходе которого ряд средств, предоставляемых библиотекой, устаревает. Как сообщить о том, что некоторые компоненты библиотеки устарели? Разумеется, можно включить соответствующие данные в поставляемую потребителю документацию, однако это не позволит учесть эту информацию на этапе компиляции программы. А ведь такая возможность была бы очень полезной, поскольку программисту бывает трудно уследить за всеми возможными изменениями в используемой библиотеке.

Проиллюстрируем идею специализированных атрибутов на несложном примере. Представим себе некоторую упрощенную модель. Пусть имеется класс, обеспечивающий воспроизведение некоторой музыкальной информации:

```
namespace Broadcasting {
    // Библиотека. Версия 1. Воспроизведение музыки
    public class Play {
        //...
        public void PlayMusic() {
            // Playing music...
        }
    }
}
```

Пусть также имеется некоторая клиентская программа, использующая библиотечный код:

```
using Broadcasting;
namespace Client {
    // Клиент. Версия 1.
    class ClientMain {
        static void Main() {
```

```
            Play play = new Play();
            play.PlayMusic();
        }
    }
}
```

Предположим, что в связи с развитием форм представления мультимедийных данных разработчики библиотеки решают модифицировать библиотечный класс, реализовав поддержку различных форматов мультимедиа (листинг 11.11).

Листинг 11.11. Использование атрибута `Obsolete`

```
// Библиотека PlayLibrary
using System;
namespace Broadcasting {
    // Библиотека. Версия 2. Воспроизведения мультимедиа
    public class Play {
        // Типы мультимедийных форматов
        public enum MediaType { music, video, dvd, mpeg }
        // ...
        public void PlayMedia( MediaType mt ) {
            // Playing media...
        }

        // Определению метода PlayMusic() предшествует атрибут
        // Obsolete, указывающий на то, что этот метод устарел
        [Obsolete("This method is deprecated, use PlayMedia() instead",
        false)]
        public void PlayMusic() {
            // Old fashioned playing music...
        }
    }
}
```

Посредством использования специализированного атрибута `Obsolete` разработчики не просто снабдили объявление метода класса дополнительной информацией (для этих целей подошел бы и комментарий), но обеспечили дополнительной информацией компилятор. Как видно из предложенного фрагмента, специализированный атрибут может иметь параметры. В данном случае их два. Первый параметр определяет строку, которая будет воспроизведена компилятором при трансляции программы, использующей устаревшие элементы библиотеки. Второй параметр здесь имеет булевский тип. В соответствии с определением атрибута `Obsolete` в библиотеке классов платформы

.NET Framework — Framework Class Library (FCL), если второй параметр атрибута `Obsolete` имеет значение `false`, то компилятор будет выдавать предупреждение (warning) об использовании устаревшего метода. Если же второй параметр атрибута `Obsolete` имеет значение `true`, то компилятор будет выдавать сообщение об ошибке (error). Итак, атрибут `Obsolete` является примером атрибута, используемого компилятором.

Например, при компиляции примера программы, представленной в листинге 11.12 и использующей библиотечный класс `Play` (см. листинг 11.11), компилятор выдаст предупреждение к строке, содержащей обращение к методу `PlayMusic()`.

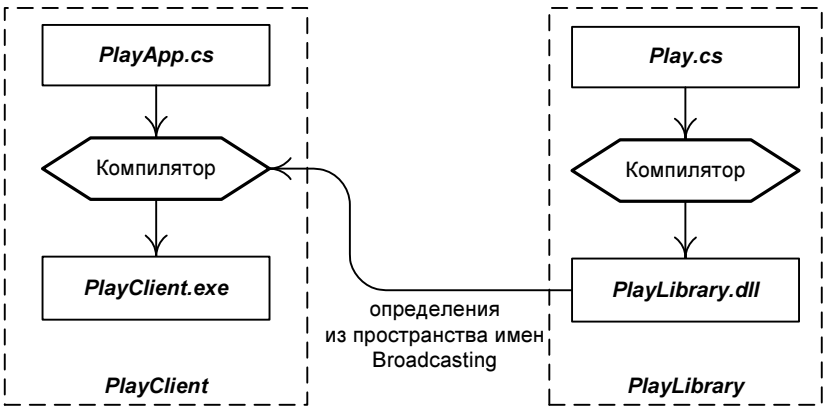


Рис. 11.10. Взаимодействие модулей при компиляции приложения PlayClient

Листинг 11.12. Влияние установленного атрибута `Obsolete` на компиляцию клиентского кода

```
// Приложение PlayClient
using Broadcasting;
namespace Client {
    // Клиент. Версия 2.
    class PlayApp {
        static void Main(){
            Play play = new Play();

            play.PlayMusic(); // Здесь будет предупреждение
                              // компилятора:
                              // This method is deprecated,
                              // use PlayMedia() instead
```

```
            play.PlayMedia( Play.MediaType.music );
        }
    }
}
```

Диаграмма взаимодействия модулей приложения, использующего класс `Play`, представлена на рис. 11.10.

Читатель может самостоятельно установить значение `true` для второго параметра атрибута `Obsolete` и убедиться в том, что в этом случае клиентский код не компилируется: в точке вызова метода `PlayMusic()` выдается сообщение об ошибке.

Использование атрибутов в период выполнения

Второй и, видимо, главной возможностью использования атрибутов, является предоставление механизма для обращения к дополнительной информации, определяемой атрибутом, в ходе выполнения программы. Доступ к этой информации обеспечивается благодаря так называемой технологии отражения. Отражение — это механизм, позволяющий программе изменять свое поведение на основе анализа особенностей другого кода и его метаданных, и, как результат, позволяющий создавать динамически расширяемые приложения вроде тех, что похожи на интегрированные среды проектирования.

Здесь мы ограничимся простой иллюстрацией действия механизма отражения для анализа в ходе выполнения программы информации, определяемой специализированным атрибутом. Для этого усложним задачу клиентского компонента, рассмотренного ранее (см. листинг 11.12). Попробуем проанализировать значение атрибута, относящегося к методу библиотечного класса `Play()` "на лету", изменяя в результате анализа поведение клиентского кода.

Для наблюдаемости процесса обращения к методам библиотечного класса снабдим эти методы печатью информационных сообщений так, как показано в листинге 11.13.

Листинг 11.13. Библиотечный класс с печатью информационных сообщений

```
// Библиотека PlayLibraryWithMessages
using System;
namespace Broadcasting {
    // Библиотека. Версия 3.
    public class Play {
        // Типы мультимедийных форматов
        public enum MediaType { music, video, dvd, mpeg }
```

```
        //...
        public void PlayMedia( MediaType mt ) {
            Console.WriteLine( "Playing media: " + mt.ToString() );
        }

        // Определению метода PlayMusic() предшествует атрибут
        // Obsolete, указывающий на то, что этот метод устарел
        [Obsolete("This method is deprecated, use PlayMedia() instead",
        false)]
        public void PlayMusic() {
            Console.WriteLine( "Playing music" );
        }
    }
}
```

Пример клиентской программы в листинге 11.14 иллюстрирует извлечение информации из метаданных библиотечного модуля и реализацию альтернативных ветвей исполнения программы в случаях установленного и неустановленного атрибута `Obsolete` у метода `PlayMusic()`.

Листинг 11.14. Анализ атрибута в ходе выполнения клиентского кода

```
// Приложение PlayClientReflection
using System;
using System.Reflection;
using Broadcasting;
namespace Client {
    // Клиент. Версия 3.
    class PlayApp {
        static void Main() {
            Play play = new Play();

            // Выявление сущности, к которой относится атрибут
            MemberInfo typeinfo;
            typeinfo = typeof( Play ).GetMethod( "PlayMusic" );

            // Попытка извлечь набор атрибутов
            object [] att =
                typeinfo.GetCustomAttributes(
                    typeof( ObsoleteAttribute ), // Тип атрибута
                    false                        // Не анализировать
                                                 // иерархию классов
                );
```

```
            ObsoleteAttribute obsAtt = null; // Экземпляр атрибута
                                             // ObsoleteAttribute
            if( att.Length != 0 ) { // Если атрибуты есть...

                // Извлекаем информацию, определяемую атрибутом,
                // из библиотечного кода
                obsAtt = ( ObsoleteAttribute ) att[ 0 ];
                Console.WriteLine( "Obsolete attribute found." );
                Console.WriteLine( "It contains message: \'{0}\'",
                                   obsAtt.Message );

                // Вызываем PlayMedia()
                Console.WriteLine( "Calling PlayMedia()..." );
                play.PlayMedia( Play.MediaType.music );
            }
            else { // Если атрибутов нет...

                // Вызываем PlayMusic()
                Console.WriteLine( "Calling PlayMusic()..." );
                play.PlayMusic();
            }
        }
    }
}
```

При исполнении клиентской программы, скомпилированной с библиотечным модулем, полученным в результате компиляции исходного кода в листинге 11.13 (атрибут `Obsolete` установлен), выводятся следующие сообщения:

```
Obsolete attribute found.
It contains message: 'This method is deprecated, use PlayMedia() instead'
Calling PlayMedia()...
Playing media: music
```

Если убрать атрибут `Obsolete` из заголовка метода `PlayMusic()` в библиотечном коде, то после перекомпиляции клиентской программы с измененным библиотечным модулем, она будет порождать следующий вывод:

```
Calling PlayMusic()...
Playing music
```

Этот пример не очень полезен в реальной проектной практике, однако позволяет проиллюстрировать общую схему действия механизма отражения в простейшем случае.

Атрибуты, определяемые пользователем

Из листинга 11.14 очевидно, что на уровне кода тип атрибута — это класс, а экземпляр атрибута — объект класса. Таким образом, для того чтобы воспользоваться атрибутом в ходе выполнения, нужно создать экземпляр класса.

Язык программирования, поддерживающий концепцию специализированных атрибутов, обычно предоставляет специальный синтаксис, упрощающий определение атрибута. Например, в C# атрибут, применяемый к элементу программы, заключается в квадратные скобки. Однако концептуально важно понять: для того чтобы определить атрибут, нужно определить класс в некотором пространстве имен (в .NET Framework любой класс атрибута должен быть унаследован от класса FCL `System.Attribute`).

Раз для определения специализированного атрибута нужно определить класс, воплощающий абстракцию атрибута в объектно-ориентированной программе, значит, открыт путь для определения собственных атрибутов, используемых одним или группой программистов в своих проектах. С разрешения автора приведу в качестве примера (листинг 11.15) немного отредактированное определение атрибута, хранящего информацию о разработчике и дате создания некоторого программного компонента (например, класса) [Дубовцев, 2004].

Листинг 11.15. Определение пользовательского атрибута

```
// Автор: Дубовцев Алексей.
// Подключим основное пространство имен общей библиотеки классов.
using System;
namespace MyAttribute {
    // Определяем пользовательский атрибут
    // В начале - его собственный атрибут:
    //     может применяться ко всем программным элементам,
    //     не наследуется (будучи применен к одному классу,
    //                     не считается действующим для
    //                     производных от него)
    [AttributeUsage( AttributeTargets.All, Inherited = false )]
    public sealed class DelevoperAttribute : Attribute {
        // Служебные поля для хранения данных атрибута
        private string m_Name; // Имя разработчика
        private DateTime m_CreateTime; // Дата создания

        // Конструктор атрибута по умолчанию
        public DelevoperAttribute( string name,
                                   int year, int month, int day ) {
```

```
            m_Name = name;
            m_CreateTime = new DateTime( year, month, day );
        }

        // Общедоступные свойства, позволяющие получить доступ
        // к данным атрибута
        public string Name {
            get {
                return m_Name;
            }
        }
        public DateTime CreateTime {
            get {
                return m_CreateTime;
            }
            set {
                 m_CreateTime = value;
            }
        }
    }
}
```

Листинг 11.16 иллюстрирует применение нового атрибута в клиентском коде.

Листинг 11.16. Применение и запрос атрибута в клиентском коде

```
using System;
using System.Reflection;
using MyAttribute;
// Применим атрибут к основному классу приложения.
namespace ClientNamespace {
    [Delevoper("Алексей Дубовцев", 2003,3,3)]
    class App {
        public static void Main() {
            // Запросим пользовательский атрибут типа DelevoperAttribute
            // от программного элемента типа SomeClass
            DelevoperAttribute da =
               (DelevoperAttribute) Attribute.GetCustomAttribute (
                                        typeof( App ),
                                        typeof( DelevoperAttribute ) );
            // Проверим, был ли связан, необходимый нам атрибут,
            // с указанным программным элементом
            if( da != null ) {
```

```
                // Запрос атрибута удался, можно приступать
                // к получению информации из него
                Console.WriteLine( "Deleveper Name = {0}", da.Name );
                Console.WriteLine( "Create time = {0}",
                                   da.CreateTime.ToString() );
            }
        }
    }
}
```

В результате работы программы на консоль будут выведены следующие строки.

```
Deleveper Name = Алексей Дубовцев
Create time = 03.03.2003 0:00:00
```

Процесс извлечения информации, задаваемой атрибутом, из метаданных модуля иллюстрирует рис. 11.11.

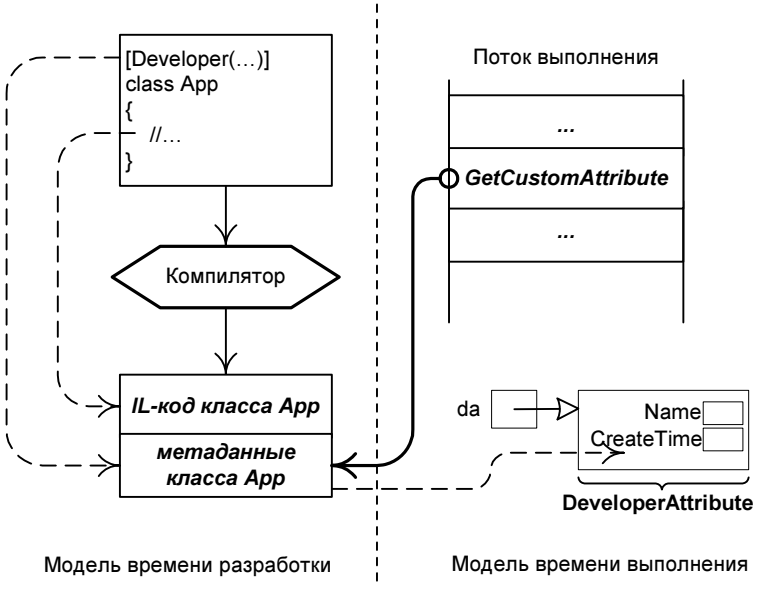


Рис. 11.11. Атрибуты как средство доступа к информации времени разработки в процессе выполнения программы

11.6. Отражение (рефлексия)

В *разд. 11.5 мы рассмотрели* один из случаев действия механизмов, обеспечиваемых технологией отражения — извлечение из метаданных модуля информации, определяемой специализированными атрибутами, а также использование этой информации в ходе исполнения программы. Итак, отражение — это, в первую очередь, процесс извлечения информации из метаданных модуля. Отражение предоставляет комплекс механизмов, позволяющих изменять поведение приложения, используя типы, которые еще не были известны на момент построения исполняемого модуля, и другие сведения, извлекаемые из метаданных этого или других приложений.

Одним из основных элементов любого приложения, использующего отражения, является создание экземпляров типов, определенных в другом управляемом модуле, и обращение к членам этих типов. Однако прежде чем использовать функциональность, предоставляемую типом, нужно извлечь из метаданных модуля информацию о самом типе.

Извлечение информации о типе

Пусть имеется некоторый управляемый модуль, содержащий код, полученный в результате компиляции исходного текста, представленного в листинге 11.17.

Листинг 11.17. Модуль, содержащий определение тестового класса

```
using System;

namespace TestLib {
    public class SomeClass {
        public SomeClass() {
            Console.WriteLine( "SomeClass constructor" );
        }

        public SomeClass( Int32 arg ) {
            Console.WriteLine( "SomeClass constructor: {0}", arg );
        }

        public void Method1() {
            Console.WriteLine( "Method1 called" );
        }

        public int Method2( string arg ) {
            Console.WriteLine( "Method2 called with arg \"{0}\"", arg );
```

```
            return arg.Length;
        }
    }
}
```

Анализ действий, выполняемых методами класса `SomeClass`, нас сейчас не должен очень интересовать. Важно в первую очередь разобраться со структурой управляемого модуля. В данном случае модуль `TestLib` содержит определение единственного класса `SomeClass`, имеющего два перегруженных конструктора и два метода с различными сигнатурами.

Здесь мы абстрагируемся от вопросов идентификации расположения модуля и рассмотрим упрощенный пример, когда пространство имен модуля явно подключено к приложению, обращающемуся к типу `SomeClass` посредством отражения. Разумеется, в этом случае функциональность класса `SomeClass` может быть использована без всякого отражения, поскольку класс доступен во время компиляции. Мы же просто представим, что это не так. Такое допущение приходится сделать, поскольку вопросы доступа к произвольному, обособленному от "отражающего" приложения, модулю неизбежно приведут к необходимости рассмотрения деталей реализации взаимодействия модулей в платформе .NET, что представляет собой довольно обширный материал (некоторые аспекты организации взаимодействия компонентов .NET *рассматриваются в разд. 12.5*). Наша задача заключается в описании самого механизма отражения, поэтому сделанное допущение представляется вполне оправданным.

Листинг 11.18 иллюстрирует процесс извлечения информации о типе из метаданных.

Листинг 11.18. Извлечение информации о типе ("отражающее" приложение)

```
using System;
using System.Reflection;
using TestLib;

namespace ReflectType {
    class ClassMain {
        static void Main() {
            Type t = typeof( SomeClass );

            Console.WriteLine( "Class: {0}", t.FullName );
            Console.WriteLine( "Members:" );
            foreach( MemberInfo mi in t.GetMembers() ) {
                Console.WriteLine( "{0}: {1}", mi.MemberType, mi );
```

```
            }
        }
    }
}
```

В результате исполнения программы будет напечатан следующий текст:

```
Class: TestLib.SomeClass
Members:
Method: Int32 GetHashCode()
Method: Boolean Equals(System.Object)
Method: System.String ToString()
Method: Void Method1()
Method: Int32 Method2(System.String)
Method: System.Type GetType()
Constructor: Void .ctor()
Constructor: Void .ctor(Int32)
```

Читатель может заметить, что часть методов класса `SomeClass` не определена нами явно в модуле `TestLib`. Речь идет о методах, унаследованных от корневого класса иерархии классов FCL `System.Object`.

Заметим, что вместо инструкции `Type t = typeof( SomeClass )` в реалистичной ситуации должна быть реализована последовательность действий по динамической загрузке управляемого модуля и извлечению из него информации об интересующем нас типе.

Создание экземпляров типов

В предположении, что тип известен (создан объект класса `Type`), рассмотрим процесс создания экземпляра типа, информация о котором извлечена из метаданных внешнего модуля.

Создание экземпляра класса обеспечивается вызовом конструктора класса. Таким образом, чтобы создать экземпляр типа, определенного в другом управляемом модуле, нужно вызвать конструктор этого типа. Как известно, определение класса может содержать несколько конструкторов, различающихся списками параметров. Следовательно, общая процедура создания экземпляра "отраженного" типа содержит следующие шаги:

1. Определить, какой из перегруженных конструкторов должен быть вызван.
2. Сформировать аргументы конструктора.
3. Вызвать конструктор, передав сформированный набор аргументов.

Программа, представленная в листинге 11.19, иллюстрирует создание объекта типа `SomeClass` в предположении, что сам тип не доступен в процессе компиляции *(см. информацию о сделанном допущении в предыдущем разделе)*.

Листинг 11.19. Создание экземпляра "отраженного" типа

```
using System;
using System.Reflection;
using TestLib;

namespace CreateInstance {
    class ClassMain {
        static void Main() {
            Type t = typeof( SomeClass );

            ConstructorInfo ctor1 = t.GetConstructor( Type.EmptyTypes );
            object obj1 = ctor1.Invoke( null );

            ConstructorInfo ctor2 =
                t.GetConstructor(
                    new Type[] { Type.GetType( "System.Int32" ) } );
            object[] argsCtor2 = new Object[] { 100 };
            object obj = ctor2.Invoke( argsCtor2 );
        }
    }
}
```

Сообщения, печатаемые конструкторами класса `SomeClass`, подтверждают создание двух экземпляров "отраженного" типа.

```
SomeClass constructor
SomeClass constructor: 100
```

Отражение методов класса

Теперь рассмотрим обращение к методам "отраженного" типа. В *главе 3 было показано*, что при обращении к нестатическому методу класса этот метод получает неявный аргумент — адрес объекта. Отсюда и выражение "вызывать метод на объекте". Когда же речь идет о вызове метода на объекте, сконструированном посредством отражения, адрес этого объекта следует передать явно (листинг 11.20).

Листинг 11.20. Вызов методов на экземпляре "отраженного" типа

```
using System;
using System.Reflection;
using TestLib;
```

```
namespace InvokeMethods {
    class ClassMain {
        static void Main() {
            Type t = typeof( SomeClass );

            ConstructorInfo ctor = t.GetConstructor( Type.EmptyTypes );
            object obj = ctor.Invoke( null );

            MethodInfo m1i = obj.GetType().GetMethod( "Method1" );
            m1i.Invoke( obj, null );

            MethodInfo m2i = obj.GetType().GetMethod( "Method2" );
            object[] argsM2 = new Object[] { "test" };
            int x = (int) m2i.Invoke( obj, argsM2 );
            Console.WriteLine( "Method2 returned {0}", x );
        }
    }
}
```

Сообщения, печатаемые программой, позволяют убедиться в том, что после создания объекта последовательно вызываются два метода класса `SomeClass`:

```
SomeClass constructor
Method1 called
Method2 called with arg "test"
Method2 returned 4
```

11.7. Интеграция кода и документации

Разработка, поддержка и распространение кода требуют тщательного документирования содержания проектируемого кода. При этом цель документирования исходных текстов заключается в обеспечении правильного и по возможности более легкого восприятия исходных текстов читателем. Достижение этой цели обеспечивается ясными комментариями. Любой язык, предназначенный для проектирования, имеет встроенные средства комментирования исходного кода.

Однако потребитель кода (например, пользователь библиотеки или программного компонента) не обязательно имеет доступ к исходным текстам. Даже если этот доступ существует, анализ исходных текстов вряд ли является самым быстрым и комфортным способом изучения функциональности, обеспечиваемой библиотечным кодом или кодом программного компонента. В связи с этим возникает необходимость сопровождения кода комплектами документации, содержащими подробную информацию о том, какие возможности предоставляет код и как правильно воспользоваться этими возможно-

стями. Поддержка документации является довольно сложной задачей, требующей самостоятельных проектных усилий. Если код и его документация существуют раздельно, то, оказывается, довольно сложно обеспечить точное постоянное соответствие документации и текущей версии библиотеки или компонента. Эта сложность образует первую проблему документирования программного обеспечения.

Вторая проблема заключается в обеспечении по возможности единообразного представления документации разными группами разработчиков. Этот аспект очень важен в свете увеличения доли кода, проектируемого с использованием компонентов, созданных другими разработчиками.

Решению этих проблем способствует интеграция кода и документации, которая представляет собой механизм реализации модели информационного обмена между разработчиками, обеспечивающей не только построение целостной и непротиворечивой документации, но и автоматизацию модификации существующей документации в процессе внесения изменений в проектируемые программы [Пышкин, 2004-2].

Документирующие комментарии: модель Java

В Java интеграция кода и документации реализована следующим образом. Комментарий, начинающийся с последовательности `/**`, считается документирующим комментарием. Заканчивается он так же, как и обычный многострочный комментарий, комбинацией `*/`. Таким образом, для компилятора нет разницы между обычными и документирующими комментариями. Однако появляется возможность реализовать отдельный инструмент для извлечения комментариев из кода. В рамках платформы Java таким инструментом является утилита javadoc. Эта утилита не только извлекает документирующие комментарии, но также использует некоторые возможности компилятора Java для распознавания имен классов, методов и других членов класса, к которым относятся документирующие комментарии. Результатом работы javadoc является набор HTML-файлов, формат которых соответствует формату стандартной документации Java от Sun Microsystems. Это позволяет разным разработчикам создавать документацию, представленную в одном стиле и организованную на основе одинаковой структуры представления информации.

Внутри документирующих комментариев можно использовать специальные теги (tags), начинающиеся с символа `@` и обязательно размещенные в отдельных строках комментария. Применение документирующих тегов иллюстрирует фрагмент определения класса Java:

```
package Triangle;

import Point.*;
```

```
/**
 *  Plain triangle construction and processing class
 *
 *  @version 1.1
 *  @author  Evgueni V. Pychkine, 2001
 *  @author  http://aivt.ftk.spbstu.ru
 *
 *  @see Point
 */
public class Triangle {
    //...

    /**
     *  Triangle construction by using three points
     *  @param p1 First point reference
     *  @param p2 Second point reference
     *  @param p3 Third point reference
     *  @return Returns true, if successful
     *  @return Otherwise returns false
     *  @throws Exceptions Throws no exception
     */
    public boolean ConstructTriangle( Point p1,
                                      Point p2,
                                      Point p3 ) {
        // ...
    }
}
```

Поскольку результатом работы javadoc является HTML-файл, внутри документирующих комментариев можно помещать HTML-теги (это называется внедренным HTML), управляя таким образом представлением результирующего HTML-кода:

```
package Triangle;

import Point.*;

/**
 *  <b>Plain triangle construction and processing class</b>
 *
 *  @version 1.1
 *  @author  Evgueni V. Pychkine, 2001
 *  @author  <br><a href="http://aivt.ftk.spbstu.ru" Web</a>
 *
 *  @see Point
 */
```

```
public class Triangle {
    // ...
}
```

Описание тегов документирующих комментариев с примерами их использования см., например, в [Naughton, Schildt, 1999].

Документирующие комментарии: модель C#

Разработчики C# также реализовали возможность использования встроенной в код документации. Признаком начала однострочного документирующего комментария является последовательность ///.

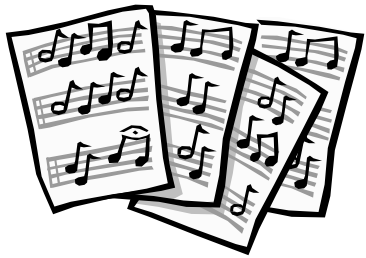
В качестве носителя документации выступает XML-код. Ряд XML-тегов предопределен в качестве документирующих тегов (они выступают аналогом документирующих тегов в Java, однако оформляются с использованием стандартного синтаксиса XML). В отличие от Java, обработку документирующих комментариев осуществляет компилятор C#, а не специальная утилита. Следующий фрагмент иллюстрирует использование документирующих комментариев C#.

```
/// <summary>
/// Plain triangle construction and processing class
/// </summary>
/// <remarks>
/// <see cref="Point">Point</see>
/// </remarks>
public class Triangle {
    //...

    ///
    /// Triangle construction by using three points
    /// <param name = "p1">First point reference</param>
    /// <param name = "p2">Second point reference</param>
    /// <param name = "p3">Third point reference</param>
    /// <returns>Returns true, if successful
    /// Otherwise returns false
    /// </returns>
    public boolean ConstructTriangle( Point p1,
                                      Point p2,
                                      Point p3 ) {
        // ...
    }
}
```

Компилятор C# контролирует полноту и целостность документации. Это означает, что если, например, один параметр метода класса документирован, то должны быть документированы и остальные параметры. Некоторые подробности использования документирующих комментариев C# можно почерпнуть из [Gunnerson, 2000].

Полученный XML-документ может быть отображен с использованием различных механизмов представления (таблиц каскадных стилей, сценариев объектной модели документа, расширенных таблиц стилей XLS), а также связан с HTML-документами (см., например, [Young, 2000]). Среда проектирования Visual Studio .NET содержит встроенные средства генерации набора HTML-файлов, создаваемых на основе извлекаемых из исходных текстов документирующих комментариев (Comment Web Page Report).

ГЛАВА 12



Введение в компонентное программирование

> Компьютер — это почти человек. Единственное серьезное отличие: ему несвойственно сваливать свои ошибки на другой компьютер.
>
> *Из компьютерного фольклора*

Необходимое предварительное замечание

Так же как и в науке, где конец одного исследования обычно становится началом новой работы, в образовании завершение изучения одного курса (предмета, технологии) часто становится отправной точкой для постановки новых задач, требующих овладения новыми знаниями. В этом смысле предлагаемая глава является своего рода "мостом" к другому курсу, содержание и возможность успешного освоения которого основывается на понимании объектно-ориентированных концепций и механизмов. Поэтому данная глава, вероятно, больше других ориентирована на самостоятельную работу читателя, определяя ряд основных направлений изучения современных технологий, связанных с компонентной моделью.

При современном уровне развития программно-аппаратных комплексов в качестве элементов повторного использования чаще выступают не отдельные типы и ресурсы, а целостные компоненты, представляющие собой библиотеки типов (и/или функций). В связи с этим основной проблемой становится реализация механизмов, обеспечивающих взаимосвязь и совместное использование различных компонентов, а также реализация механизмов доступа к отдельным ресурсам компонентов извне.

При этом проблема распределения компонентов связана со сложностью обеспечения межплатформенной независимости ресурсов на уровне двоичного кода (а иногда — и на уровне исходных текстов программ), а также со сложностью обеспечения взаимодействия кода, разработанного на различных языках с использованием различных технологий программирования.

Организация взаимодействия относительно обособленных компонентов программного обеспечения является непростой задачей. Даже отлаженные и протестированные компоненты могут некорректно работать в новой исполнительной среде. Одним из классических примеров катастрофической ошибки в программном обеспечении, связанной с некорректным вторичным использованием отлаженных компонентов, является авария ракеты-носителя Ariane 5 в 1996 году [Богатырев, 1997].

Изучение всего разнообразия моделей компонентного программирования не является основной задачей этой небольшой главы. Здесь мы сосредоточимся на представлении основных принципов построения приложений, ориентированных на использование внешних компонентов, и на изучении влияния компонентного программирования на развитие объектно-ориентированной парадигмы проектирования.

В качестве основной модели компонентной архитектуры для иллюстрации исследуемых в этой главе механизмов и разработки примеров программ используется архитектура COM (Component Object Model).

12.1. Понятие о компонентной архитектуре

Для начала рассмотрим идеи, лежащие в основе компонентного подхода. Некоторые из них вписываются в объектно-ориентированный подход, некоторые — не связаны напрямую с объектной ориентированностью программ. Следуя основной тематике книги, мы сосредоточимся, главным образом, на тех элементах компонентной архитектуры, которые работают во взаимодействии с объектно-ориентированными механизмами.

Сокрытие реализации

Современные программы редко пишутся раз и навсегда. Изменение требований пользователя, изменения в операционных системах, необходимость учета новых типов входных данных, развитие интерфейса приложения приводят к необходимости вносить изменения в уже работающее приложение. В частности, такие изменения могут быть связаны с исправлением ошибок, обнаруженных в спроектированных программах после начала их использования. При этом во многих случаях не вся совокупность исходного кода находится под контролем группы программистов, выпускающих приложение: изменения могут затрагивать и внешний код. В связи с этим актуальна проблема распределения кода: как внести изменения в код, используемый во многих клиентских проектах, с минимальным воздействием на целостность клиентского кода? Ясно, что обеспечение подобной независимости компонентов невозможно без понимания концепции сокрытия реализации, или отделения интерфейса от реализации.

Отметим, что развитие компонентных архитектур является, вероятно, важным фактором, обусловившим появление в языковых спецификациях таких элементов как свойства *(см. разд. 11.2)* и события *(см. разд. 11.4)*. Свойства в явном виде выражают идею возможного различия интерфейса и реализации данных. События на более высоком уровне абстрагируют делегаты, выражая тот факт, что только класс (читай — компонент), в котором объявлено событие, может инициировать его.

Примечание

Сущность свойств с точки зрения компонентно-ориентированного программирования довольно ярко характеризует возможность объявления абстрактных свойств (см., например, спецификацию языка C#) наряду с абстрактными (чисто виртуальными) методами класса. Это делает свойства естественным элементом определения абстрактного класса или интерфейса.

Повторное использование кода

Одна из основных глобальных задач, решение которой обеспечивается объектно-ориентированным подходом, заключается в предоставлении механизмов повторного использования ранее спроектированных компонентов (в основном — на базе наследования классов). При этом возможны три основных варианта.

Первый вариант предполагает повторное использование классов, доступных в форме исходных текстов (рис. 12.1). С одной стороны, достоинством этого варианта является отсутствие проблем с возможными различиями в реализации компиляторов, поскольку клиентский код компилируется вместе с библиотечным. С другой стороны, повторное использование компонентов, доступных в форме исходных текстов, затрудняет использование модифицированных версий библиотечного кода. Это связано не только с необходимостью перекомпиляции, но и с тем, что в ходе разработки сторонний код зачастую подвергается исправлениям и модификациям уже на клиентской стороне.

Второй вариант связан с помещением библиотечного кода в состав статически подключаемых библиотек (распространяемых в виде файлов с расширением lib), содержащих объектный код. В этом случае связывание клиентского и библиотечного кода происходит на стадии компоновки, а для компиляции клиентского кода требуются только определения типов, используемых в библиотечном коде. Это не позволяет полностью отделить клиентский код от библиотечного, поскольку для успешной компиляции клиентского кода требуется полная информация об определении классов, включая закрытые части, а значит — независимая модификация библиотечного кода по-прежнему за-

труднена. Кроме этого, код библиотек должен включаться во все клиентские проекты.

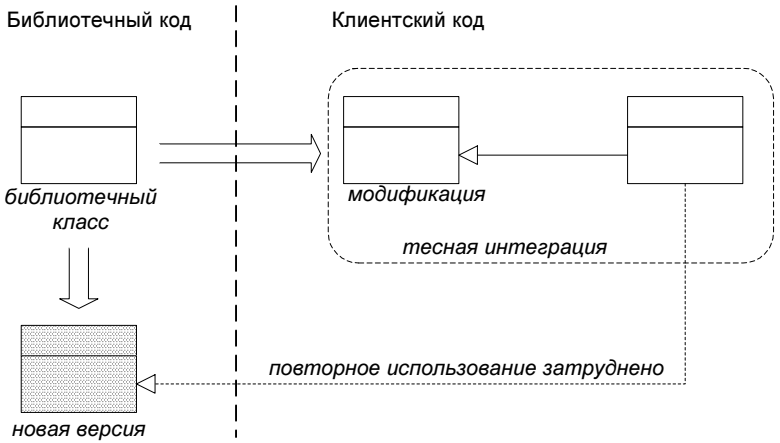


Рис. 12.1. Повторное использование кода через наследование классов

Третий вариант предполагает использование динамически подключаемых библиотек (распространяемых в виде файлов с расширением dll). При этом возникает, по меньшей мере, две проблемы (см. [Добрынин, 2003]).

- Отсутствие двоичного стандарта C++ приводит к тому, что различные компиляторы могут по-разному реализовывать поддержку различных особенностей языка (например, таблиц виртуальных функций), поэтому в общем случае нельзя гарантировать, что DLL-модуль, подготовленный с использованием одного компилятора, будет корректно работать с клиентом, компилируемым другим компилятором.
- Динамическая компоновка позволяет скрыть от клиента реализацию класса на уровне исходных текстов, что вроде бы позволяет модифицировать библиотечный код при отсутствии изменений в клиентском коде. Но и при этом перекомпиляция клиента в общем случае необходима. Это связано с тем, что именно клиент решает вопросы размещения экземпляра библиотечного класса в памяти. Поэтому даже если при изменении реализации класса был изменен только порядок следования данных класса (не говоря уже о составе данных), то клиент не будет работать с новой версией DLL без перекомпиляции.

Таким образом, следование объектно-ориентированной парадигме позволяет реализовать механизмы повторного использования кода на уровне исходных текстов, но не преодолевает проблемы распространения библиотечного кода в двоичной форме.

Динамическая компоновка и совместимость интерфейсов

Концептуально, следование компонентной архитектуре приложения предполагает представление приложения не в виде монолитного кода, а в виде совокупности взаимодействующих компонентов, распространяемых в виде двоичных кодов. Если изменения затрагивают только один компонент, то в составе приложения заменяется только этот компонент.

Поэтому для реализации полноценной компонентной архитектуры приложения важно обеспечить возможность динамической компоновки приложения. Динамическая компоновка нужна для того, чтобы можно было заменить используемые приложением компоненты без перекомпиляции и перекомпоновки всего приложения в целом. Во многих случаях это единственный механизм, позволяющий обновлять версии приложений "по частям". В противном случае разработчик должен был бы постоянно рассылать всем клиентам новые полные версии приложения. В случае наличия динамической компоновки пользователю нужно прислать только измененный компонент. В противоположность статическим библиотекам (распространяемым в виде файлов с расширением lib), которые связываются с использующим их кодом в процессе компоновки, связывание кода, содержащегося в динамически подключаемых библиотеках, с клиентским кодом осуществляется в процессе исполнения программы.

Разумеется, замена компонента с использованием динамической компоновки может быть корректной только в том случае, когда новый компонент полностью поддерживает интерфейс прежней версии компонента. Отсюда следует, что организация взаимодействия компонентов невозможна без понимания концепции интерфейса и механизмов обеспечения совместимости интерфейсов *(см. гл. 7)*.

12.2. Реализация основных элементов компонентной архитектуры на базе COM

В этом разделе мы рассмотрим механизмы реализации компонентной архитектуры в том виде, в каком они обеспечиваются в рамках модели компонентной архитектуры COM. Спецификация COM является вариантом организации компонентной архитектуры, оказавшей существенное влияние на развитие компонентных приложений, работающих под управлением Windows. Исследование основных проблем, возникающих в связи с использованием COM, не только позволяет разъяснить основные задачи и проблемы компонентного программирования как такового, но и в новом свете взглянуть на те

задачи, которые ставились перед разработчиками архитектуры и инструментальных средств платформы Microsoft. NET, во многом рассматриваемой как альтернатива средствам COM для приложений, работающих под управлением операционных систем Windows.

Доступ к компоненту посредством запроса интерфейса

Последовательно рассматривая несколько примеров в этом и последующих разделах, мы выделим важные концепции, лежащие в основе компонентной архитектуры, и механизмы реализации этих концепций. Начнем с рассмотрения примера, иллюстрирующего, каким образом обеспечивается взаимодействие компонентов на базе поддерживаемых компонентами интерфейсов. Вспомним пример с геометрическими фигурами и рассмотрим модель компонента, реализующего некоторую функциональность в связи с представлением геометрических фигур. Пусть для внешнего пользователя наш компонент предоставляет два интерфейса: интерфейс, обеспечивающий "рисование" графического объекта, и интерфейс управления цветом графического объекта. Описание интерфейсов в этом случае может иметь следующий вид:

```
#include <objbase.h> // содержит #define interface struct
interface IShape
{
    virtual void __stdcall Draw() = 0;
    virtual void __stdcall Erase() = 0;
};

interface IColorable
{
    virtual void __stdcall SetColor( int ) = 0;
    virtual int __stdcall GetColor() = 0;
    virtual const char* __stdcall ToString() = 0;
};
```

Определение класса `ColoredShape`, реализующего указанные интерфейсы, в самом общем плане может выглядеть следующим образом (как и раньше, мы ограничимся некоторой модельной схемой решения):

```
class ColoredShape: public IShape,
                    public IColorable {

    // Реализация интерфейса IShape
    virtual void __stdcall Draw() { cout << "Shape drawn" << endl; }
    virtual void __stdcall Erase() { cout << "Shape erased" << endl; }
```

```
    // Реализация интерфейса IColorable
    virtual void __stdcall SetColor( int color ) {
        this->color = color;
        cout << "Color is set to " << color << endl;
    }
    virtual int __stdcall GetColor() { return color; }
    virtual const char* __stdcall ToString() {
        switch( color ) {
            case 0 : return "white";
            case 1 : return "red";
            case 2 : return "green";
            case 3 : return "yellow";
            case 4 : return "blue";
            default: return "unknown";
        }
    }

private:
    int color;
};
```

Примечание

Обратите внимание на служебное слово `__stdcall`, использованное в заголовках функций. Такое служебное слово поддерживается всеми компиляторами C++, генерирующими код для Windows. Функция, помеченная таким образом, использует соглашение о вызове, в соответствии с которым стек очищается перед возвратом управления вызывающей функции (это соглашение называют "стандартным соглашением о вызове функции"). Обычный порядок в C/C++ предполагает, что за очистку стека "ответственна" вызывающая функция, а не вызываемая. Необходимость следования стандартному соглашению связана с тем, что именно такое соглашение о вызове используется в большинстве языков (в частности, в языке Pascal). Вместо `__stdcall` можно использовать определение `STDMETHODCALL` из objbase.h или определение `pascal` из windef.h.

В нашем примере компонентом является, в конечном счете, класс C++. Отметим, что компонент — это не обязательно один класс, он может включать и несколько классов. Кроме того, компонент вообще не обязан быть объектно-ориентированным. Соответственно, все поддерживаемые компонентом интерфейсы не обязательно должны быть реализованы в одном классе. В данной книге по уже указанным выше причинам мы ограничиваемся рассмотрением объектно-ориентированных компонентов.

Клиент обращается к компоненту посредством так называемого запроса интерфейса. Это означает, что клиенту нужно знать, какие интерфейсы поддерживает компонент. Сама реализация компонента клиенту недоступна. Это

предоставляет возможность замены компонента другим, не влияя на работу клиентского кода, если модификация компонента не затрагивает используемые клиентом интерфейсы (рис. 12.2).

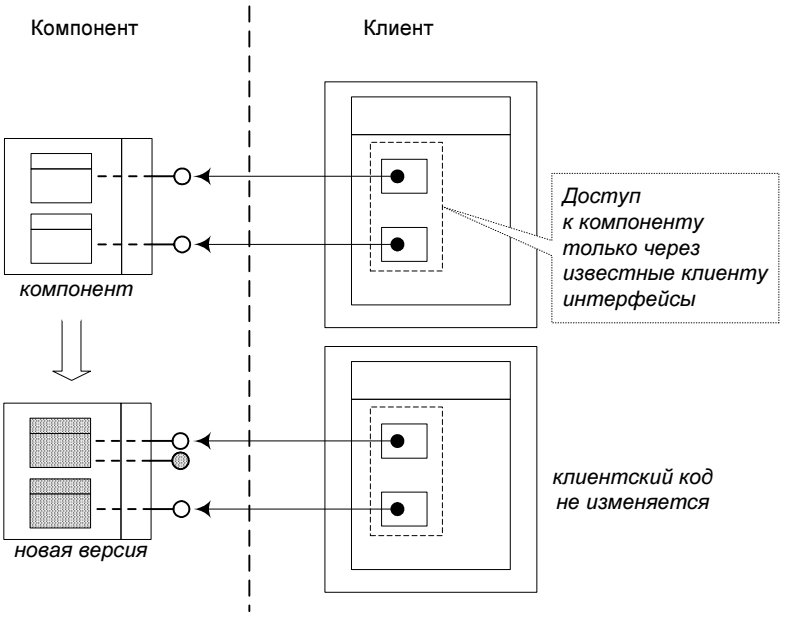


Рис. 12.2. Повторное использование кода при компонентном подходе

Казалось бы, запрос интерфейса у компонента может быть выполнен почти тривиально:

```
ColoredShape* pC = new ColoredShape; // Создать экземпляр компонента
IShape *pIShape = pC; // Получить указатель на интерфейс IShape
pIShape->Draw(); // Использовать интерфейс IShape
IColorable *pIColorable = pC; // Получить указатель на IColorable
pIColorable->SetColor( 0 ); // Использовать интерфейс IColorable
```

Однако такое решение не годится, т. к. клиент в этом случае не следует правилу взаимодействия с компонентом только через интерфейс. Для компиляции клиентского кода требуется знание типа `ColoredShape`, который, с точки зрения компонентной архитектуры, должен быть скрыт от пользователя. Очевидно, что в этом случае невозможно обеспечить независимость клиентского кода от возможных изменений в реализации компонента.

Поэтому требуется какое-то имя типа, которое могло бы выступать в роли интерфейса, известного всем компонентам и поддерживаемого всеми компо-

нентами. В рамках архитектуры COM таким интерфейсом является интерфейс с несколько парадоксальным именем `IUnknown` ("неизвестный интерфейс"). В соответствии с требованиями спецификации COM, все интерфейсы должны наследовать интерфейсу `IUnknown`. Это позволяет клиенту, имея указатель на `IUnknown`, запросить через него другие интерфейсы, поддерживаемые компонентом. Интерфейс `IUnknown`, определение которого располагается в заголовочном файле unknwn.h, входящем в состав Win32 SDK, содержит объявления трех методов:

```
interface IUnknown {
    virtual HRESULT __stdcall QueryInterface( const IID&, void** ) = 0;
    virtual ULONG __stdcall AddRef() = 0;
    virtual ULONG __stdcall Release() = 0;
}
```

Поскольку все интерфейсы COM наследуют `IUnknown`, у каждого интерфейса первые три элемента таблицы виртуальных функций содержат указатели на три функции, определяемые интерфейсом `IUnknown`. Таким образом, любой интерфейс можно использовать для запроса другого интерфейса, поддерживаемого компонентом.

Функция `QueryInterface(const IID& iid, void** ppI)` должна быть реализована следующим образом:

- если компонент поддерживает интерфейс, идентифицируемый параметром `iid`, то по адресу `ppI` должно быть записано значение указателя на запрашиваемый интерфейс, при этом функция возвращает код `S_OK`;
- если компонент не поддерживает интерфейс, идентифицируемый параметром `iid`, функция возвращает код `E_NOINTERFACE`, а по адресу `ppI` записывается значение `NULL`.

Значение, возвращаемое функцией `QueryInterface()` (так называемый код `HRESULT`), является значением, представленным 32-разрядным кодом. Структура кода `HRESULT` определяется спецификацией COM; предопределенные возможные значения `HRESULT` находятся в заголовочном файле winerror.h, входящем в состав Win32 SDK.

Отложим на некоторое время обсуждение других функций, объявляемых в интерфейсе `IUnknown`, и рассмотрим, как запросить интерфейс, поддерживаемый компонентом, при наличии указателя на `IUnknown`. Сначала разберемся с тем, как получить значение указателя на `IUnknown`, который в действительности был бы связан с экземпляром используемого компонента. Поскольку внутри компонентного модуля тип компонентного класса `ColoredShape` известен, не составляет труда определить в этом модуле функцию примерно следующего содержания:

```
IUnknown* CreateInstance() {
    // Создать экземпляр компонента и преобразовать указатель
    // на объект класса ColoredShape к указателю на IUnknown
    IUnknown* pI = static_cast<IShape*>( new ColoredShape );

    return pI; // Вернуть значение указателя на IUknown
}
```

Функция `CreateInstance()` экспортируется компонентом для того, чтобы в клиентском коде можно было получить доступ к функциональности, предоставляемой компонентом, используя только информацию о поддерживаемых компонентом интерфейсах. Корректность преобразования указателя `static_cast`, выполняемого в функции `CreateInstance()`, обеспечивается соглашением о том, что все компоненты COM обязаны предоставить реализацию интерфейса `IUnknown`.

Примечание

Обратите внимание, что в функции `CreateInstance()` указатель на объект класса `ColoredShape`, созданный в динамической памяти, явно преобразуется к указателю на `IShape`, поскольку непосредственное преобразование к указателю на `IUnknown` неоднозначно (класс `ColoredShape` непосредственно не наследует интерфейса `IUnknown`). После успешного преобразования к типу `IShape*` указатель неявным повышающим приведением типа преобразуется к типу `IUnknown*`.

Теперь, обращаясь в клиентском коде к функции `CreateInstance()`, мы можем сформировать "легальное" значение указателя на `IUnknown` и, используя его, запросить любой интересующий нас интерфейс:

```
// Клиент
void main() {
    HRESULT hr;

    // Создать экземпляр компонента и получить указатель на IUnknown
    IUnknown* pIUnknown = CreateInstance();

    // Запрашиваем интерфейс IShape
    IShape* pIShape = NULL;
    hr = pIUnknown->QueryInterface(
        IID_IShape,       // Идентификатор интерфейса (см. далее)
        ( void** )&pIShape  // Адрес указателя на интерфейс
   );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс
        pIShape->Draw();
```

```
        pIShape->Erase();
    }
    // Работаем с другими интерфейсами
    // ...

    // Удаляем объект
    delete pIUnknown;
}
```

Обратите внимание, что поскольку значение HRESULT представляет собой двоичный код определенной структуры, то для проверки успешности операции запроса интерфейса следует использовать специальный макрос SUCCEEDED, а не просто осуществлять сравнение значения HRESULT с одним из возможных кодов.

Теперь, когда отдельные элементы общей схемы организации взаимодействия с компонентом выявлены, рассмотрим законченный пример, обращая внимание на некоторые детали (листинг 12.1). Для изучения основных элементов компонентной архитектуры мы используем простой пример без динамически подключаемых модулей, весь исходный текст которого располагается в одном файле. Немного позднее мы осуществим декомпозицию кода, выделяя самостоятельные элементы, относящиеся к компоненту, клиенту и общие файловые единицы компонента и клиента.

Листинг 12.1. Взаимодействие с компонентом через поддерживаемые им интерфейсы

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

/*
 * Интерфейсы, поддерживаемые компонентом
 */

// Интерфейс IShape наследует "универсальному" интерфейсу IUnknown
interface IShape : IUnknown {
    virtual void __stdcall Draw() = 0;
    virtual void __stdcall Erase() = 0;
};

// Интерфейс IColorable наследует "универсальному" интерфейсу IUnknown
interface IColorable : IUnknown {
    virtual void __stdcall SetColor( int ) = 0;
    virtual int __stdcall GetColor() = 0;
    virtual const char* __stdcall ToString() = 0;
};
```

```
/*
 * Предварительные объявления уникальных идентификаторов интерфейсов
 *  (GUID — Global Uniqie IDentifier)
 */

extern const IID IID_IShape;
extern const IID IID_IColorable;

/*
 * Компонент (еще не совсем COMпонент)
 */
class ColoredShape: public IShape,
                    public IColorable
{
    // Реализация интерфейса IUnknown
    virtual HRESULT __stdcall QueryInterface( const IID&, void** );
    virtual ULONG __stdcall AddRef() { return 0; }
    virtual ULONG __stdcall Release() { return 0; }

    // Реализация интерфейса IShape
    virtual void __stdcall Draw() { cout << "Shape drawn" << endl; }
    virtual void __stdcall Erase() { cout << "Shape erased" << endl; }

    // Реализация интерфейса IColorable
    virtual void __stdcall SetColor( int color ) {
        this->color = color;
        cout << "Color is set to " << color << endl;
    }
    virtual int __stdcall GetColor() {
        return color;
    }
    virtual const char* __stdcall ToString() {
        switch( color ) {
            case 0 : return "white";
            case 1 : return "red";
            case 2 : return "green";
            case 3 : return "yellow";
            case 4 : return "blue";
            default: return "unknown";
        }
    }

private:
    int color;
};
```

```
// Запрос интерфейса
HRESULT __stdcall ColoredShape::QueryInterface(
        const IID& iid, void** ppI ) {

    if( iid == IID_IUnknown ) {
        *ppI = static_cast<IShape*>( this );
    }
    else if( iid == IID_IShape ) {
        *ppI = static_cast<IShape*>( this );
    }
    else if( iid == IID_IColorable ) {
        *ppI = static_cast<IColorable*>( this );
    }
    else {
        *ppI = NULL;
         return E_NOINTERFACE;
    }

    return S_OK;
}

/*
 * Функция создания экземпляра компонента
 */
IUnknown* CreateInstance() {
    // Создать экземпляр компонента и преобразовать указатель
    // на объект класса ColoredShape к указателю на IUnknown
    IUnknown* pI = static_cast<IShape*>( new ColoredShape );

    return pI; // Вернуть значение указателя на IUnknown
}

/*
 * Определения уникальных идентификаторов интерфейсов
 * (генерируются утилитой uuidgen.exe)
 */
// {c7f4f55a-3a4f-438a-9b50-a0cb5b955073}
static const IID IID_IShape =
    {0xc7f4f55a, 0x3a4f, 0x438a,
    {0x9b, 0x50, 0xa0, 0xcb, 0x5b, 0x95, 0x50, 0x73}};

// {78c71e52-6b12-4e18-ae4d-a104206728c6}
static const IID IID_IColorable =
    {0x78c71e52, 0x6b12, 0x4e18,
    {0xae, 0x4d, 0xa1, 0x04, 0x20, 0x67, 0x28, 0xc6}};
```

```
/*
 * Клиент
 */
void main() {
    HRESULT hr;

    // Создать экземпляр компонента и получить указатель на IUnknown
    IUnknown* pIUnknown = CreateInstance();

    // Запрашиваем интерфейс IShape
    IShape* pIShape = NULL;
    hr = pIUnknown->QueryInterface( IID_IShape, ( void** )&pIShape );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IShape
        pIShape->Draw();
        pIShape->Erase();
    }

    // Запрашиваем интерфейс IColorable
    IColorable* pIColorable = NULL;
    hr = pIUnknown->QueryInterface( IID_IColorable,
                                    ( void** )&pIColorable );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IColorable
        pIColorable->SetColor( 2 );
        cout << "Returned color is " << pIColorable->GetColor() << endl;
        cout << "Color is " << pIColorable->ToString() << endl;
    }

    // Удаляем объект
    delete pIUnknown;
}
```

Результат исполнения данной программы позволяет убедиться в том, что происходит обращение к методам, определяемым используемыми интерфейсами компонента:

```
Shape drawn
Shape erased
Color is set to 2
Returned color is 2
Color is green
```

Идентификация интерфейсов, поддерживаемых компонентом

Поскольку доступ к компонентам происходит через запрос интерфейсов, необходимо обеспечить уникальность идентификации интерфейсов. Схема, принятая в COM, основывается на использовании глобальных уникальных идентификаторов (Global Unique IDentifiers, GUID). Существует также термин "вселенски уникальный идентификатор" (Universally Unique IDentifier, UUID), но он, кажется, получил меньшее распространение. Разработчик на Visual C++ для генерации GUID может использовать специальную утилиту uuidgen.exe, поставляемую вместе со средствами разработки. При каждом новом запуске утилиты получается новый GUID, причем он будет отличен не только от GUID, сгенерированных ранее на этой машине, но и от GUID, сгенерированных другими разработчиками на их компьютерах. Для обеспечения уникальности идентификаторов GUID утилита uuidgen.exe использует информацию о сетевом адресе компьютера (если он подключен к сети) и точном времени, что практически исключает возможность совпадения GUID для разных интерфейсов [Rogerson, 1997]. Не вдаваясь в подробности внутреннего устройства GUID, отметим, что GUID являются структурами, соответствующими 128-разрядному двоичному коду. Использование структур вместо обычных 128-битных чисел объясняется тем, что не все языки программирования, поддерживающие COM, содержат встроенные средства для манипулирования такими большими числами. Примеры использования GUID для идентифицирования интерфейсов представлены в листинге 12.1 в виде констант `IID_IShape` и `IID_IColorable`.

Управление временем жизни компонента

Если абстрагироваться от того факта, что в примере листинга 12.1 и компонент, и клиент реализованы в пределах одного монолитного приложения, представленный код клиента выглядит вполне отделенным от кода компонента, за исключением одной неприятной детали. В данной реализации клиенту следует заботиться об удалении экземпляра компонента из динамической памяти (фактически, это означает, что клиенту следует заботиться об управлении временем жизни компонента):

```
delete pIUnknown;
```

Необходимость явного управления временем жизни компонента не позволяет считать клиентский код полностью изолированным от особенностей реализации компонента ввиду следующих обстоятельств. В общем случае, клиент может представлять собой сложную программу, в разных местах которой многократно используются различные интерфейсы одного и того же компо-

нента. В связи с этим, может оказаться затруднительным выяснение вопроса о том, нужно ли удалять компонент, или его интерфейсы еще где-то используются. Можно, конечно, не удалять компоненты до завершения работы всего приложения, однако такое решение, очевидно, неэффективно. Кроме того, непосредственно перед выгрузкой компонента, возможно, требуется выполнение каких-либо завершающих действий, недоступных (или неизвестных) клиенту. Таким образом, хорошо бы избавить клиентский код от необходимости явного управления временем жизни компонента. Здесь мы и подходим к назначению двух функций интерфейса `IUnknown`, которые не были рассмотрены ранее: `AddRef()` и `Release()`.

Данные функции предназначены для ведения учета ссылок на интерфейсы, поддерживаемые компонентом. Поскольку ссылки создаются в процессе взаимодействия клиентского кода с компонентом, подсчет ссылок позволяет компоненту самому удалить себя из памяти, когда число ссылок становится равным нулю. В зависимости от особенностей реализации, компонент может вести общий учет ссылок на все интерфейсы или раздельный учет ссылок на отдельные интерфейсы — все это определяется тем, как реализованы функции `AddRef()` и `Release()` в составе компонента. С точки зрения клиента, учет ссылок ведется раздельно для каждого интерфейса. Вот как может выглядеть реализация механизма подсчета ссылок в самом типичном случае:

```
class ColoredShape: public IShape,
                    public IColorable
{
    // Реализация интерфейса IUnknown
    virtual HRESULT __stdcall QueryInterface( const IID&, void** );
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

    // Реализация других интерфейсов
    // ...

private:
    long refCount; // Счетчик ссылок на интерфейсы
    // ...
};

// Увеличение refCount при создании ссылки на интерфейс
ULONG __stdcall ColoredShape::AddRef() {
    return InterlockedIncrement( &refCount );
}

// Освобождение интерфейса
ULONG __stdcall ColoredShape::Release() {
```

```
    if( InterlockedDecrement( &refCount ) == 0 ) {
        // Удаляем компонент, если больше нет ссылок
        // на его интерфейсы
        delete this;
        return 0;
    }
    return refCount;
}
```

Использование вместо операций ++ и -- функций `InterlockedIncrement()` и `InterlockedDecrement()` обусловлено возможностью работы с компонентами многопоточных приложений. Применение этих функций гарантирует, что значение переменной `refCount` в каждый момент времени может быть изменено только одним потоком управления. В рамках исследуемых нами однопоточных приложений обращения к этим функциям могут быть заменены обычными операциями увеличения и уменьшения на единицу.

Несмотря на то, что функции `AddRef()` и `Release()` в нашей задаче возвращают текущее значение счетчика ссылок, в общем случае клиент не должен полагаться на осмысленность этих значений и использовать их для управления поведением программы. Использовать их можно только для отладочных целей.

Функции, которые возвращают указатель на интерфейс, должны вызывать функцию `AddRef()` непосредственно перед возвратом управления. В нашем примере это относится к функции `CreateInstance()`, возвращающей указатель на интерфейс `IUnknown`, и к функции `QueryInterface()`, обеспечивающей доступ к запрошенному интерфейсу. Таким образом, реализации указанных функций должны быть дополнены обращением к функции `AddRef()`. В листинге 12.2 приведен пример определения компонента `ColoredShape` и клиента, использующего этот компонент, с учетом изменений, необходимых в связи с обеспечением управления временем жизни компонента.

Листинг 12.2. Управление временем жизни компонента: подсчет работающих интерфейсов

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

/*
 * Интерфейсы, поддерживаемые компонентом
 */

// Интерфейс IShape наследует "универсальному" интерфейсу IUnknown
interface IShape : IUnknown {
```

```
    virtual void __stdcall Draw() = 0;
    virtual void __stdcall Erase() = 0;
};

// Интерфейс IColorable наследует "универсальному" интерфейсу IUnknown
interface IColorable : IUnknown {
    virtual void __stdcall SetColor( int ) = 0;
    virtual int __stdcall GetColor() = 0;
    virtual const char* __stdcall ToString() = 0;
};

/*
 * Предварительные объявления уникальных идентификаторов интерфейсов
 *  (GUID — Global Uniqie IDentifier)
 */

extern const IID IID_IShape;
extern const IID IID_IColorable;

/*
 * Компонент (уже почти COMпонент)
 */
class ColoredShape: public IShape,
                    public IColorable
{
    // Реализация интерфейса IUnknown
    virtual HRESULT __stdcall QueryInterface( const IID&, void** );
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

    // Реализация интерфейса IShape
    virtual void __stdcall Draw() { cout << "Shape drawn" << endl; }
    virtual void __stdcall Erase() { cout << "Shape erased" << endl; }

    // Реализация интерфейса IColorable
    virtual void __stdcall SetColor( int color ) {
        this->color = color;
        cout << "Color is set to " << color << endl;
    }
    virtual int __stdcall GetColor() {
        return color;
    }
    virtual const char* __stdcall ToString() {
        switch( color ) {
            case 0 : return "white";
            case 1 : return "red";
```

```
            case 2 : return "green";
            case 3 : return "yellow";
            case 4 : return "blue";
            default: return "unknown";
        }
    }

public:
    // Конструктор компонента
    // NB! Обеспечивает инициализацию refCount
    ColoredShape() : refCount( 0 ) {
        cout << "ColoredShape instance created" << endl;
    }

    // Деструктор компонента
    ~ColoredShape() {
        cout << "ColoredShape instance destroyed" <<endl;
    }

private:
    long refCount; // Счетчик ссылок на интерфейсы
    int  color;
};

/*
 * Реализация методов интерфейса IUnknown
 */

// Запрос интерфейса
HRESULT __stdcall ColoredShape::QueryInterface(
        const IID& iid, void** ppI ) {

    if( iid == IID_IUnknown ) {
        *ppI = static_cast<IShape*>( this );
    }
    else if( iid == IID_IShape ) {
        *ppI = static_cast<IShape*>( this );
    }
    else if( iid == IID_IColorable ) {
        *ppI = static_cast<IColorable*>( this );
    }
    else {
        *ppI = NULL;
         return E_NOINTERFACE;
    }
```

```
    reinterpret_cast<IUnknown*>(*ppI)->AddRef(); // Важное дополнение!
    return S_OK;
}

// Увеличение refCount при создании ссылки на интерфейс
ULONG __stdcall ColoredShape::AddRef() {

    cout << "ColoredShape: refCount = " << refCount+1 << endl;

    return InterlockedIncrement( &refCount );
}

// Освобождение интерфейса
ULONG __stdcall ColoredShape::Release() {

    cout << "ColoredShape: refCount = " << refCount-1 << endl;

    if( InterlockedDecrement( &refCount ) == 0 ) {
        // Удаляем компонент, если больше нет ссылок
        // на его интерфейсы
        delete this;
        return 0;
    }
    return refCount;
}

/*
 * Функция создания экземпляра компонента
 */
IUnknown* CreateInstance() {
    // Создать экземпляр компонента и преобразовать указатель
    // на объект класса ColoredShape к указателю на IUnknown
    IUnknown* pI = static_cast<IShape*>( new ColoredShape );

     pI->AddRef(); // Важное дополнение!
    return pI; // Вернуть значение указателя на IUnknown
}

/*
 * Определения уникальных идентификаторов интерфейсов
 * (генерируются утилитой uuidgen.exe)
 */
// {c7f4f55a-3a4f-438a-9b50-a0cb5b955073}
static const IID IID_IShape =
    {0xc7f4f55a, 0x3a4f, 0x438a,
    {0x9b, 0x50, 0xa0, 0xcb, 0x5b, 0x95, 0x50, 0x73}};
```

```
// {78c71e52-6b12-4e18-ae4d-a104206728c6}
static const IID IID_IColorable =
    {0x78c71e52, 0x6b12, 0x4e18,
    {0xae, 0x4d, 0xa1, 0x04, 0x20, 0x67, 0x28, 0xc6}};

/*
 * Клиент
 */
void main() {
    HRESULT hr;

    // Создать экземпляр компонента и получить указатель на IUnknown
    IUnknown* pIUnknown = CreateInstance();

    // Запрашиваем интерфейс IShape
    IShape* pIShape = NULL;
    hr = pIUnknown->QueryInterface( IID_IShape, ( void** )&pIShape );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IShape
        pIShape->Draw();
        pIShape->Erase();
        // Освобождаем интерфейс IShape
        pIShape->Release();
    }

    // Запрашиваем интерфейс IColorable
    IColorable* pIColorable = NULL;
    hr = pIUnknown->QueryInterface( IID_IColorable,
                                    ( void** )&pIColorable );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IColorable
        pIColorable->SetColor( 2 );
        cout << "Returned color is " << pIColorable->GetColor() << endl;
        cout << "Color is " << pIColorable->ToString() << endl;
        // Освобождаем интерфейс IColorable
        pIColorable->Release();
    }

    // Освобождаем интерфейс IUnknown
    pIUnknown->Release();
}
```

Для наблюдения за процессом взаимодействия клиента и компонента конструктор и деструктор компонента, а также функции `AddRef()` и `Release()`, снабжены печатью диагностических сообщений.

В результате исполнения программы будут получены следующие сообщения:

```
ColoredShape instance created
ColoredShape: refCount = 1
ColoredShape: refCount = 2
Shape drawn
Shape erased
ColoredShape: refCount = 1
ColoredShape: refCount = 2
Color is set to 2
Returned color is 2
Color is green
ColoredShape: refCount = 1
ColoredShape: refCount = 0
ColoredShape instance destroyed
```

Анализ вывода программы показывает, что при каждом успешном запросе интерфейса счетчик `refCount` увеличивается на единицу, а при каждом освобождении интерфейса — уменьшается на единицу. Конструктор класса `ColoredShape` вызывается при создании экземпляра компонента, деструктор вызывается в тот момент, когда счетчик `refCount` становится равным нулю (это означает, что больше нет используемых интерфейсов). В клиентском коде теперь не требуется явное завершение компонента, и клиент работает только с интерфейсами компонента. Для обеспечения корректной работы с компонентом клиент должен по завершении работы с каждым интерфейсом (в том числе — с интерфейсом `IUnknown`) вызывать метод `Release()`. Решение о необходимости явного вызова метода `AddRef()` более неоднозначно. Если указатель на интерфейс формируется как результат использования возвращаемого значения функций, подобных функциям `QueryInterface()` или `CreateInstance()`, то клиенту не нужно вызывать `AddRef()`.

Ситуация меняется, если в клиентском коде создается копия указателя на интерфейс:

```
// Предположим, что pIShape существует и связан с интерфейсом IShape
IShape *copy = pIShape;
```

В этом случае, поскольку появляется новая ссылка на интерфейс, следует вызвать метод `AddRef()` сразу после инициализации `copy`, а метод `Release()` — по завершении использования интерфейса `IShape` посредством `copy`:

```
IShape *copy = pIShape;
copy->AddRef(); // Учитываем новую ссылку на интерфейс
// Используем интерфейс
// ...
copy->Release(); // Освобождаем интерфейс
```

Примечание

Роджерсон приводит важное замечание по поводу копирования указателей на интерфейс. Если указатель является локальной переменной с ограниченной несколькими строками кода областью действия, а время жизни этого дополнительного указателя (в нашем случае речь идет об указателе `copy`) совпадает со временем жизни "базового" указателя (`pIShape`), то, вообще говоря, с целью оптимизации кода можно опустить пару вызовов `AddRef()` и `Release()` для интерфейса, связанного с указателем-копией, ограничившись подсчетом ссылок для "базового" указателя. Однако, по замечанию Роджерсона, "в реальных, не простых случаях гораздо сложнее понять, отсутствует ли вызов `AddRef()` и `Release()` по ошибке или вследствие оптимизации" [Rogerson, 1997]. Поэтому в общем случае следует руководствоваться правилом обязательного вызова `AddRef()` и `Release()` при создании копии указателя на интерфейс и его последующем освобождении.

Необходимость подсчета ссылок на интерфейсы является едва ли не главной проблемой управления компонентом. Кроме этого, источником ошибок при проектировании клиента может являться функция `QueryInterface()`, использующая аргумент, явно преобразуемый к типу `(void**)`. Поскольку такое преобразование успешно для любого указателя, достаточно трудно будет обнаружить ошибку передачи в качестве аргумента функции `QueryInterface()` адреса указателя на какой-либо другой интерфейс, а не на тот, что мы действительно хотели запросить. Таким образом, запрос интерфейса не относится к операциям, безопасным по отношению к типам. Для разрешения обозначенных проблем обычно используют классы smart-указателей, инкапсулирующих указатель на интерфейс и содержащих определение операций, используемых с указателем на интерфейс, или классы-оболочки (wrappers), инкапсулирующие интерфейс в целом и выступающие в роли своего рода прокси-класса для используемого интерфейса (рис. 12.3).

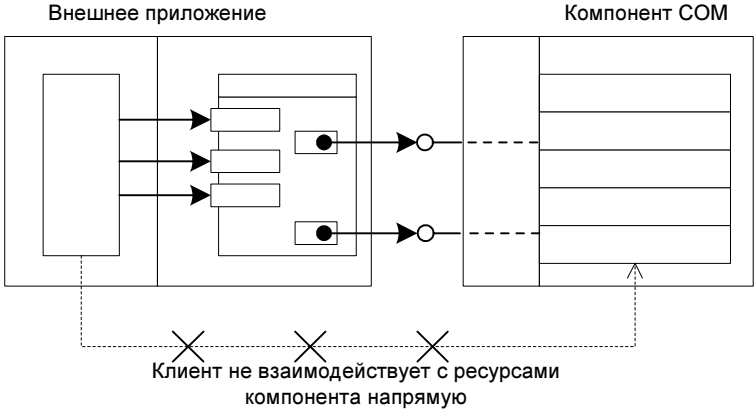


Рис. 12.3. Опосредованное взаимодействие клиента и компонента

Современные технологии проектирования, ориентированные на взаимодействие с компонентами COM (например, средства Microsoft Visual Studio), обычно предоставляют инструментальные средства, автоматизирующие создание классов оболочек для различных компонентов COM.

Явное выделение элементов архитектуры компонентного приложения

С целью упрощения изложения в предыдущих разделах мы рассматривали примеры программ, в которых и серверная, и клиентская части кода были реализованы в составе одной программы. Теперь осуществим декомпозицию проекта, представленного в листинге 12.2, с тем, чтобы выделить в качестве самостоятельных единиц трансляции участки кода, соответствующие и компоненту, и клиенту, а также совместно используемые файловые единицы. Результат реорганизации представлен в листинге 12.3.

Листинг 12.3. Элементы компонентной архитектуры приложения

```
/*
 * Файл: Interface.h
 * Определение интерфейсов, поддерживаемых компонентом
 *
 * Этот файл используется и компонентом, и клиентом
 */
#ifndef _Interfaces_h_
#define _Interfaces_h_

#include <objbase.h>

/*
 * Интерфейсы, поддерживаемые компонентом
 */

// Интерфейс IShape наследует "универсальному" интерфейсу IUnknown
interface IShape : IUnknown {
    virtual void __stdcall Draw() = 0;
    virtual void __stdcall Erase() = 0;
};

// Интерфейс IColorable наследует "универсальному" интерфейсу IUnknown
interface IColorable : IUnknown {
    virtual void __stdcall SetColor( int ) = 0;
    virtual int __stdcall GetColor() = 0;
    virtual const char* __stdcall ToString() = 0;
};
```

```
/*
 * Предварительные объявления уникальных идентификаторов интерфейсов
 *  (GUID - Global Uniqie IDentifier)
 */
extern "C"
{
    extern const IID IID_IShape;
    extern const IID IID_IColorable;
};
#endif

/*
 * Файл: ColoredShape.h
 * Определение класса компонента
 *
 * Этот файл используется только компонентом
 */
#ifndef _ColoredShape_h_
#define _ColoredShape_h_

#include <iostream.h>
#include <objbase.h>

#include "Interfaces.h"

/*
 * Класс в составе компонента
 */
class ColoredShape: public IShape,
                    public IColorable
{
    // Реализация интерфейса IUnknown
    virtual HRESULT __stdcall QueryInterface( const IID&, void** );
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

    // Реализация интерфейса IShape
    virtual void __stdcall Draw() { cout << "Shape drawn" << endl; }
    virtual void __stdcall Erase() { cout << "Shape erased" << endl; }

    // Реализация интерфейса IColorable
    virtual void __stdcall SetColor( int color ) {
        this->color = color;
        cout << "Color is set to " << color << endl;
    }
```

```
    virtual int __stdcall GetColor() {
        return color;
    }
    virtual const char* __stdcall ToString() {
        switch( color ) {
            case 0 : return "white";
            case 1 : return "red";
            case 2 : return "green";
            case 3 : return "yellow";
            case 4 : return "blue";
            default: return "unknown";
        }
    }

public:
    // Конструктор компонента
    // NB! Обеспечивает инициализацию refCount
    ColoredShape() : refCount( 0 ) {
        cout << "ColoredShape instance created" << endl;
    }

    // Деструктор компонента
    ~ColoredShape() {
        cout << "ColoredShape instance destroyed" <<endl;
    }

private:
    long refCount; // Счетчик ссылок на интерфейсы
    int  color;
};
#endif

/*
 * Файл: ColoredShape.cpp
 * Определение методов класса ColoredShape
 *
 * Этот файл используется только компонентом
 */
#include <iostream.h>
#include <objbase.h>

#include "Interfaces.h"
#include "ColoredShape.h"
```

```
/*
 * Реализация методов интерфейса IUnknown
 */

// Запрос интерфейса
HRESULT __stdcall ColoredShape::QueryInterface(
        const IID& iid, void** ppI ) {

    if( iid == IID_IUnknown ) {
        *ppI = static_cast<IShape*>( this );
    }
    else if( iid == IID_IShape ) {
        *ppI = static_cast<IShape*>( this );
    }
    else if( iid == IID_IColorable ) {
        *ppI = static_cast<IColorable*>( this );
    }
    else {
        *ppI = NULL;
         return E_NOINTERFACE;
    }

    reinterpret_cast<IUnknown*>( *ppI )->AddRef();
    return S_OK;
}

// Увеличение refCount при создании ссылки на интерфейс
ULONG __stdcall ColoredShape::AddRef() {

    cout << "ColoredShape: refCount = " << refCount+1 << endl;

    return InterlockedIncrement( &refCount );
}

// Освобождение интерфейса
ULONG __stdcall ColoredShape::Release() {

    cout << "ColoredShape: refCount = " << refCount-1 << endl;

    if( InterlockedDecrement( &refCount ) == 0 ) {
        // Удаляем компонент, если больше нет ссылок
        // на его интерфейсы
        delete this;
        return 0;
    }
    return refCount;
}
```

```
/*
 * Файл: Export.h
 * Объявление функций, экспортируемых компонентом
 *
 * Этот файл используется и компонентом, и клиентом
 */
#ifndef _Export_h_
#define _Export_h_

#include <unknwn.h>

// Прототип функции создания экземпляра компонента
IUnknown* CreateInstance();

#endif

/*
 * Файл: Export.cpp
 * Определение функций, экспортируемых компонентом
 *
 * Этот файл используется только компонентом
 */

#include "Export.h"
#include "Interfaces.h"
#include "ColoredShape.h"

/*
 * Функция создания экземпляра компонента
 */
IUnknown* CreateInstance() {
    // Создать экземпляр компонента и преобразовать указатель
    // на объект класса ColoredShape к указателю на IUnknown
    IUnknown* pI = static_cast<IShape*>( new ColoredShape );

     pI->AddRef();
    return pI; // Вернуть значение указателя на IUnknown
}

/*
 * Файл: IIDs.cpp
 * Определение GUIDs
 *
 * Этот файл используется и компонентом, и клиентом
 */
```

```
#include <objbase.h>

/*
 * Определения уникальных идентификаторов интерфейсов
 * (генерируются утилитой uuidgen.exe)
 */
extern "C"
{
    // {c7f4f55a-3a4f-438a-9b50-a0cb5b955073}
    extern const IID IID_IShape =
        {0xc7f4f55a, 0x3a4f, 0x438a,
        {0x9b, 0x50, 0xa0, 0xcb, 0x5b, 0x95, 0x50, 0x73}};

    // {78c71e52-6b12-4e18-ae4d-a104206728c6}
    extern const IID IID_IColorable =
        {0x78c71e52, 0x6b12, 0x4e18,
        {0xae, 0x4d, 0xa1, 0x04, 0x20, 0x67, 0x28, 0xc6}};
};

/*
 * Файл: Client.cpp
 * Реализация простого клиента
 *
 * Этот файл используется только клиентом
 */
#include <iostream.h>
#include <objbase.h>

#include "Interfaces.h"
#include "Export.h"

/*
 * Клиент
 */
void main() {
    HRESULT hr;

    // Создать экземпляр компонента и получить указатель на IUnknown
    IUnknown* pIUnknown = CreateInstance();

    // Запрашиваем интерфейс IShape
    IShape* pIShape = NULL;
    hr = pIUnknown->QueryInterface( IID_IShape, ( void** )&pIShape );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IShape
        pIShape->Draw();
        pIShape->Erase();
```

```
        // Освобождаем интерфейс IShape
        pIShape->Release();
    }

    // Запрашиваем интерфейс IColorable
    IColorable* pIColorable = NULL;
    hr = pIUnknown->QueryInterface( IID_IColorable,
                                    ( void** )&pIColorable );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IColorable
        pIColorable->SetColor( 2 );
        cout << "Returned color is " << pIColorable->GetColor() << endl;
        cout << "Color is " << pIColorable->ToString() << endl;
        // Освобождаем интерфейс IColorable
        pIColorable->Release();
    }

    // Освобождаем интерфейс IUnknown
    pIUnknown->Release();
}
```

Несмотря на то, что файлы, содержащиеся в листинге 12.3, все еще являются частями одного монолитного приложения, уже осуществлено отделение функциональности клиентского кода от определений, используемых исключительно компонентом (рис. 12.4).

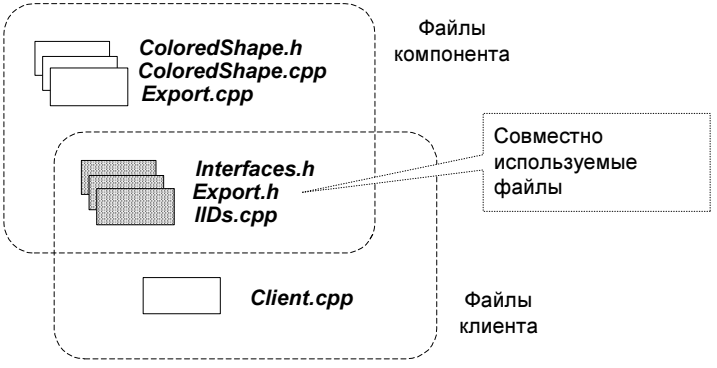


Рис. 12.4. Файловая структура компонента и клиента

Как явствует из изучения файла Client.cpp, для работы клиента необходима только информация об интерфейсах, поддерживаемых компонентом, и информация об экспортируемых компонентом функциях (в нашем случае речь идет о функции, обеспечивающей создание экземпляра компонента). Резуль-

таты исполнения этой программы не должны отличаться от результатов для листинга 12.2.

12.3. COM-сервер и COM-клиент: действующий макет

В *разд. 12.2* мы выделили обособленные элементы файловой структуры приложения, имитирующего взаимодействие клиентского кода и кода компонента. Теперь приступим к реализации компонента (COM-сервера) и клиента в виде полностью изолированных приложений, ведь достижение полной независимости серверного и клиентского кода и является основной задачей, решаемой на основе компонентной архитектуры. В этом разделе мы рассмотрим организацию доступа к компоненту COM, предоставляемому в виде динамически подключаемой библиотеки, а затем рассмотрим проблемы, возникающие при попытке использовать компонент COM клиентами, разработанными на разных языках программирования.

Реализация и использование DLL-компонента

После того как выделены обособленные элементы компонентной архитектуры, можно приступить к реализации компонента, хранящегося в DLL-модуле, с последующим использованием функциональности компонента на уровне двоичного кода. Оказывается, осталось проделать не слишком много работы. Файлы, связанные с реализацией компонента в листинге 12.3, могут быть без изменений использованы при проектировании модуля DLL. Поскольку компоновка клиента и компонента теперь осуществляется динамически, файл Export.h не нужен (во время компиляции клиентского кода информация о `CreateInstance()` не требуется). Вместо этого компоновщик использует информацию об экспортируемых компонентом функциях, размещаемую в специальном файле с расширением def. Помимо перечисления этих функций DEF-файл содержит сведения, описывающие DLL-модуль в целом. В нашем случае компонент экспортирует только функцию `CreateInstance()`, поэтому DEF-файл имеет следующее содержание:

```
;
; Файл: ShapeCOMponent.def
; Определение модуля для ShapeCOMponent
;
LIBRARY ShapeCOMponent.dll
DESCRIPTION 'Sample 12.4 (C) Copyright Pyshkin, Evgeny, 2004'
EXPORTS
    CreateInstance @1 PRIVATE
```

При построении кода компонента следует установить ключ транслятора, обеспечивающий генерацию DLL-модуля. В режиме командной строки вызов компилятора может выглядеть следующим образом:

```
cl /LD Component.cpp IIDs.cpp Export.cpp UUID.lib ShapeCOMponent.def
```

Наибольшие изменения затронут клиентский код, т. к. теперь требуется определить клиента таким образом, чтобы функции, экспортируемые компонентом, связывались с клиентским кодом динамически. Для правильной компиляции клиентской программы по-прежнему необходимы файлы Interface.h и IIDs.cpp, являющиеся общими для клиента и компонента. Возможная реализация клиента, использующего спроектированную библиотеку DLL, представлена в листинге 12.4.

Листинг 12.4. Реализация клиента, использующего DLL-компонент

```
/*
 * Файл: Interface.h
 * Определение интерфейсов, поддерживаемых компонентом
 *
 * См. листинг 12.3...
 */

/*
 * Файл: IIDs.cpp
 * Определение GUIDs
 *
 * См. листинг 12.3
 */

/*
 * Файл: Client.cpp
 * Реализация клиента, динамически связываемого с DLL-компонентом
*/
#include <iostream.h>
#include <objbase.h>

#include "Interfaces.h" // Теперь требуется только этот заголовочный файл

/*
 * Определение типа функции, экспортируемой компонентом
 */
typedef IUnknown* ( *CreateInstanceFunctionPtr )();
/*
 * Собственно реализация клиента
 */
```

```
int main() {
    HRESULT hr;

    // Подключиться к модулю компонента
    HINSTANCE hShapeCOMponent = ::LoadLibrary( "ShapeCOMponent.dll" );
    if( hShapeCOMponent == NULL ) {
        cout << "Unable to load \"ShapeCOMponent.dll\"" << endl;
        return 1;
    }

    // Получить адрес функции CreateInstance
    // (динамическая компоновка)
    CreateInstanceFunctionPtr CreateInstance =
        (CreateInstanceFunctionPtr)::GetProcAddress(
            hShapeCOMponent,
            "CreateInstance");
    if( CreateInstance == NULL ) {
        cout << "Unable to find \"CreateInstance\"" << endl;
        return 2;
    }

    // Создать экземпляр компонента и получить указатель на IUnknown
    IUnknown* pIUnknown = CreateInstance();

    // Запрашиваем интерфейс IShape
    IShape* pIShape = NULL;
    hr = pIUnknown->QueryInterface( IID_IShape, ( void** )&pIShape );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IShape
        pIShape->Draw();
        pIShape->Erase();
        // Освобождаем интерфейс IShape
        pIShape->Release();
    }

    // Запрашиваем интерфейс IColorable
    IColorable* pIColorable = NULL;
    hr = pIUnknown->QueryInterface( IID_IColorable,
                                    ( void** )&pIColorable );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IColorable
        pIColorable->SetColor( 2 );
        cout << "Returned color is " << pIColorable->GetColor() << endl;
        cout << "Color is " << pIColorable->ToString() << endl;
```

```
        // Освобождаем интерфейс IColorable
        pIColorable->Release();
    }

    // Освобождаем интерфейс IUnknown
    pIUnknown->Release();

    return 0;
}
```

На рис. 12.5 наглядно представлены элементы компонентной архитектуры в связи с рассматриваемым примером (использование IDL-файлов рассматривается далее в этой главе).

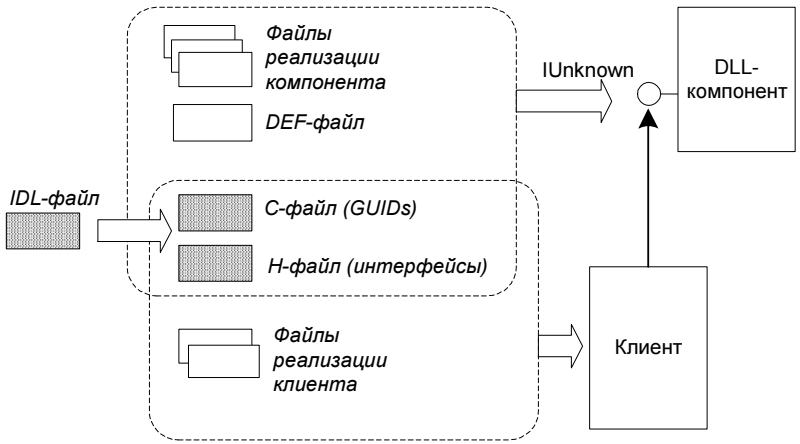


Рис. 12.5. Элементы компонентной архитектуры

В реализации клиента используются функции Win32 SDK `LoadLibrary()` и `GetProcAddress()`. Подробный анализ функций Win32 SDK, обеспечивающих реализацию взаимодействия с DLL-модулями, не входит в задачу этой книги. Отметим, что для упрощения работы с примером здесь мы предполагаем, что DLL-модуль ShapeCOMponent.dll помещен в тот же каталог, что и остальные элементы клиентского приложения. В общем случае, файл динамически подключаемой библиотеки, содержащей код компонента, может располагаться в любом месте. Соответственно, при формировании аргумента функции `LoadLibrary()` для доступа к DLL-модулю требуется указать полное имя файла.

Если в ходе реализации клиента при загрузке DLL-модуля не возникло никаких ошибок, вывод данной программы должен совпадать с результатами исполнения программ, представленных в листингах 12.2—12.3.

Таким образом, мы получили реализацию архитектуры, в рамках которой клиент может работать с различными компонентами без перекомпоновки и перекомпиляции в случае замены или модификации компонентов.

Фабрики классов и регистрация компонентов

Несмотря на то, что пример компонента, представленного в листинге 12.4, является, безусловно, настоящим объектом COM, важно указать на ряд проблем в связи с его использованием. Во-первых, способ создания экземпляра класса `ColoredShape` является уникальной особенностью компонента. Это означает, что каждый компонент должен экспортировать специфические функции, обеспечивающие создание компонента в процессе клиента. Это не слишком удобно, поэтому, очевидно, требуется какое-то унифицирующее решение. Во-вторых, предложенная организация взаимодействия клиента и компонента основывается на том, что клиент должен знать точное местоположение компонента и имя DLL-модуля. Это обуславливает недостаточную гибкость взаимодействия с компонентом и не позволяет клиентскому коду полностью абстрагироваться от физической реализации компонента и его размещения. Кроме того, в общем случае, COM-сервер может располагаться и на удаленном компьютере. Поэтому целесообразно обеспечить такую реализацию компонентной архитектуры, при которой клиент не обязан знать, где именно расположен COM-сервер, а изменение расположения COM-сервера не требует обязательных изменений в клиентском коде.

Таким образом, необходимо решить две задачи:

- стандартизация интерфейса для создания экземпляра компонента;
- использование компонента без привязки к его местоположению (обеспечение прозрачности местоположения).

Вопросы взаимодействия клиента с COM-сервером, размещаемым вне процесса клиента (что включает и вариант удаленного размещения), мы оставим за рамками этой книги. Сосредоточимся на анализе варианта размещения COM-сервера в процессе клиента (т. е. на взаимодействии клиента с компонентом, код которого расположен в DLL-модуле, связываемом с клиентским кодом в процессе исполнения).

Итак, первая проблема в связи с отделением клиента от особенностей реализации компонента заключается в необходимости унифицировать создание COM-объекта в процессе клиента, т. е. нужен стандартный интерфейс создания компонента. Компоненты, единственное назначение которых состоит в создании экземпляров других компонентов, называют *фабриками классов*. В COM стандартным интерфейсом фабрики класса является интерфейс `IClassFactory` (так же как и интерфейс `IUnknown`, он определен в заголовочном файле unknwn.h).

Интерфейс `IClassFactory` задает имена и сигнатуры двух методов:

- метод `CreateInstance()` должен обеспечивать создание экземпляра компонента и запрос требуемого интерфейса, поддерживаемого компонентом;
- метод `LockServer()` должен обеспечивать блокировку компонента для исключения преждевременного его удаления при работе в многопоточной среде.

Листинг 12.5 содержит определение фабрики класса (`ColoredShapeFactory`), обеспечивающей создание экземпляра класса `ColoredShape`. По аналогии с предшествующими примерами, конструктор и деструктор фабрики класса, а также метод `CreateInstance()` снабжены выводом диагностических сообщений для наблюдения за процессом создания экземпляра самой фабрики класса и компонента COM.

Листинг 12.5. Фабрика класса компонента `ColoredShape`

```
/*
 * Файл: ColoredShapeFactory.h
 * Определение фабрики класса ColoredShape
 */

#ifndef _ColoredShapeFactory_h_
#define _ColoredShapeFactory_h_

#include <objbase.h>

class ColoredShapeFactory : public IClassFactory {
public:
    ColoredShapeFactory();
    virtual ~ColoredShapeFactory();

    // Реализация интерфейса IUnknown
    virtual HRESULT __stdcall QueryInterface( const IID&, void** );
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

    // Реализация интерфейса IClassFactory
    virtual HRESULT __stdcall CreateInstance( IUnknown *,
                                              const IID&,
                                              void** );
    virtual HRESULT __stdcall LockServer( BOOL );

private:
   long refCount; // Счетчик ссылок на объект фабрики класса
};
#endif
```

```
/*
 * Файл: ColoredShapeFactory.cpp
 * Реализация методов фабрики класса
 */

#include <objbase.h>
#include <iostream.h>

#include "ColoredShapeFactory.h"
#include "ColoredShape.h"

extern long g_lockCount; // Глобальный счетчик блокировок
                         // COM-сервера
extern long g_objCount;  // Глобальный счетчик активных
                         // объектов COM-сервера

/*
 * Реализация конструктора и деструктора
 */

ColoredShapeFactory::ColoredShapeFactory() {
    cout << "ColoredShapeFactory instance created" << endl;

    refCount = 0;
    InterlockedIncrement( &g_objCount );

}

ColoredShapeFactory::~ColoredShapeFactory() {
    cout << "ColoredShapeFactory instance destroyed" << endl;

    InterlockedDecrement( &g_objCount );
}

/*
 * Реализация методов интерфейса IUnknown
 */

// Запрос интерфейса
HRESULT __stdcall ColoredShapeFactory::QueryInterface(
        const IID& iid, void** ppI ) {

    if( iid == IID_IUnknown ) {
        *ppI = static_cast<IClassFactory*>( this );
    }
    else if( iid == IID_IClassFactory ) {
        *ppI = static_cast<IClassFactory*>( this );
    }
```

```
    else {
        *ppI = NULL;
         return E_NOINTERFACE;
    }

    reinterpret_cast<IUnknown*>(*ppI)->AddRef();
    return S_OK;
}

// Увеличение refCount при создании ссылки на интерфейс
ULONG __stdcall ColoredShapeFactory::AddRef() {

    return InterlockedIncrement( &refCount );
}

// Освобождение интерфейса
ULONG __stdcall ColoredShapeFactory::Release() {

    if( InterlockedDecrement( &refCount ) == 0 ) {
        // Удаляем объект фабрики класса, если больше нет ссылок
        // на его интерфейсы
        delete this;
        return 0;
    }
    return refCount;
}

/*
 * Реализация методов интерфейса IClassFactory
 */
HRESULT __stdcall ColoredShapeFactory::CreateInstance(
        IUnknown *pIUnknownOuter,
        const IID& iid,
        void** ppI ) {
    cout << "ColoredShapeFactory: Creating ColoredShape instance";
    cout << endl;

    if( pIUnknownOuter != NULL )    return CLASS_E_NOAGGREGATION;

    ColoredShape *pShape = new ColoredShape;
    if( pShape == NULL )    return E_OUTOFMEMORY;

    HRESULT hr = pShape->QueryInterface( iid, ppI );
    if( FAILED( hr ) ) {
        delete pShape;
    }
```

```
    return hr;
}

HRESULT __stdcall ColoredShapeFactory::LockServer( BOOL bLock ) {
    if( bLock )    InterlockedIncrement( &g_lockCount );
    else           InterlockedDecrement( &g_lockCount );

    return S_OK;
}
```

Итак, фабрика класса построена. Теперь нужно разобраться, каким образом обеспечивается инициирование объекта самой фабрики класса. Для этого следует разработать файл, который содержит определения экспортируемых DLL-функций, имена и сигнатуры которых стандартизированы спецификацией COM. Эти функции являются, по существу, функциями обратного вызова и вызываются библиотечной функцией COM `CoGetClassObject()`.

Рассмотрим подробнее ту часть определения COM-сервера, которая обеспечивает реализацию экспортируемых функций. В минимальной реализации соответствующий модуль компонента будет иметь следующий вид:

```
/*
 * Файл: ShapeCOMponent.cpp
 * Реализация функций, экспортируемых DLL-компонентом
 */

#include <objbase.h>

#include "ColoredShapeFactory.h"
#include "Interfaces.h"

long g_lockCount( 0 ); // Глобальный счетчик блокировок COM-сервера
long g_objCount( 0 );  // Глобальный счетчик активных объектов
                       // COM-сервера

// Функция, используемая для проверки возможности
// выгрузки компонента
STDAPI DllCanUnloadNow() {
    // Компонент удаляется из памяти, если он не заблокирован
    // каким-либо процессом, и на него нет ссылок
    if( g_lockCount==0 && g_objCount==0 )  return S_OK;
    else                                   return S_FALSE;
}

// Функция создает экземпляр фабрики класса и вызывает метод
// QueryInterface() фабрики класса
```

```
STDAPI DllGetClassObject( const CLSID& clsid,
                          const IID& iid,
                          void** ppI ) {
    HRESULT hr;
    ColoredShapeFactory* pFactory = NULL;

    if( clsid != CLSID_ColoredShape ) return CLASS_E_CLASSNOTAVAILABLE;

    pFactory = new ColoredShapeFactory;
    if( pFactory == NULL ) return E_OUTOFMEMORY;

    hr = pFactory->QueryInterface( iid, ppI );
    if( FAILED( hr ) ) {
        delete pFactory;
    }
    return hr;
}
```

DEF-файл содержит следующие определения экспортируемых компонентом функций:

```
;
; Файл: ShapeCOMponent.def
; Определение модуля для ShapeCOMponent
;
LIBRARY ShapeCOMponent.dll
DESCRIPTION 'Sample 12.5 (C) Copyright Pyshkin, Evgeny, 2004'
EXPORTS
    DllCanUnloadNow @1 PRIVATE
    DllGetClassObject @2 PRIVATE
```

Как явствует из реализации функции `DllGetClassObject()`, чтобы различать GUID классов и GUID интерфейсов, для уникальной идентификации классов в COM используется тип `CLSID` (CLaSs IDentifier). Хотя и `CLSID`, и `IID` имеют, по существу, один и тот же смысл. Идентификатор `CLSID_ColoredShape` должен быть помещен в соответствующий блок объявлений в файле Interfaces.h:

```
/*
 * Предварительные объявления уникальных идентификаторов интерфейсов
 * и классов
 */
extern "C"
{
    extern const IID IID_IShape;
    extern const IID IID_IColorable;
    extern const CLSID CLSID_ColoredShape; // Добавленное объявление
};
```

Определение идентификатора `CLSID_ColoredShape` должно быть помещено в файл IIDs.cpp:

```
/*
 * Файл: IIDs.cpp
 * Определение GUIDs
 */

#include <objbase.h>

/*
 * Определения уникальных идентификаторов интерфейсов
 * (генерируются утилитой uuidgen.exe)
 */
extern "C"
{
    // {c7f4f55a-3a4f-438a-9b50-a0cb5b955073}
    extern const IID IID_IShape =
        {0xc7f4f55a, 0x3a4f, 0x438a,
        {0x9b, 0x50, 0xa0, 0xcb, 0x5b, 0x95, 0x50, 0x73}};

    // {78c71e52-6b12-4e18-ae4d-a104206728c6}
    extern const IID IID_IColorable =
        {0x78c71e52, 0x6b12, 0x4e18,
        {0xae, 0x4d, 0xa1, 0x04, 0x20, 0x67, 0x28, 0xc6}};

    // {ebe6b2f7-7bdd-4f8a-8909-afef5eed0e93}
    extern const CLSID CLSID_ColoredShape =
        {0xebe6b2f7, 0x7bdd, 0x4f8a,
        {0x89, 0x09, 0xaf, 0xef, 0x5e, 0xed, 0x0e, 0x93}};
};
```

Структура полученной программы COM-сервера представлена на рис. 12.6.

Теперь рассмотрим, каким образом решается проблема обеспечения прозрачности местоположения COM-сервера.

Для исключения привязки клиента к имени и расположению компонента DLL, компонент должен быть зарегистрирован в реестре Windows. В этом случае клиент использует не имя DLL-модуля с указанием пути к каталогу на диске, а уникальный идентификатор компонента, зарегистрированного в реестре.

В соответствии с определением, данным в книге [Solomon, Russinovich, 2000], реестр Windows представляет собой системную базу данных, содержащую информацию, необходимую для загрузки и конфигурирования системы. Ре-

естр является хранилищем системных и пользовательских параметров. По своей структуре реестр Windows напоминает структуру логического диска. Он содержит разделы, или ключи (keys), и параметры (values). Каждый раздел имеет древовидную структуру и включает подразделы (subkeys). В связи с поддержкой COM нас интересует один из корневых разделов реестра — раздел `HKEY_CLASSES_ROOT`. Именно в этом разделе содержатся регистрационные данные COM-классов. Для хранения регистрационной информации о классах COM используется подраздел `CLSID` раздела `HKEY_CLASSES_ROOT`. В разделе `CLSID` перечислены GUID всех компонентов, установленных в системе. Для того чтобы облегчить поиск зарегистрированного компонента в реестре, в каждом разделе `CLSID` задается параметр по умолчанию — дружественное имя компонента.

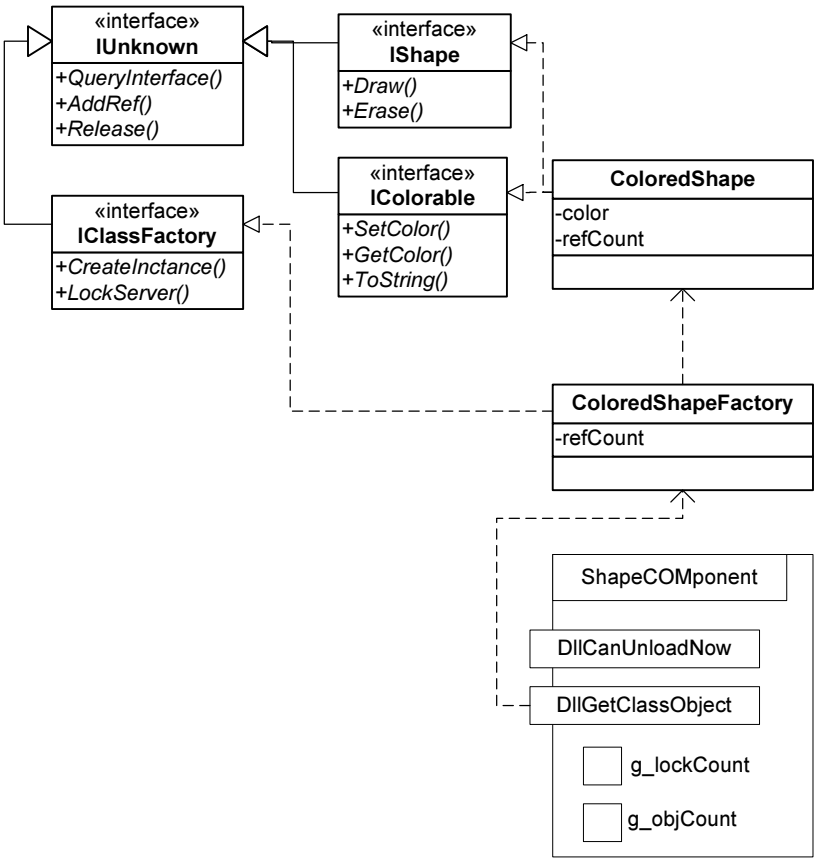


Рис. 12.6. Структура COM-сервера

Для регистрации компонента необходимо:

1. Запустить редактор реестра Windows (в Windows XP — это программа regedit.exe).
2. Открыть раздел `HKEY_CLASSES_ROOT`.
3. В подразделе `CLSID` создать ключ с GUID регистрируемого компонента. В нашем случае — это `CLSID` класса `ColoredShape`, т. е. нужно создать ключ `{ebe6b2f7-7bdd-4f8a-8909-afef5eed0e93}`.
4. Под только что созданным ключом создать подраздел `ProgID`, обеспечивающий хранение программного идентификатора компонента, используемого как альтернативное локальное имя компонента, более удобное для восприятия, чем GUID. Таким образом, `ProgID` — это "дружественное" имя компонента, понятное программисту, уникальность которого не гарантируется. Формат задания `ProgID` следующий: `имя-сервера.имя-класса`. Следуя предписанному формату, в качестве значения параметра по умолчанию нужно задать строку `ShapeCOMponent.ColoredShape`.
5. Для задания пути к DLL-компоненту под ключом `{ebe6b2f7-7bdd-4f8a-8909-afef5eed0e93}` нужно создать раздел `InprocServer32` и в качестве значения параметра по умолчанию задать полный путь к DLL-модулю.

Графически структура информации, относящейся к регистрации компонента `ShapeCOMponent` в реестре, представлена на рис. 12.7.

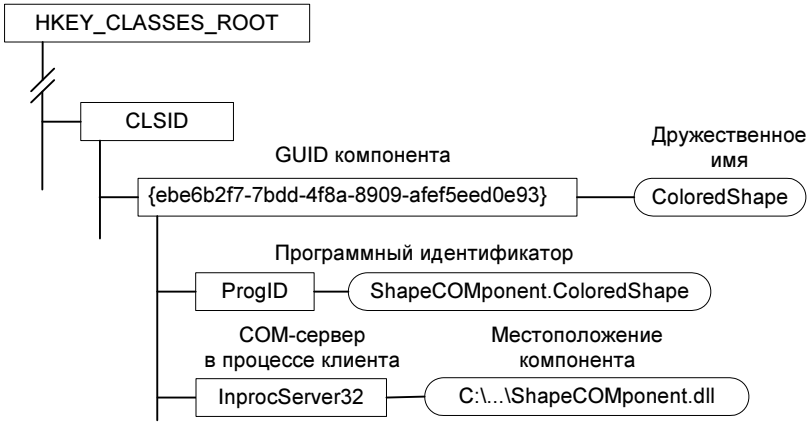


Рис. 12.7. Регистрация COM-сервера ShapeCOMponent.dll в реестре Windows

Теперь рассмотрим код клиента для спроектированного COM-сервера (листинг 12.6).

Листинг 12.6. Клиент, непосредственно использующий фабрику класса

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

#include "Interfaces.h"

/*
 * Реализация клиента
 */
int main() {

    CoInitialize( NULL ); // Инициализация COM

    IClassFactory *pShapeFactory = NULL;

    HRESULT hr;

    // Создание экземпляра фабрики класса и получение указателя
    // на интерфейс IClassFactory
    hr = CoGetClassObject( CLSID_ColoredShape,     // GUID компонента
                           CLSCTX_INPROC_SERVER,  // Сервер в процессе
                                                  // клиента
                           NULL,                  // Нет информации
                                                  // об удаленном сервере
                           IID_IClassFactory,     // Запрашиваемый
                                                  // интерфейс
                           ( void** )&pShapeFactory // Адрес указателя
                                                    // на запрашиваемый
                                                    // интерфейс
                         );
    if( FAILED( hr ) ) {
        cout << "Creating class factory instance failed" << endl;
        return 1;
    }

    // Создаем экземпляр класса ColoredShape и
    // запрашиваем интерфейс IShape
    IShape *pIShape = NULL;
    hr = pShapeFactory->CreateInstance( NULL,
                                        IID_IShape,
                                        ( void** )&pIShape
                                      );

    // Освобождаем интерфейс IClassFactory
    pShapeFactory->Release();
```

```
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IShape
        pIShape->Draw();
        pIShape->Erase();
    }

    // Запрашиваем интерфейс IColorable
    IColorable* pIColorable = NULL;
    hr = pIShape->QueryInterface( IID_IColorable,
                                  ( void** )&pIColorable );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IColorable
        pIColorable->SetColor( 2 );
        cout << "Returned color is " << pIColorable->GetColor() << endl;
        cout << "Color is " << pIColorable->ToString() << endl;
        // Освобождаем интерфейс IColorable
        pIColorable->Release();
    }

    // Освобождаем интерфейс IShape
    pIShape->Release();

    // Завершение работы с COM
    CoUninitialize();
    return 0;
}
```

Последовательность исполнения кода представлена на рис. 12.8.

В результате исполнения клиентской программы на консоль должны быть выведены следующие сообщения:

```
ColoredShapeFactory instance created
ColoredShapeFactory: Creating ColoredShape instance
ColoredShape instance created
ColoredShapeFactory instance destroyed
Shape drawn
Shape erased
Color is set to 2
Returned color is 2
Returned color is green
ColoredShape instance destroyed
```

Вывод клиентской программы позволяет убедиться в том, что представленная на рис. 12.8 последовательность исполнения соответствует действительности.

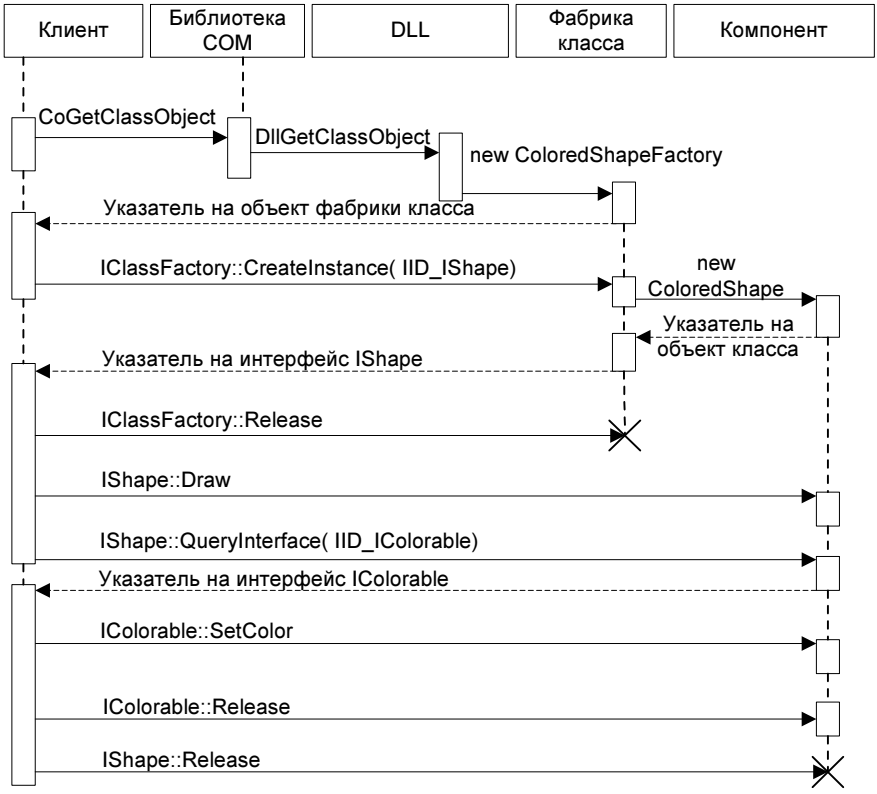


Рис. 12.8. Последовательность выполнения кода при явном запрашивании интерфейса фабрики класса

Метод `CoGetClassObject()` обычно используется тогда, когда требуется получить явный доступ к функциональности, предоставляемой фабрикой класса (например, в том случае, когда в клиентском коде требуется работа с множеством экземпляров компонента). Если же нужен только один экземпляр компонента, реализацию клиента можно существенно упростить (листинг 12.7).

Листинг 12.7. Клиент, неявно использующий фабрику класса

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

#include "Interfaces.h"

/*
 * Реализация клиента
 */
```

```
int main() {

    CoInitialize( NULL ); // Инициализация COM

    HRESULT hr;
    // Создаем экземпляр компонента и
    // запрашиваем интерфейс IShape
    IShape *pIShape = NULL;
    hr = CoCreateInstance( CLSID_ColoredShape,   // GUID компонента
                           NULL,                 // Нет агрегирования
                                                 // компонента
                           CLSCTX_INPROC_SERVER, // Сервер в процессе
                                                 // клиента
                           IID_IShape,           // Запрашиваемый
                                                 // интерфейс
                           ( void** )&pIShape    // Адрес указателя
                                                 // на интерфейс
                         );
    if( FAILED( hr ) ) {
        cout << "Creating class factory instance failed" << endl;
        return 1;
     }

    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IShape
        pIShape->Draw();
        pIShape->Erase();
    }

    // Используя интерфейс IShape, запрашиваем интерфейс IColorable
    IColorable* pIColorable = NULL;
    hr = pIShape->QueryInterface( IID_IColorable,
                                  ( void** )&pIColorable );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IColorable
        pIColorable->SetColor( 2 );
        cout << "Returned color is " << pIColorable->GetColor() << endl;
        cout << "Color is " << pIColorable->ToString() << endl;
        // Освобождаем интерфейс IColorable
        pIColorable->Release();
    }

    // Освобождаем интерфейс IShape
    pIShape->Release();
```

```
    CoUninitialize();
    return 0;
}
```

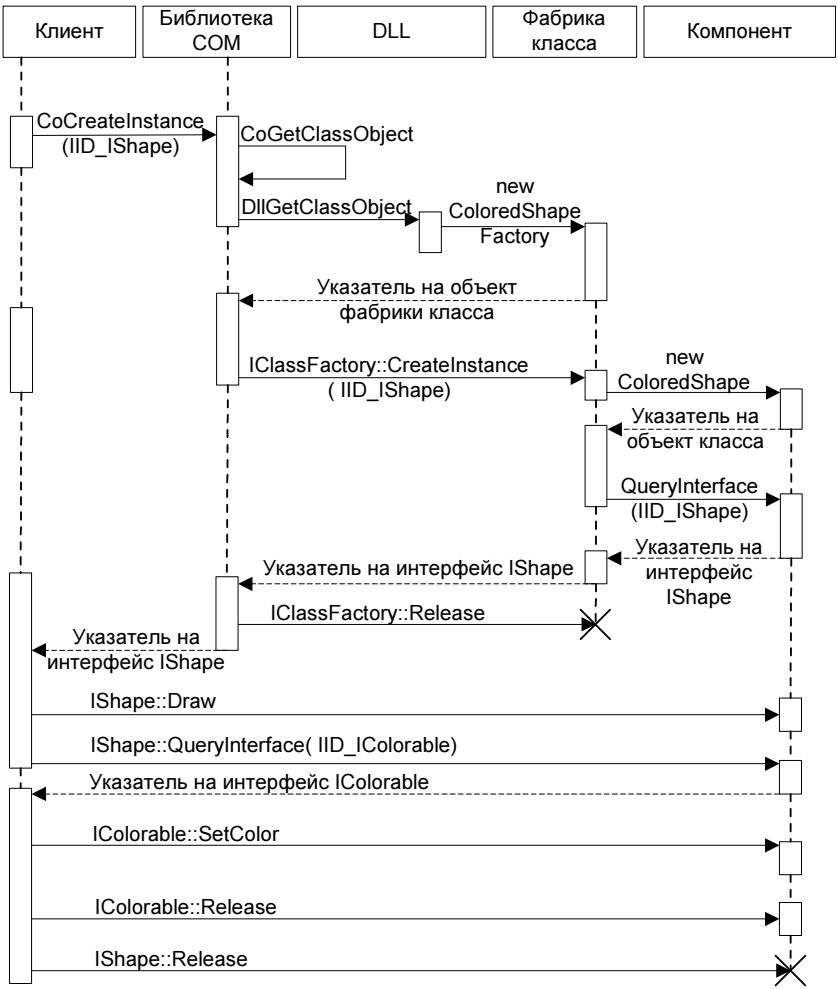


Рис. 12.9. Последовательность исполнения кода при использовании функции `CoCreateInstance()`

В этом примере клиентский код не работает явно с фабрикой класса, сразу получая доступ к одному из интерфейсов компонента. Однако анализ вывода этой программы (который ничем не отличается от вывода предыдущего клиента) показывает, что объект фабрики класса создается, и именно посредством использования интерфейса фабрики класса создается экземпляр компонента. На рис. 12.9 наглядно представлено, что многие действия в связи с

созданием экземпляров фабрики класса и компонента с последующим запросом интерфейса инкапсулированы в библиотечном коде.

Преодоление языковой зависимости

В рассмотренных примерах клиентский код использовал заголовочный файл Interfaces.h, содержащий определения поддерживаемых компонентом интерфейсов, и файл IIDs.cpp, содержащий определения GUIDs этих интерфейсов. Это решение вполне работоспособно до тех пор, пока и клиент, и COM-сервер разработаны на языке C++. Однако одно из принципиальных требований компонентной архитектуры предполагает возможность использования компонентов, разработанных на одном языке программирования, в клиентском коде, разработанном на другом языке программирования. Это означает, что распространение информации об интерфейсах COM-сервера в виде файлов C++ неприемлемо — нужен иной стандартный механизм описания интерфейсов, инвариантный к используемому языку программирования. Таким языком стал язык IDL (Interface Description Language), обеспечивающий возможность описания интерфейсов и классов COM без привязки к конкретному языку программирования.

Общая схема разработки компонента с использованием IDL состоит в следующем.

Сначала создается IDL-файл, содержащий описание всех поддерживаемых компонентом интерфейсов. Наличие в языке описаний интерфейсов директивы `import` позволяет импортировать ранее спроектированные описания и избежать дублирования одних и тех же интерфейсов при описании различных компонентов. В этой книге мы не будем рассматривать правила конструирования IDL-файлов, поскольку читатель всегда может воспользоваться более подробными источниками по компонентному программированию. В листинге 12.8 представлен пример IDL-файла, описывающего интерфейсы компонента `ShapeCOMponent`.

Заметим, что в представленном в листинге 12.8 определении сигнатуры функций интерфейсов приведены в соответствие с требованиями спецификации COM. В частности, все функции возвращают код `HRESULT`, а остальные результаты работы передаются через список параметров. Наиболее существенные изменения затронули функцию `ToString()`, которая теперь изменяет объект `BSTR`, эквивалентный в C++ строке символов `wchar_t`. Использование двухбайтовых символов обусловлено требованиями совместимости компонентов, разработанных на разных языках программирования.

Обратите внимание, что для библиотечного модуля DLL также нужно сгенерировать собственный GUID.

IDL-файл обрабатывается транслятором, который генерирует выходные описания на том языке программирования, на котором разрабатывается реализация компонента, а также двоичный файл с расширением tlb, содержащий описание библиотеки типов. Если компонент используется клиентом, разрабатываемом на C/C++, для обработки IDL-файла можно задействовать транслятор MIDL (Microsoft IDL compiler). При этом выходом транслятора является набор файлов C/C++, содержащих описания интерфейсов и GUIDs, подобные тем, что раньше разрабатывались нами "напрямую" (см. описание файлов Interface.h и IIDs.cpp в листинге 12.3). В частности, для приведенного выше IDL-файла транслятор MIDL сгенерирует файл ShapeCOMponent.h, содержащий определения всех интерфейсов, поддерживаемых компонентом, и файл ShapeCOMponent_i.c, содержащий уникальные идентификаторы всех интерфейсов и классов, определенных в составе компонента. Напомним, что транслятор MIDL формирует также двоичный файл библиотеки типов ShapeCOMponent.tlb (рис. 12.10).

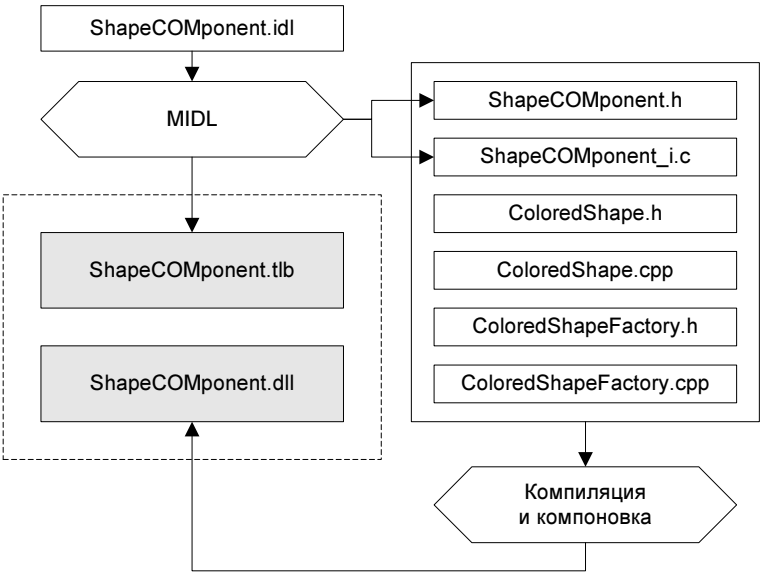


Рис. 12.10. Файловая структура компонента при использовании IDL-файлов

Замечание

Классы, определяемые в составе компонента COM, обычно называют кокласами, и поэтому при их именовании используют префикс `Co` (так, в соответствии с рекомендациями COM, класс `ColoredShape` следовало бы назвать `CoColoredShape`). В рассмотренных примерах мы отошли от этой традиции в связи с тем, что основное назначение данной главы состоит не в освоении тех-

нологий компонентного программирования в том виде, в каком они реализованы в COM, а в формировании у читателя основных представлений о компонентной архитектуре (что, впрочем, не мешает нам использовать для иллюстрации работающую модель компонента COM).

Поскольку вся информация об интерфейсах и классах компонента (равно как и информация об их уникальных идентификаторах) сосредоточена в спроектированном IDL-файле, необходимо исключить из проекта те файлы C/C++, которые совместно использовались клиентом и компонентом. Вместо них будут использоваться файлы, автоматически сгенерированные транслятором IDL (разумеется, изменения нужно внести и в файлы проекта компонента, и в файлы проекта клиента). Наряду с определением интерфейса IDL-файла листинг 12.8 содержит отредактированную с учетом изменений интерфейсов версию файла ColoredShape.h, содержащего определение основного класса компонента. На прилагаемом к книге компакт-диске содержатся полные версии всех файлов COM-сервера.

Листинг 12.8. Описания интерфейсов и классов компонента

```
// Файл: ShapeCOMponent.idl — содержит описания интерфейсов
// Импорт стандартных описаний (в том числе — интерфейса IUnknown)
import "oaidl.idl";
// Интерфейс IShape
[ object,
uuid( c7f4f55a-3a4f-438a-9b50-a0cb5b955073 ),
helpstring( "Plane shape in general" ) ]
interface IShape : IUnknown
{
    HRESULT Draw();
    HRESULT Erase();
};
// Интерфейс IColorable
[ object,
uuid( 78c71e52-6b12-4e18-ae4d-a104206728c6 ),
helpstring( "Colorable object" ) ]
interface IColorable : IUnknown
{
    HRESULT SetColor( [ in ] int color );
    HRESULT GetColor( [ out ] int* pColor );
    HRESULT ToString( [ in, out ] BSTR str );
};
// Компонент
[ uuid( c424ac00-d2c6-41a7-96c4-0e2f0dd9985d ),
version(1.0),
helpstring( "Shape library" ) ]
```

```
library ShapeCOMponent
{
    importlib( "stdole32.tlb" );
    [ uuid( ebe6b2f7-7bdd-4f8a-8909-afef5eed0e93 ) ]
    coclass ColoredShape
    {
        [ default ] interface IShape;
        interface IColorable;
    }
};

/*
 * Файл: ColoredShape.h
 * Определение класса компонента в соответствии с интерфейсами,
 * определяемыми в ShapeCOMponent.idl
 */
#ifndef _ColoredShape_h_
#define _ColoredShape_h_

#include <iostream.h>
#include <objbase.h>
#include <string.h>

#include "ShapeCOMponent.h" // Этот файл сгенерирован MIDL в результате
                            // трансляции IDL-файла

/*
 * Класс в составе компонента
 */
class ColoredShape: public IShape,
                    public IColorable
{
public:
    // Реализация интерфейса IUnknown
    virtual HRESULT __stdcall QueryInterface( const IID&, void** );
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

    // Реализация интерфейса IShape
    virtual HRESULT __stdcall Draw() {
        cout << "Shape drawn" << endl;
        return S_OK;
    }
    virtual HRESULT __stdcall Erase() {
        cout << "Shape erased" << endl;
```

```
        return S_OK;
    }

    // Реализация интерфейса IColorable
    virtual HRESULT __stdcall SetColor( int color ) {
        this->color = color;
        cout << "Color is set to " << color << endl;
        return S_OK;
    }
    virtual HRESULT __stdcall GetColor( int * pColor ) {
        *pColor = color;
        return S_OK;
    }

    virtual HRESULT __stdcall ToString( wchar_t* str ) {
        char temp[ 10 ];
        switch( color ) {
            case 0 : strcpy( temp, "white" ); break;
            case 1 : strcpy( temp, "red" ); break;
            case 2 : strcpy( temp, "green" ); break;
            case 3 : strcpy( temp, "yellow" ); break;
            case 4 : strcpy( temp, "blue" ); break;
            default: strcpy( temp, "unknown" ); break;
        }
        // Копирование строки с преобразованием к строке
        // символов wchar_t (к строке типа BSTR)
        mbstowcs( str, temp, 2*strlen( temp ) );
        return S_OK;
    }

public:
    // Конструктор компонента
    // NB! Обеспечивает инициализацию refCount
    ColoredShape() : refCount( 0 ) {
        cout << "ColoredShape instance created" << endl;
    }

    // Деструктор компонента
    ~ColoredShape() {
        cout << "ColoredShape instance destroyed" <<endl;
    }

private:
    long refCount; // Счетчик ссылок на интерфейсы
    int  color;
};
#endif
```

Для того чтобы COM-сервер можно было использовать, его следует зарегистрировать в реестре Windows. Часть работы по регистрации компонента была уже выполнена ранее. Остается зарегистрировать библиотеку типов ShapeCOMponent.tlb и осуществить привязку информации о библиотеке типов к информации о компоненте COM. Для этого нужно выполнить следующие действия:

1. В подразделе `TypeLib` раздела `HKEY_CLASSES_ROOT` создать раздел `{c424ac00-d2c6-41a7-96c4-0e2f0dd9985d}`, описывающий библиотеку типов. В качестве значения параметра по умолчанию можно задать имя компонента `ShapeCOMponent`.
2. В разделе `{c424ac00-d2c6-41a7-96c4-0e2f0dd9985d}` создать подраздел `1.0`, определяющий номер версии.
3. В только что созданном разделе `1.0` создать подраздел `0` (идентификатор языка).
4. В только что созданном разделе `0` создать три подраздела:
   - подраздел `Win32`: в качестве значения аргумента по умолчанию указывается путь к библиотеке типов;
   - подраздел `FLAGS`: в качестве значения аргумента по умолчанию устанавливается `0`;
   - подраздел `HELPSTRING`: в качестве значения аргумента по умолчанию указывается путь к файлу помощи (если имеется).
5. Под созданным ранее ключом `{ebe6b2f7-7bdd-4f8a-8909-afef5eed0e93}` в дополнение к существующим ключам создать раздел `TypeLib`. В качестве значения аргумента по умолчанию указывается ключ библиотеки типов: `{c424ac00-d2c6-41a7-96c4-0e2f0dd9985d}`.
6. Если DLL-модуль версии компонента, использующей определение IDL-файла (см. листинг 12.8), размещается в другом каталоге по отношению к компоненту, рассмотренному в предшествующих примерах (см. листинги 12.5—12.7), изменить параметр по умолчанию ключа `InprocServer32`, задав новое значение пути к DLL-модулю.

Графически состояние фрагмента реестра после регистрации библиотеки типов представлено на рис. 12.11.

После того как COM-сервер зарегистрирован, можно приступить к испытанию клиентского кода. При разработке клиента на C++ используются файлы ShapeCOMponent.h и ShapeCOMponent_i.c, сгенерированные MIDL в результате трансляции IDL-файла. Исправленный с учетом этих изменений код клиента представлен в листинге 12.9. Ввиду простоты клиентского кода и отсут-

ствия необходимости работы с несколькими экземплярами класса компонента, для создания экземпляра компонента здесь используется функция `CoCreateInstance()`.

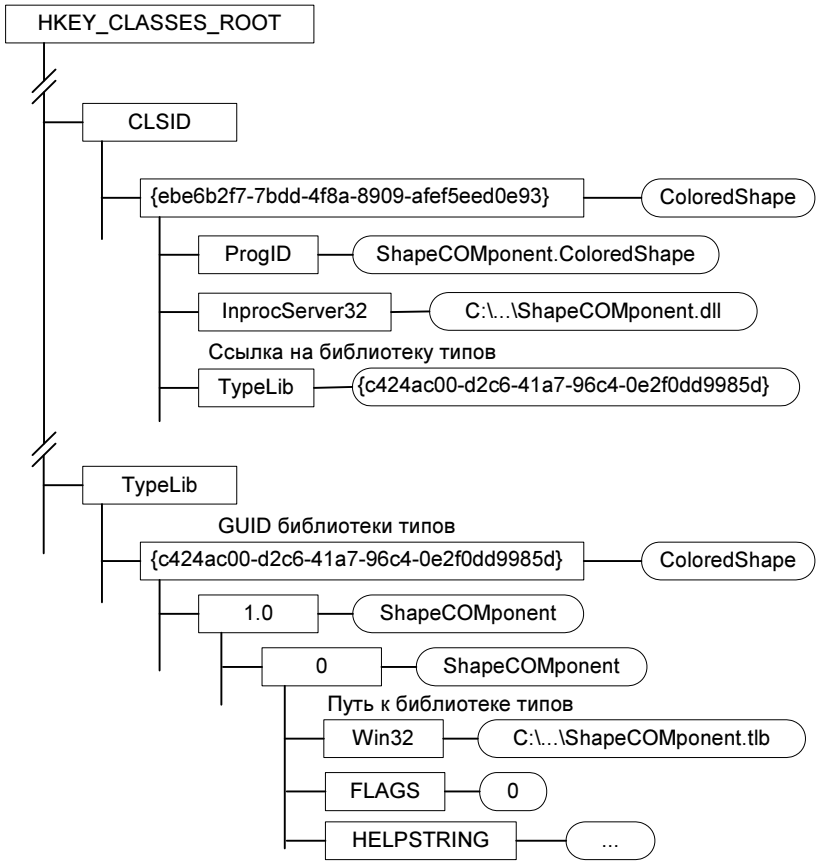


Рис. 12.11. Регистрация библиотеки типов ShapeCOMponent.tlb в реестре Windows

Листинг 12.9. Клиент COM-сервера, построенного с использованием IDL-файла

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

#include <objbase.h>

#include "ShapeCOMponent.h"
```

```
/*
 * Реализация клиента
 */
int main() {

    CoInitialize( NULL );

    HRESULT hr;

    // Создаем экземпляр компонента и
    // запрашиваем интерфейс IShape
    IShape *pIShape = NULL;
    hr = CoCreateInstance( CLSID_ColoredShape,
                           NULL,
                           CLSCTX_INPROC_SERVER,
                           IID_IShape,
                           ( void** )&pIShape );
    if( FAILED( hr ) ) {
        cout << "Creating class instance failed" << endl;
        return 1;
    }

    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IShape
        pIShape->Draw();
        pIShape->Erase();
    }

    // Используя интерфейс IShape, запрашиваем интерфейс IColorable
    IColorable* pIColorable = NULL;
    hr = pIShape->QueryInterface( IID_IColorable,
                                  ( void** )&pIColorable );
    if( SUCCEEDED( hr ) ) {
        // Используем интерфейс IColorable
        pIColorable->SetColor( 2 );

        int color;
        pIColorable->GetColor( &color );
        cout << "Returned color is " << color << endl;

        wchar_t * wstrColor = new wchar_t[ 10 ];
        pIColorable->ToString( wstrColor );

        cout << "Returned color is ";
        wcout << wstrColor << endl;
        delete wstrColor;
```

```
        // Освобождаем интерфейс IColorable
        pIColorable->Release();
    }
    // Освобождаем интерфейс IShape
    pIShape->Release();

    CoUninitialize();
    return 0;
}
```

Нужно ли говорить о том, что, если при регистрации компонента и написании клиента не допущено никаких ошибок, вывод этой программы ничем не отличается от результатов исполнения программ, представленных в листингах 12.6—12.7?!

Как явствует из последнего примера, регистрация компонента в реестре не слишком приятное и легкое занятие. К счастью, большинство компонентов регистрируют и разрегистрируют себя сами посредством специально предназначенных для этого функций, экспортируемых DLL-компонентом: `DLLRegisterServer()` и `DllUnregisterServer()`. С примерами реализации функций, обеспечивающих регистрацию компонентов, читатель может познакомиться в специализированной литературе (см., например, [Rogerson, 1997]). Отметим, что проведенная регистрация компонентов "в ручном режиме", безусловно, небесполезна, поскольку позволяет освоить общие принципы обеспечения прозрачности местоположения компонентов COM и организацию взаимодействия клиента и COM-сервера.

Основные проблемы COM

Материал, рассмотренный выше в данной главе, имеет конспективный характер и позволяет познакомиться лишь с основными механизмами обеспечения взаимодействия компонентов. Однако приведенных сведений достаточно для первоначального анализа технологии компонентного программирования на основе COM. В этом разделе мы обобщим мнения различных программистов о присущих архитектуре COM проблемах.

Несмотря на то, что технология COM, безусловно, является примером успешной и широко используемой технологии, для проектных решений, основывающихся на архитектуре COM, характерна некоторая громоздкость и негибкость. Среди основных причин этого называют, в частности, следующие проблемы.

Хотя взаимодействие объектов через интерфейсы (а не напрямую), безусловно, способствует отделению клиентского кода от особенностей реализации компонента, сама по себе концепция интерфейса не решает всех проблем

изоляции кода. При этом важно отметить, что в рамках интерфейсной архитектуры невозможно наследование реализации, хотя часто наследование реализации выглядит проще, чем наследование интерфейса. Абсолютная закрытость компонента не всегда способствует написанию хороших клиентов.

При этом, вопреки объектно-ориентированной сущности многих элементов, лежащих в основе архитектуры COM, управление объектами основывается на модели "запрос-ответ", которая не отличается большой гибкостью. Это означает, что даже в том случае, когда и клиент, и компонент реализованы в виде строго объектно-ориентированного кода, в основе их взаимодействия лежат не объектно-ориентированные механизмы.

Механизм управления ситуациями, возникающими при использовании COM, на базе значений HRESULT, довольно громоздок и не очень хорошо вписывается в объектно-ориентированную архитектуру. В *гл. 6* довольно подробно проанализированы недостатки и ограничения, присущие реализации обработки ошибок на основе возвращаемых значений. Необходимость определения методов классов COM таким образом, чтобы они возвращали код HRESULT, зачастую усложняет интерфейс методов, поскольку информация передается за счет связывания аргументов и параметров метода (см. исправление метода `ToString()` в предыдущем примере). Кроме того, переработка некоторого существующего класса или библиотеки классов в виде компонента COM требует значительной дополнительной работы. Вписываемость модели обработки исключений в архитектуру COM при этом весьма проблематична.

Уникальные идентификаторы GUID не слишком наглядны. Необходимость регистрации компонентов в реестре Windows затрудняет их распространение и использование.

Создание экземпляров компонентов с помощью специальных функций типа `CreateInstance()` не слишком хорошо сочетается с объектно-ориентированными проектами. Такой механизм оправдан в случае, когда типы экспортируемых объектов неизвестны и должны выясняться по ходу выполнения *(см. информацию о рефлексии в гл. 12)*, однако вряд ли слишком нагляден в случае объектов известного типа.

Использование DEF-файлов для описания ресурсов, экспортируемых компонентом, требует внимания и дополнительной работы.

Использование IDL-файлов для обеспечения языковой независимости требует дополнительной работы программиста или использования специальных средств.

Механизм управления временем жизни компонентных объектов на основе `AddRef()` и `Release()` не вполне прозрачен и может являться источником ошибок. Разработка smart-указателей или классов-оболочек требует дополни-

тельной работы и ведет к усложнению клиентского кода. Кроме того, даже при правильной организации управления временем жизни возможно появление циклических ссылок, а это означает, что при использовании только технологии подсчета ссылок в памяти могут долго оставаться экземпляры компонентов — это может быть критично для серверных компонентов. Нерешенной остается и проблема небезопасных преобразований указателей.

Несмотря на унификацию общих принципов, конкретные действия по проектированию и использованию компонентов на разных языках могут значительно отличаться.

Использование архитектуры COM не позволяет полностью преодолеть проблемы управления версиями, в особенности в тех случаях, когда DLL-компоненты совместно используются несколькими приложениями. В связи с этим распространены следующие два типа ситуаций. Первая ситуация возникает, если для работы некоторого приложения устанавливается обновленная версия DLL. Иногда это приводит к тому, что другие приложения работают с этой обновленной версией некорректно. Информация о версии не является частью GUID и не позволяет различать версии компонентов на уровне глобальной идентификации. Это связано с тем, что для того чтобы клиенты могли использовать новую версию компонента вместо старой без перекомпиляции, требуется неизменность GUID компонента. Вторая ситуация возникает, когда пользователю необходимо работать с разными версиями одного и того же приложения одновременно: при этом возможен конфликт версий, или старые версии просто перестают работать.

Даже рассмотрение простых примеров из данной главы показывает, что установка приложений, работающих с DLL-библиотеками, не слишком проста: файлы записываются в разные места на диске, требуется обязательная регистрация компонентов в реестре Windows. Все это может усложнять работу конечных пользователей, поскольку для установки даже локальных приложений могут требоваться административные полномочия. После деинсталляции приложений в системе могут оставаться некоторые файлы (в частности, DLL).

Поскольку среда клиента полностью изолирована от среды компонента, не удается организовать совместное использование клиентом и компонентом пространств имен и исключений. На пространства имен, определенные в пределах компонента, нельзя сослаться из клиентского кода.

В заключение отметим, что основная задача данного раздела заключалась в представлении фундаментальных концепций, лежащих в основе архитектуры COM, и основных механизмов организации взаимодействия компонентов. Разумеется, коммерческие программы не разрабатываются на основе использования низкоуровневых интерфейсов, подобных тем, что рассматривались в

данном разделе. Современные платформы разработки предоставляют различные средства, ускоряющие и упрощающие проектирование, регистрацию и использование компонентов. В качестве примера можно привести библиотеку ATL (Active Template Library) от Microsoft, имеющую встроенные средства для поддержки основных элементов проектирования компонентов COM, в том числе средства для инстанцирования и регистрации объектов COM. Использование подобных средств существенно уменьшает долю рутинной работы, унифицирует структуру проектов и, как следствие, исключает некоторые классы ошибок, однако не преодолевает всех проблем, присущих архитектуре COM на концептуальном уровне.

12.4. Межъязыковая и межплатформенная интеграция: проблемы и решения

Задачи обеспечения интероперабельности программного обеспечения, децентрализации вычислительных ресурсов и распределенного программирования тесно связаны с задачами межъязыковой интеграции. Это связано с тем, что при организации взаимодействия приложений, выполняемых на разных вычислительных платформах, следует обеспечивать не только платформенную, но и, по возможности, языковую независимость. Неизвестны полностью универсальные и "бесплатные" технологии обеспечения языковой независимости компонентов компьютерных программ, однако ряд концепций доказали свою устойчивость, востребованность и практическую ценность.

До последнего времени в области разработки и развертывания компонентно-ориентированного программного обеспечения конкурировали несколько технологий. Во-первых, это технология COM и ее "производные" (DCOM, COM+), разработанные Microsoft. Во-вторых, это архитектура брокеров объектных запросов (CORBA — Common Object Request Broker Architecture), разработанная и развиваемая консорциумом Object Management Group (OMG). В-третьих, это спецификация Enterprise JavaBeans (EJB) от Sun Microsystems.

Появление и развитие моделей построения и взаимодействия объектно-ориентированных компонентных систем типа COM и CORBA обусловлено тем обстоятельством, что возможностей отдельных объектно-ориентированных языков зачастую недостаточно для разработки распределенных вычислительных систем. С помощью объектных моделей компоненты, исполняемые на различных платформах и разработанные на различных языках, взаимодействуют друг с другом и реализуют бизнес-логику приложения в целом. Таким образом, функции COM и CORBA — это функции промежуточного программного обеспечения [Дубова, 1999]. Для обеспечения взаимодействия объектов и их интеграции в единую среду исполнения распределенного при-

ложения, архитектура промежуточного уровня должна реализовывать несколько базовых принципов:

- независимость от физического размещения объекта (прозрачность местоположения);
- платформенная независимость;
- языковая независимость.

Для отделения функциональности клиента и сервера и в COM, и в CORBA используется принцип взаимодействия через интерфейсы, поддерживаемые компонентом. Клиент полностью изолирован от деталей реализации интерфейса компонентом. Несмотря на это фундаментальное сходство, механизмы реализации компонентной архитектуры COM и CORBA имеют существенные различия.

Как было показано в предыдущем разделе, COM обеспечивает интеграцию компонентов на уровне двоичных кодов. В основе организации взаимодействия с компонентом через его интерфейсы лежит механизм реализации полиморфизма на основе таблиц виртуальных функций, т. е., фактически, механизм, заимствованный из языка C++ *(см. разд. 5.2)*. В связи с этим поддержка различных языков должна опираться на механизм вызовов виртуальных функций, который непосредственно применим только к C++, а для всех остальных языков будет подвергаться определенной предварительной обработке [Дубова, 1999]. Поэтому по числу поддерживаемых языков архитектура COM уступала архитектуре CORBA.

Интеграция на уровне двоичного кода ограничивает применение архитектуры одной аппаратно-операционной платформой. Решения, предоставляемые архитектурой COM, DCOM, COM+, ориентированы преимущественно на системы Microsoft, хотя и существуют разработки по переносу этих решений на другие платформы (например, Unix-ориентированная версия COM, реализация COM для Solaris и др.). В отличие от COM, архитектура CORBA с самого начала создавалась для построения распределенных систем, компоненты которых исполняются в неоднородной сетевой среде.

Независимое от языка определение интерфейсов и в COM, и в CORBA осуществляется с помощью специального языка описания интерфейсов — IDL (Interface Definition Language). При этом оказывается, что различия между языками IDL по версии OMG и Microsoft предопределяют наиболее значительные различия между двумя объектными моделями. Дубова отмечает, что в CORBA язык описания интерфейсов — это важнейшая часть архитектуры, основа схемы интеграции объектов, поскольку все интерфейсы и типы данных определяются на IDL. Таким образом, языковая независимость обеспечивается отображением описаний типов данных на IDL в определения на конкретном языке. Иными словами, язык IDL позволяет формально описать

API, сохранив при этом свободу определить методы на любом языке программирования, который обеспечивает связывание с CORBA. Примерами таких языков являются: Ada, COBOL, C, C++, Smalltalk, Java.

В отличие от языка OMG IDL, язык Microsoft IDL является одним из возможных способов определения интерфейсов компонента (но не единственным). MIDL не определяет общей системы типов, которые могут быть отображены в типы, поддерживаемые различными языками. В рамках архитектуры COM+ обеспечивается, главным образом, двоичная интеграция компонентов, разработанных на двух языках программирования: Visual C++ и Visual Basic. Таким образом, не приходится говорить о полноценной языковой независимости. Распределенная архитектура DCOM (Disributed COM) обеспечивает возможность разработки и использования компонентов, разрабатываемых на разных языках программирования (включая Java и Delphi), однако полноценное взаимодействие компонентов, написанных на разных языках, является непростой задачей.

Спецификация EJB обеспечивает построение объектно-ориентированной компонентной архитектуры на базе языка Java. Межъязыковая интеграция обеспечивается заложенной в модель EJB совместимостью с CORBA. Однако основная идея, заложенная в EJB, заключается все-таки не в межъязыковой интеграции, а в следовании принципу "write once, run anyware[1]", т. е. в обеспечении платформенной независимости архитектуры распределенных вычислений. Модель EJB представляет собой архитектуру построения распределенных приложений масштаба предприятия, которые могут функционировать и как серверные компоненты, и как компоненты программного обеспечения промежуточного слоя.

В 1999 году Microsoft анонсировала завершающий этап создания вычислительной платформы Microsoft .NET, основная задача которой состояла в преодолении проблем, присущих спецификациям COM, DCOM, COM+, обеспечении целостной проектной среды для разработки многокомпонентных приложений, в решении проблем межъязыковой интеграции и поддержке компонентной архитектуры приложений на базе Web-сервисов. К настоящему времени можно констатировать интенсивное развитие и расширение использования платформы Microsoft .NET. Она успешно конкурирует с известными технологиями в области разработки компонентного и распределенного программного обеспечения.

Интеграция на уровне двоичных кодов обеспечивается в Microsoft .NET благодаря следующим концепциям:

- использование управляемого кода и метаданных *(см. гл. 10)*;
- распространение компонентов в виде самоописываемых сборок;

[1] Пиши один раз, исполняй где угодно (*англ.*).

- общая система типов Common Type System (CTS);
- общеязыковая спецификация Common Language Specification (CLS).

В основе обеспечения интероперабельности компонентов, размещенных на различных узлах вычислительной сети, лежит использование протокола SOAP (Simple Object Access Protocol — простой протокол доступа к объекту). Разработанный на базе спецификации XML, протокол SOAP не связан с платформенными и языковыми ограничениями. Однако SOAP — это только протокол передачи данных, и сам по себе не преодолевает всех проблем, на решение которых нацелено промежуточное программное обеспечение [Борк, 2001]. Кроме того, использование SOAP при организации взаимодействия с удаленными компонентами связано с накладными расходами, поскольку на стороне клиента требуется реализация разбора данных XML. Однако использование протокола SOAP действительно улучшает организацию межплатформенного взаимодействия, особенно в тех случаях, когда на первый план выходят возможность использования Internet и простота развертывания. Разработаны средства, обеспечивающие адаптацию CORBA к совместной работе с SOAP и Web-сервисами. Поэтому справедливо заключить, что магистральным направлением развития компонентно-ориентированных и распределенных технологий является совместное использование различных подходов и дальнейшая интеграция различных архитектур, что делает эту область информатики и вычислительной техники, безусловно, довольно сложной, но от этого, наверное, даже более интересной.

12.5. Элементы компонентной архитектуры .NET Framework

Вычислительная платформа .NET развивает и расширяет идеи компонентного программирования, заложенные в основу архитектуры COM. В рамках платформы .NET выделяют три фундаментальные компонентные модели [Platt, 2001].

- Модель взаимодействия управляемых модулей, исполняющихся в среде CLR. Эта модель в наибольшей степени близка к COM, поскольку при ее реализации основной акцент делается на преодоление ряда проблем, присущих именно COM *(см. разд. 12.3)*.
- Модель взаимодействия компонентов Web-приложений на основе архитектуры ASP.NET (Active Server Pages).
- Модель распределенного компонентного программирования, ориентированного на XML Web-сервисы. В рамках этой модели реализуется взаимодействие удаленных компонентов, исполняющихся на разных платформах, на базе протокола SOAP.

Концепции управляемого кода и метаданных были *рассмотрены ранее в гл. 10*. Там же мы обсудили общее устройство исполнительной системы CLR. Поскольку основной акцент в данной главе делается на решении проблем обеспечения языковой независимости компонентов, мы сосредоточим свое внимание на тех элементах архитектуры платформы Microsoft .NET Framework, которые наиболее важны именно в свете концепций межъязыковой интеграции.

В основе обеспечения межъязыковой интеграции лежит использование двух центральных механизмов — общей системы типов (CTS) и обобщенной языковой спецификации (CLS). Однако сначала необходимо рассмотреть организацию взаимодействия компонентов .NET.

Управляемые модули и сборки

В *разд. 10.1* мы рассмотрели концепцию управляемого кода и механизм исполнения управляемого кода, в основе которого лежит использование CLR. Применительно к рассмотрению задач компонентного программирования самое время отметить, что CLR на самом деле работает не с самими управляемыми модулями, а с так называемыми сборками. С одной стороны, сборка (assembly) предоставляет механизм логического группирования управляемых модулей и объединения их с необходимыми дополнительными файлами (например, файлами ресурсов). С другой стороны, в рамках архитектуры .NET сборка является наименьшей единицей распространения, повторного использования кода, реализации политик безопасности и отслеживания версий кода. Модуль сборки содержит сведения о других сборках, на которые есть ссылки из текущей сборки. Таким образом, сборки являются самоописываемыми, поскольку содержат не только всю информацию о модулях, составляющих сборку, но и информацию о связанных сборках. Это означает, что сборка содержит всю информацию, необходимую CLR для ее исполнения, следовательно, не требуется реализовывать хранение сведений о сборках где-нибудь еще (например, в реестре Windows).

Поскольку сборки хранят управляемые модули, взаимодействие между компонентами программного обеспечения, фактически, является взаимодействием управляемых модулей. При этом, в общем случае, может быть вообще не известно, на каком языке программирования был первоначально разработан тот или иной компонент. Таким образом, архитектура .NET обеспечивает возможность прозрачного взаимодействия между типами, определяемыми в разных сборках, в рамках объектно-ориентированной среды.

Рассмотрим пример проектирования простой сборки и последующего использования определенного в ней типа из другой сборки. Для этого создадим библиотеку типов (class library), содержащую единственный тип, предостав-

ляющий функциональность контролируемых по переполнению арифметических операций. Пусть языком реализации библиотечного кода является язык Visual Basic. Исходный текст фрагмента программы на Visual Basic, содержащего реализацию только функции умножения, представлен в листинге 12.10.

Листинг 12.10. Библиотечный компонент на Visual Basic .NET

```
Imports Microsoft.VisualBasic
  Public Class Compute
    ' Метод, осуществляющий контролируемое умножение
    Public Function Mult( ByVal a As Int32, ByVal b As Int32 ) As Int32
      Dim result As Int32
      result = a * b  ' Может генерировать OverflowException
      Return result
    End Function
  End Class
```

Результирующий IL-модуль сохраним в составе сборки с именем CheckedComputation.dll. С точки зрения Windows, полученный файл является обычной динамически подключаемой библиотекой, однако код, размещенный в данной библиотеке, требует для своего исполнения обобщенной среды CLR.

Примечание

В примере кода листинга 12.10 очевидно, более логично было бы определить метод `Mult()` статическим (в Visual Basic для этой цели используется спецификатор `Shared`), поскольку его реализация явно не затрагивает состояние объекта класса `Compute`. Однако здесь в методических целях метод реализован в качестве экземплярного с тем, чтобы в клиентском коде нужно было создать экземпляр класса `Compute`.

Теперь приведем пример клиентской программы на C#, использующей спроектированный "библиотечный" код (листинг 12.11).

Листинг 12.11. Клиент на C#, использующий компонент `CheckedComputation`

```
using System;
using CheckedComputation; // Подключаем сборку, содержащую класс Compute

namespace SafeMultiplicationClient {
    class Client {
        static void Main() {
            int a, b;   // Сомножители
            int result; // Произведение
```

```
            try {
                // Ввод данных
                Console.Write( "Enter 1st multiplier:" );
                a = Int32.Parse( Console.ReadLine() );
                Console.Write( "Enter 2nd multiplier:" );
                b = Int32.Parse( Console.ReadLine() );

                // Создание экземпляра класса Compute
                Compute c = new Compute();

                // Вызов метода класса Compute и вывод результата
                result = c.Mult( a, b );
                Console.WriteLine( "Result is {0}", result.ToString() );
            }
            catch( FormatException e ) {
                // Обработка ошибки ввода
                Console.WriteLine( "Error. Not integer value" );
            }
            catch( OverflowException e ) {
                // Обработка переполнения при умножении
                Console.WriteLine( "Overflow. Result is too large" );
            }
        }
    }
}
```

В клиентском коде класс `Compute` используется точно так, как если бы он был определен непосредственно в клиентском модуле. Взаимодействие между компонентами обеспечивается помещением в клиентскую сборку ссылки на сборку, содержащую код класса `Compute`. Как следует из примера, даже исключение, сгенерированное функцией `Mult()`, может быть перехвачено в клиентском коде. Несмотря на то, что иллюстративные возможности столь компактного примера довольно ограничены, можно отметить, что, в отличие от COM, компоненты взаимодействуют напрямую, а не опосредованно. Для создания экземпляра класса `Compute` не требуется каких-либо специфических действий: он создается точно так же, как и прочие объекты. Таким образом, сборки позволяют разделить понятия повторного использования компонента и управления версиями на логическом и физическом уровнях представления (рис. 12.12).

Возможность прямого взаимодействия клиента и внешнего компонента обеспечивается тем, что в рамках архитектуры .NET клиент не изолирован от компонента. Используя метаданные компонента, клиент может получить информацию о предоставляемой компонентом функциональности. Снижается

значимость интерфейсов как средства взаимодействия с компонентами. Таким образом, повторное использование компонентов возможно не только на уровне интерфейсов, но и на уровне реализации.

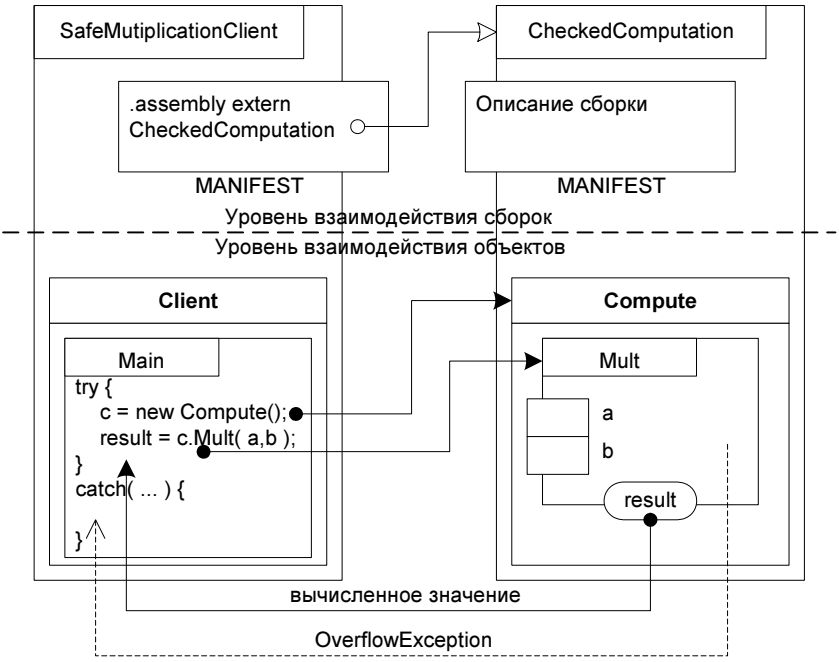


Рис. 12.12. Взаимодействие сборок `SafeMultiplicationClient` и `CheckedComputation`

В целом, взаимодействие клиента и компонента основывается на объектно-ориентированной модели управления действиями и данными. Нет необходимости использовать механизмы, основанные на анализе кодов HRESULT. Вместо этого используется поддерживаемая исполнительной средой модель, основанная на генерации и обработке исключений.

Поскольку сборки являются самоописываемыми, нет необходимости регистрировать управляемые модули в реестре Windows, а развертывание приложений .NET зачастую сводится просто к копированию файлов.

Наличие метаданных исключает необходимость файлов описаний экспортируемых ресурсов (DEF-файлов).

Поскольку создание экземпляров компонента не отличается от создания экземпляров клиентских классов, не требуется использования функций типа `CreateInstance()`. Управление временем жизни компонента основывается на использовании механизма сборки мусора *(см. разд. 10.3)*, поэтому нет необходимости в явно вызываемых методах типа `AddRef()` и `Release()`.

В основе логической модульности при реализации компонентов .NET лежит концепция пространств имен, которые предоставляют механизм логического группирования программных объектов, ортогональный по отношению к сборкам. Так, типы, принадлежащие одному и тому же пространству имен, могут быть реализованы в нескольких сборках, и наоборот, одна сборка может содержать несколько пространств имен.

В примере, представленном листингами 12.10—12.11, речь идет о сборке, используемой одним клиентом. Фактически, используемая клиентом сборка является неотъемлемой частью клиентского приложения. Это означает, что сборка CheckedComputation.dll развертывается в том же каталоге, что и само клиентское приложение. Такие сборки обычно называются сборками с закрытым развертыванием (privately deployed assemblies). Отсутствие конфликтов между различными DLL обеспечивается тем, что каждое приложение использует свои "собственные" DLL, размещаемые вместе с остальными файлами приложения. Поскольку все типы компонента определены в области видимости сборки, приложение всегда привязывается именно к тем типам, с которыми было скомпилировано. CLR не позволяет использовать одноименный тип из другой сборки. Обратите внимание, что это разительно отличается от модели развертывания типов в COM, где зарегистрированные в системном реестре типы доступны любому приложению, исполняющемуся на данном компьютере.

Однако некоторые сборки используются многими приложениями. В этом случае модель с закрытым развертыванием не подходит. Для реализации доступности сборки со стороны многих приложений, она помещается в некоторый общеизвестный каталог, автоматически проверяемый CLR при обнаружении ссылки на сборку. Этот каталог называют глобальным хранилищем сборок (Global Assembly Cache, GAC). В связи с этим необходим механизм уникальной идентификации сборок. В отличие от механизма идентификации компонентов COM, основанного на использовании GUID, уникальная идентификация сборки обеспечивается четырьмя атрибутами, которые вместе образуют так называемое строгое (strong) имя сборки:

- имя файла (без расширения);
- номер версии;
- идентификатор регионального стандарта;
- маркер открытого ключа.

Для уникальной идентификации сборок используется стандартный криптографический механизм, основанный на паре из закрытого и открытого ключей. По этой причине процесс уникальной идентификации сборки перед ее помещением в GAC называют подписанием сборки. С подробностями процесса подписания сборок и их инсталляции в GAC читатель может ознако-

миться в специализированной литературе по программированию на платформе .NET. Здесь отметим лишь, что помещение в перечень атрибутов уникальной идентификации сборки информации о версии сборки позволяет преодолеть одну из самых неприятных проблем архитектуры COM: загрузчик операционной системы не различает версии динамических библиотек *(см. разд. 12.3)*. В отличие от компонентов COM, сборки разных версий, загружаемые CLR, фактически являются разными сборками. Это позволяет без проблем работать с различными версиями "одного и того же" компонента даже в том случае, когда эти версии используются разными частями одного клиентского приложения [Дубовцев, 2004].

Общая система типов .NET

При проектировании языка программирования разработчики определяют семантику конструкций языка в строгой привязке к синтаксису языка (в конечном счете это определяет понятие "синтаксически-ориентированная трансляция"). При этом наименования встроенных типов, используемых в разных языках, могут даже при внешнем сходстве воплощать различные абстракции. В связи с этой неуниверсальностью системы типов, чаще всего встречаются следующие ситуации [Пышкин, 2004-1].

- *Для типов, соответствующих сходным абстракциям, но имеющим различное представление на уровне двоичного кода, используется идентичный синтаксис.* Например, спецификатор `int` в языке C# определяет 32-разрядное целое со знаком, а то же самое служебное слово в C++ используется для представления стандартного целого, которое, в зависимости от платформы, может соответствовать 16-разрядному или 32-разрядному двоичному коду.
- *Для типов, определяющих одинаковые абстракции данных, используется различный синтаксис.* Например, для представления булевского типа в языке C# используется спецификатор `bool`, а в языке Java — спецификатор `boolean`.
- *Одноименные типы имеют разную семантику.* Например, тип `char` в Java предназначен для работы с символами, представленными в 16-разрядном формате Unicode, в то время как в C++ тот же синтаксис используется для работы с символами кода ASCII, представляемыми в C++ в виде 8-разрядного целого со знаком.
- *Некоторые типы поддерживаются одними языками и не поддерживаются другими языками.* Так, например, Visual Basic и Java не поддерживают целых типов без знака длиной более 1 байта, в то время как в других языках (C++, C#) возможность работы с беззнаковыми типами есть; только в некоторых языках существует встроенная поддержка для работы с боль-

шими нецелыми величинами (например, денежными), требующими особой точности представления (в C# это тип `decimal`); стандартный вариант языка C не поддерживает явно булевский тип и т. д.

Таким образом, жесткая привязка концепций данных к выражающему эти концепции синтаксису также является фактором, затрудняющим взаимодействие компонентов, разработанных на разных языках программирования.

Альтернативная возможность заключается в определении базовых абстракций данных без проецирования их на определенный синтаксис. Если эти абстракции обладают достаточной полнотой и общностью, они могут быть использованы разными языками. В этом случае синтаксис и семантика языковых конструкций (по крайней мере, в связи с представлением типов данных) отделены друг от друга, что позволяет определять разные языки на базе одних и тех же фундаментальных абстракций [Chappell, 2001]. Собственно, в представлении этой общей базы системы типов и состоит основное назначение CTS (рис. 12.13).

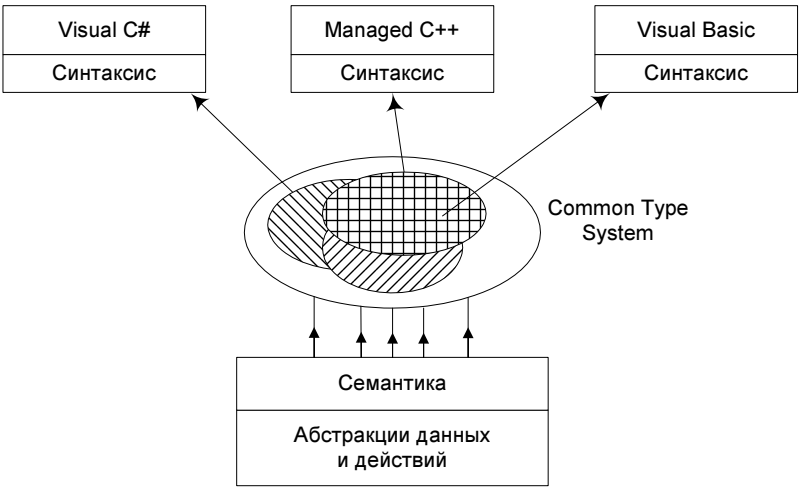


Рис. 12.13. CTS как инструмент определения общих абстракций данных

CTS определяет инфраструктуру типов, обеспечивающих межъязыковую интеграцию, безопасность и однозначность преобразований типов. При этом в основе системы типов CTS лежит объектно-ориентированная модель данных, позволяющая обеспечить реализацию выражаемых CTS общих абстракций в разных языках (возможно, с использованием различного синтаксиса). CTS устанавливает правила, которым необходимо следовать для того, чтобы объекты, разработанные на одном языке, могли прозрачно взаимодействовать с объектами, реализованными на другом языке. Как фундаментальная часть

исполнительной среды платформы .NET Framework (CLR), CTS определяет не универсальный синтаксис, а общий набор типов (рис. 12.14). Вероятно, единственным языком .NET, который поддерживает систему типов CTS полностью, является язык IL *(см. разд. 10.1)*.

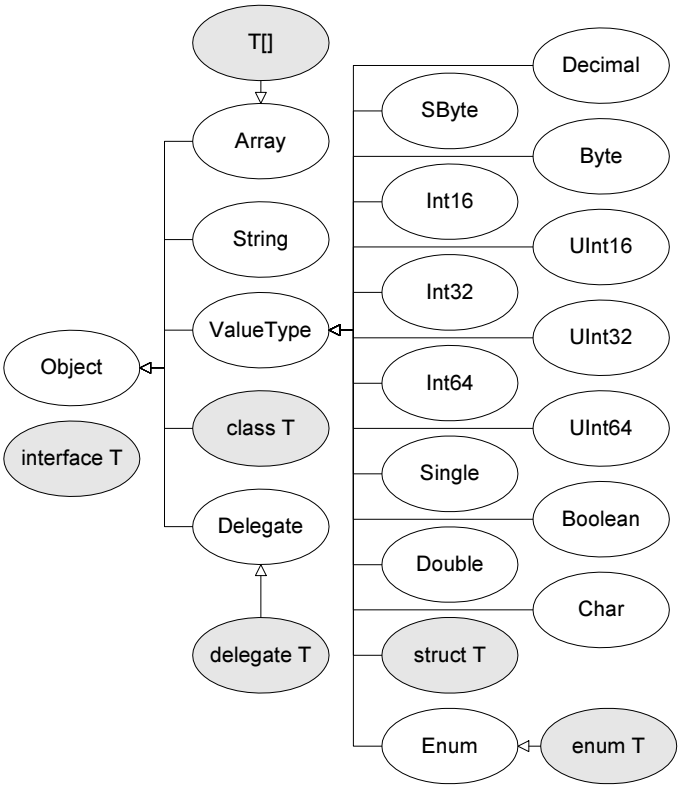


Рис. 12.14. Общая система типов CLR/CTS

Общая система типов определяет две общих категории типов: размерные типы (корнем которых в иерархии типов FCL является тип `System.ValueType`) и ссылочные типы *(см. разд. 10.2)*. Все типы CLR/CTS определены в библиотеке FCL в пространстве имен `System`. Это означает, что при программировании на любом языке, поддерживающем .NET, программист вместо встроенных типов языка имеет возможность явно использовать типы FCL. Элементы, представленные на рис. 12.14 с литерой `T`, соответствуют типам, определяемым пользователем. Из иерархии типов видно, что все пользовательские типы (за исключением интерфейсов) неявно унаследованы от других типов CTS, образуя однокоренную иерархию типов. Например, определение перечисления с использованием служебного слова `enum`, фактически, означает оп-

ределение типа, производного от типа `System.Enum`, который, в свою очередь, является производным от `System.ValueType`.

Общеязыковая спецификация

Для обеспечения прозрачного взаимодействия объектов независимо от языка их реализации объекты должны использовать только те возможности, которые являются общими для всех языков реализации. В связи с этим в рамках архитектуры .NET определена общеязыковая спецификация (CLS), устанавливающая минимальный набор общих свойств языков, необходимых для интеграции с CLR. Иными словами, чтобы использовать тип, доступный из другого языка, нужно использовать только те возможности языка, которые доступны и в других языках. Это означает, что общая система типов предоставляет гораздо больше возможностей, чем CLS, и вы можете использовать все эти возможности, если вопросы межъязыковой интеграции компонентов в вашем случае неактуальны. Однако если необходимо, чтобы разрабатываемый тип был доступен компонентам, разработанным на другом языке, следует использовать только те возможности, которые определяются в CLS.

CLS образуется как подмножество интерфейсов, которые являются общим для CLS-совместимых языков .NET, реализующих эти интерфейсы. Язык .NET должен обеспечивать возможность выступать в роли клиента любого CLS-совместимого API (в английской традиции используется термин CLS consumer). Примерами таких языков являются языки JScript .NET и COBOL .NET. Некоторые языки обеспечивают возможность проектирования CLS-совместимого API (т. е. являются CLS producers). Заметим, что не все типы FCL являются CLS-совместимыми (CLS-complaint).

Компиляторы .NET обычно предоставляют возможность контроля CLS-совместимости разрабатываемых типов, если это требуется. Ниже приводится пример типа, определяемого на языке C#, относительно которого необходимо обеспечить контроль CLS-совместимости:

```
using System;

[ assembly: CLSCompliant( true ) ] // Атрибут указывает на необходимость
                                   // контроля CLS-совместимости типа
                                   // в ходе трансляции
public class TestCompliance {

    public unsigned SomeMethod() { // Вызовет сообщение об ошибке,
                                   // поскольку беззнаковое целое
                                   // поддерживается не всеми языками
       // ...
    }
```

```
    public void SOMEMETHOD() { // Вызовет сообщение об ошибке,
                               // поскольку не все языки обеспечивают
                               // различение идентификаторов, набранных
                               // в разном регистре
        // ...
    }
}
```

В данном примере тип C# `unsigned` (который соответствует типу CTS `System.UInt32`) не является CLS-совместимым. Аналогично, спецификация CLS не поддерживает идентификаторы, различающиеся только регистром. В табл. 12.1 содержится перечень типов CTS, определенных в библиотеке FCL. Для каждого типа указано, является ли он CLS-совместимым, а также приведены спецификаторы соответствующих встроенных типов некоторых языков программирования (прочерк означает, что язык не поддерживает такой тип).

Таблица 12.1. *Типы CTS и CLS-совместимость типов*

Типы CTS	CLS-совместимость	Типы C#	Типы VB.NET	Типы Java
System.Boolean	да	bool	Boolean	boolean
System.Char	да	char	Char	char
System.SByte	нет	sbyte	-	-
System.Int16	да	short	Short	short
System.Int32	да	int	Integer	int
System.Int64	да	long	Long	long
System.Byte	нет	byte	Byte	byte
System.UInt16	нет	ushort	-	-
System.UInt32	нет	uint	-	-
System.UInt64	нет	ulong	-	-
System.Single	да	float	Single	float
System.Double	да	double	Double	double
System.Decimal	да	decimal	Decimal	-
System.Object	да	object	Object	Object
System.String	да	string	String	String

В связи с различиями поддержки типов CTS разными языками программирования возникает вопрос о том, какие типы лучше использовать в своих про-

граммах: типы FCL или встроенные типы языка. Как было отмечено выше, некоторые типы могут однозначно проецироваться на типы FCL, некоторые — нет. Спецификация C# содержит рекомендацию преимущественно использовать встроенные типы *as a matter of style*[1], однако такой подход критикуется некоторыми специалистами, в авторитете и квалификации которых вряд ли можно сомневаться. Рихтер приводит следующие доводы в пользу использования типов FCL [Richter, 2002].

- Во-первых, в разных языках могут отличаться реализации одноименных типов. Из того, что тип `long` в C# интерпретируется как 32-разрядное целое, не следует идентичная интерпретация в другом языке, где `long` может соответствовать, скажем, типу `System.Int64`. Представления некоторых типов могут вообще зависеть от вычислительной платформы (например, уже упоминавшийся тип `int` в C++). Таким образом, программист, приступающий к изучению кода, написанного на новом для себя языке, может неверно истолковать некоторые определения.
- Во-вторых, типы FCL могут содержать методы, использование которых с объектами встроенных типов (проецируемых на типы FCL) может выглядеть не вполне естественно.

Второй довод иллюстрируют следующие фрагменты кода на C#:

```
string s;
s = Console.ReadLine(); // Чтение строки, которая должны быть целым
                        // числом
int value1;
value1 = Int32.Parse( s ); // Корректно, но выглядит не вполне
                           // естественно, т. к. Parse(),
                           // согласно спецификации, возвращает
                           // объект типа Int32, а не int
Int32 value2;
value2 = Int32.Parse( s ); // Корректно, выглядит естественно

// ...

// Обратная задача: преобразуем целое в строку
string s1 = value1.ToString(); // Корректно, но выглядит не вполне
                               // естественно, т. к. int — встроенный,
                               // а встроенные типы не имеют методов:
                               // работает механизм упаковки/распаковки
                               // (см. раздел 10.2)
```

[1] С точки зрения стиля [программирования] (*англ.*).

```
s1 = value2.ToString();// Корректно, выглядит естественно, т. к.
                       // value2 — объект типа Int32 (а не int)
```

Несмотря на обоснованность мнения Рихтера, многие разработчики (вероятно, просто в силу привычки) используют встроенные типы.

12.6. Вопросы и упражнения

1. Объясните, почему в рамках архитектуры COM взаимодействие компонентов осуществляется через интерфейсы. Является ли такая форма взаимодействия компонентов обязательным элементом компонентной архитектуры?
2. Формулировка задачи основывается на примерах из книги [Добрынин, 2003]. Определите интерфейсы COM `IPub` (публикация) и производные от него интерфейсы `IBook` (книга) и `IJournal` (журнал). В интерфейсе `IPub` определите три метода: `SetTitle` (позволяет задать название публикации), `SetYear` (позволяет задать год публикации), `GetInfo` (позволяет получить информацию о публикации). В интерфейсе `IBook` определите метод `SetAuthor` (позволяет задать имя автора книги). В интерфейсе `IJournal` определите метод `SetNumber` (позволяет задать номер журнала). Определите компонентный класс `CoBook`, реализующий интерфейс `IBook`, и компонентный класс `CoJournal`, реализующий интерфейс `IJournal`. При этом метод `GetInfo()` должен формировать строку, содержащую все известные сведения о публикации. Определите фабрики обоих классов. Определите функции, экспортируемые компонентом COM (не забудьте DEF-файл). Зарегистрируйте спроектированный компонент COM в реестре Windows. Напишите клиентскую программу, создающую набор разнообразных публикаций и распечатывающую на консоль сведения обо всех созданных публикациях.
3. На языке программирования C# спроектируйте сборку .NET, реализующую функциональность, подобную функциональности компонента COM из упражнения 2. Напишите клиентскую программу. Обсудите с коллегами отличия взаимодействия объектов .NET и объектов COM.
4. Рассмотрите реализацию компонента .NET из предыдущего упражнения. Использовали ли Вы типы CTS или встроенные типы языка? Является ли спроектированный компонент CLS-совместимым? Если нет, то внесите необходимые исправления. Напишите клиентскую программу на Managed C++, использующую компонент, разработанный на языке C#.

Заключение

Из "неотъемлемых прав читателя":
Право не читать.
Право читать, перепрыгивая через страницы.
Право не окончить книгу.
Право перечитывать заново.
Право читать что угодно.
Право читать где угодно.
Право читать с любой страницы.

Даниэль Пеннак[1]

Вместо заключения я хотел бы адресовать слова благодарности моим студентам. Многое в этой книге является результатом опыта проведения занятий с вами. Ваше внимание и невнимание, ваши вопросы и ответы во многих случаях становились побудительными мотивами к подробному рассмотрению отдельных разделов курса и помогали формированию курса в целом.

Отдельная благодарность тем, кто добрался до этого заключения, внимательно прочитав большинство предшествующих глав. Надеюсь, что обнаруженные вами ошибки и недостатки помогут дальнейшему совершенствованию книги.

Те же читатели, которые обращаются к заключению до того, как освоили остальные главы, вправе утешиться процитированными правами любого читателя (хотя, может, это просто такая привычка — начинать чтение с выяснения того, чем все закончилось).

Возвращаясь к тезису, упомянутому во введении, отмечу, что, безусловно, многие важные детали остались за рамками этой книги. Но если читателям захотелось подробностей, я бы считал основную задачу книги выполненной. Надеюсь, что издания из довольно обширного списка использованных источников будут интересны студентам, стремящимся разобраться в наиболее сложных вопросах практического программирования и расширить профессиональный кругозор.

---

[1] Daniel Pennac. Comme un roman (*пер. с франц.* — Е. П.).

ПРИЛОЖЕНИЯ

Приложение 1. Примеры заданий практикума

Приложение 2. Введение в потоковый ввод-вывод C++

Приложение 3. Описание компакт-диска

ПРИЛОЖЕНИЕ 1

Примеры заданий практикума

Помещенные в этом приложении примеры заданий для практикума представляют собой последовательность тематически связанных между собой заданий, позволяющих студенту постепенно осваивать основные концепции и механизмы объектно-ориентированного программирования.

Практикум. Задание № 1

Цель задания

Целью практикума является выработка у студентов понимания назначения и принципов применения таких элементов объектной модели, как абстрагирование и инкапсуляция. Выполнение этого задания предваряет подробное рассмотрение структуры и элементов объектно-ориентированной программы.

Основные умения, которые должны быть продемонстрированы студентом

- ❒ Умение определить класс C++ и организовать открытый интерфейс класса.
- ❒ Умение обеспечить грамотное документирование программного текста.
- ❒ Умение объяснить смысл отдельных элементов объектной модели и проиллюстрировать свои объяснения на примере разрабатываемой программы.
- ❒ Умение разработать комплекс обоснованных тестовых примеров для проверки корректности реализации.

Постановка задачи

Создайте класс `Rational`, воплощающий абстракцию рациональных чисел, представляемых в виде дробей с числителем (numerator) и знаменателем (denominator). Инкапсулируйте в классе `Rational` функции-члены класса, реализующие основные действия над рациональными числами (сложение, вычитание, умножение и деление). Не забудьте о необходимости сокращения дробей в ходе вычислений. Определите функцию печати рационального числа. Продумайте способ задания рационального числа, представляемого переменными-членами класса (нужно ли ограничить доступ к ним извне?). Какие еще методы класса Вы могли бы предложить?

Напишите тестирующую программу для проверки спроектированного класса.

Оформление результатов выполнения упражнения

Предоставьте преподавателю на проверку комплект следующих документов:

1. Данное задание.
2. Текст программы.
3. Описание открытого интерфейса разработанного класса.

Описание открытого интерфейса разработанного класса представляет собой своего рода руководство пользователя класса. В описании открытого интерфейса должны содержаться ответы на следующие вопросы: как создать экземпляр данного класса (объект данного типа); какие методы можно использовать и как для манипуляции с данными-членами класса. Дополнительно поясните Ваши проектные решения в связи с инкапсуляцией.

Практикум. Задание № 2

Цель задания

Целью практикума является дальнейшее изучение основных элементов объектной модели и организации объектно-ориентированной программы на C++. Задача, решаемая в этом упражнении, является развитием задачи предыдущего упражнения.

Основные умения, которые должны быть продемонстрированы студентом

- ❒ Умение перегрузить операции для работы с данными определяемого типа.
- ❒ Умение правильно организовать открытый интерфейс класса.

- Умение обеспечить реализацию иерархии исключений и обработку ошибочных ситуаций с применением механизма генерации и обработки исключений.
- Умение разработать комплекс обоснованных тестовых примеров для проверки корректности реализации, включая моделирование ошибочных ситуаций.

Постановка задачи

Усовершенствуйте класс `Rational`, обеспечив реализацию следующей функциональности:

- инициализация рационального числа значением целого типа;
- перегруженные арифметические операции для выполнения основных арифметических действий над рациональными числами;
- перегруженные операции >> и << для ввода и вывода рациональных чисел *(информацию о перегрузке операций ввода-вывода см. в приложении П2)*;
- преобразование рационального числа в строковое представление;
- преобразование из типа `Rational` к типу `double`.

Встройте в свой код механизм обработки исключений. Для этого создайте иерархию исключений "от общего к частному" и используйте блоки генерации и обработчики исключений.

Напишите тестирующую программу для проверки корректности разработанных определений, включая проверку механизма генерации и обработки исключений.

Оформление результатов выполнения упражнения

Предоставьте преподавателю на проверку комплект следующих документов:

1. Данное задание.
2. Текст программы.
3. Описание открытого интерфейса спроектированного класса, ориентированное на программистов, использующих класс, с указанием типов исключений, генерируемых функциями-членами класса.

Практикум. Задание № 3

Цель задания

Целью практикума является изучение отношения агрегирования и элементов проектирования конкретных контейнерных классов.

Основные умения, которые должны быть продемонстрированы студентом

- Умение определить отношение агрегирования классов.
- Умение организовать логическую структуру проекта с использованием пространств имен.
- Умение обеспечить обработку ошибочных ситуаций с применением механизма генерации и обработки исключений.

Постановка задачи

С использованием класса `Rational` создайте класс `RationalList`, реализующий концепцию списка рациональных чисел.

По крайней мере, предусмотрите следующие методы:

- конструкторы, с помощью которых можно инициализировать список. Возможны, например, два способа инициализации списка: инициализация пустого списка; инициализация списка, элементами которого становятся элементы передаваемого массива рациональных чисел;
- занесение данных в начало (конец) списка;
- извлечение данных из списка (с удалением и без удаления);
- поиск данных в списке;
- печать элемента списка или всего списка в выходной поток.

Поместите все нелокальные имена, за исключением `main()`, в пространства имен. Обеспечьте отделение объявления пространства имен как общей среды для разрабатываемых функциональных элементов проекта (и их логического группирования) от объявления пространства имен для программиста, использующего класс. Продумайте файловую структуру проекта в связи с использованием пространств имен.

Продумайте ошибки, которые могут возникать в ходе работы со списком, и предусмотрите генерацию исключений в случае возникновения ошибочных ситуаций. Включите новые исключения в иерархию исключений, реализованную в предыдущем задании.

Напишите тестирующую программу для проверки корректности разработанных определений классов. Не забудьте протестировать исключительные ситуации.

Оформление результатов выполнения упражнения

Предоставьте преподавателю на проверку комплект следующих документов:

1. Данное задание.
2. Текст программы.
3. Описание открытого интерфейса класса `RationalList`.

Практикум. Задание № 4

Цель задания

Целью практикума является изучение основных видов отношений между классами и основных принципов проектирования контейнерных классов.

Основные умения, которые должны быть продемонстрированы студентом

- ❒ Умение определить простую иерархию классов с использованием отношений агрегирования и наследования.
- ❒ Умение правильно организовать открытый интерфейс класса.
- ❒ Умение реализовать основные элементы в связи с организацией контейнера (хранилище данных, итератор).
- ❒ Умение разработать несложные обобщенные функции.
- ❒ Умение спроектировать контейнер в соответствии с принципами STL.

Постановка задачи

Основываясь на результатах выполнения предыдущего задания, определите итерируемый контейнер, позволяющий просматривать список без возможности его модификации. Убедитесь в работоспособности контейнера, написав обобщенную функцию копирования элементов итерируемого списка комплексных чисел в обычный массив C++.

Продумайте ошибки, которые могут возникать в ходе работы со списком, и предусмотрите генерацию исключений в случае возникновения ошибочных

ситуаций. Включите новые исключения в иерархию исключений, реализованную в предыдущем задании.

Напишите определение шаблонного класса, воплощающего абстракцию списка для хранения данных произвольных типов по аналогии с STL-контейнером `list`. Проверьте его работоспособность на примере хранения объектов класса `Rational`. Не забудьте протестировать исключительные ситуации.

Оформление результатов выполнения упражнения

Предоставьте преподавателю на проверку комплект следующих документов:

1. Данное задание.
2. Текст программы.
3. Описания открытых интерфейсов спроектированных классов.

Практикум. Задание № 5

Цель задания

Целью практикума является выработка у студентов начальных представлений о работе в составе коллектива разработчиков. Студенты изучают основы организации и сопровождения программного проекта, а также организацию взаимодействия разработчиков отдельных частей проекта между собой и межмодульную интеграцию.

Основные умения, которые должны быть продемонстрированы студентом

- ❒ Умение разобраться в постановке задачи, решаемой другим разработчиком.
- ❒ Умение обеспечить другого разработчика качественной и полной информацией о текущем состоянии проекта, особенностях реализации и о предполагаемых направлениях развития проекта.
- ❒ Умение сформулировать обоснованные критические замечания по решению задачи другим разработчиком и, возможно, найти ошибки реализации, а также сформулировать конкретные предложения по исправлению (развитию, модернизации) проекта.
- ❒ Умение включиться в процесс разработки.

Постановка задачи

- Внимательно изучите постановки задач, сформулированных в материалах по практикуму (задания № 2—4) для назначенного преподавателем студента. В ходе изучения текстов программ и отчетных материалов, а также в результате взаимодействия с автором проекта постарайтесь разобраться в особенностях организации проектов и реализации их отдельных элементов.
- Напишите собственные версии тестирующих программ для классов, спроектированных этим студентом, на основании заданий практикума № 2—4 с целью выявления ошибок и недостатков реализаций.
- Объясните Вашему рецензенту (назначенному преподавателем) особенности своих проектов, реализованных в соответствии с заданиями № 2—4. Предоставьте рецензенту необходимую документацию.

Рецензия на чужой проект

Напишите рецензию на проект назначенного Вам коллеги. В рецензии укажите, какие элементы реализации Вы считаете удачными, а какие — нет. Свое мнение обоснуйте. Сформулируйте предложения по исправлению и (или) модернизации проекта. Обнаружили Вы какие-нибудь серьезные ошибки в проекте? Сформулируйте вопросы к разработчику. Оцените проделанную работу по пятибалльной системе отдельно по каждой составляющей (табл. П1.1), а также в целом.

Таблица П1.1. *Оценка результатов проектирования*

Составляющие работы	Оценка
Соответствие решений постановке задачи	
Организация программ	
Сопровождение программ, документирование	
Способность объяснить проектные решения	
Законченность разработки	
Оценка в целом	

Обсуждение Вашего проекта

Ответьте на замечания и вопросы, содержащиеся в рецензии на Ваш проект, а также на вопросы других участников обсуждения. Выскажите аргументированное согласие или несогласие с отдельными оценками Вашего проекта.

Подведение итогов

Результирующая оценка за выполнение заданий практикума складывается из следующих элементов:

- ❒ грамотная рецензия на чужой проект;
- ❒ наличие положительной рецензии на свой проект;
- ❒ успешное ведение дискуссии.

Практикум. Курсовой проект

Цель задания

Целью курсового проекта является формирование у студентов представлений о проектировании и организации программных проектов, разрабатываемых на основе принципов объектно-ориентированного программирования, а также на основе совместного применения разных парадигм программирования (преимущественно с использованием объектно-ориентированного и процедурного программирования).

Основные умения, которые должны быть продемонстрированы студентом

- ❒ В связи с постановкой задачи:
  - умение грамотно составить техническое задание на разработку программы на основании предложенной постановки задачи;
  - умение выбрать и корректно применить модель программирования или правильно использовать сочетание разных моделей.
- ❒ В связи с построением программного проекта:
  - умение принять грамотные и обоснованные решения по организации и взаимодействию данных и правильно применить управляющие конструкции языка программирования;
  - умение организовать обоснованную модульную структуру проекта.
- ❒ В связи с построением пользовательского интерфейса:
  - способность принять конструктивные решения по организации пользовательского интерфейса, исходя из особенностей постановки задачи;
  - умение организовать ввод данных из файла на диске и вывод результатов в файл на диске с использованием средств потокового ввода-вывода.

- ❒ В связи с испытаниями программы:
  - умение разработать комплекс обоснованных тестовых примеров для проверки корректности функционирования программы;
  - умение смоделировать ошибочные ситуации, а также типичные ситуации, составляющие множество граничных случаев.
- ❒ В связи с качеством проектирования:
  - умение разработать ясно документированную и хорошо читаемую программу;
  - умение предусмотреть обработку ошибочных ситуаций, которые могут возникнуть в ходе исполнения программы;
  - умение продемонстрировать возможность построения квалифицированной программы в рамках используемой модели программирования, имея в виду, что *качество проектирования*, прежде всего, *определяется квалификацией и творческими способностями проектировщиков*, а не моделью программирования. При этом технология проектирования помогает грамотному разработчику продемонстрировать свои умения и построить качественный программный продукт.

Постановка задачи

Разработайте программу, обеспечивающую распознавание в исходном тексте арифметических выражений, оперирующих с рациональными числами. Синтаксис выражений определяют синтаксические диаграммы, представленные на рис. П1.1.

Для идентификатора и целого (десятичного) литерала используйте синтаксические определения из вашего языка программирования.

Результатом работы программы является файл, содержащий значения всех переменных, для которых это значение можно вычислить. В качестве примера приведем следующий исходный текст:

```
alpha = -1\2;
beta = alpha + gamma * 2\3;
gamma = 4\6;
delta = beta - epsilon;
tau = alpha / (2\3 — gamma)
teta = beta;
```

Пример результирующего файла (порядок вывода не регламентируется):

```
alpha = -1\2
beta = -1\18
gamma = 2\3
```

```
delta = undefined: unable to evaluate
epsilon = undefined: no such name
tau = undefined: zero division
teta = -1\18;
```

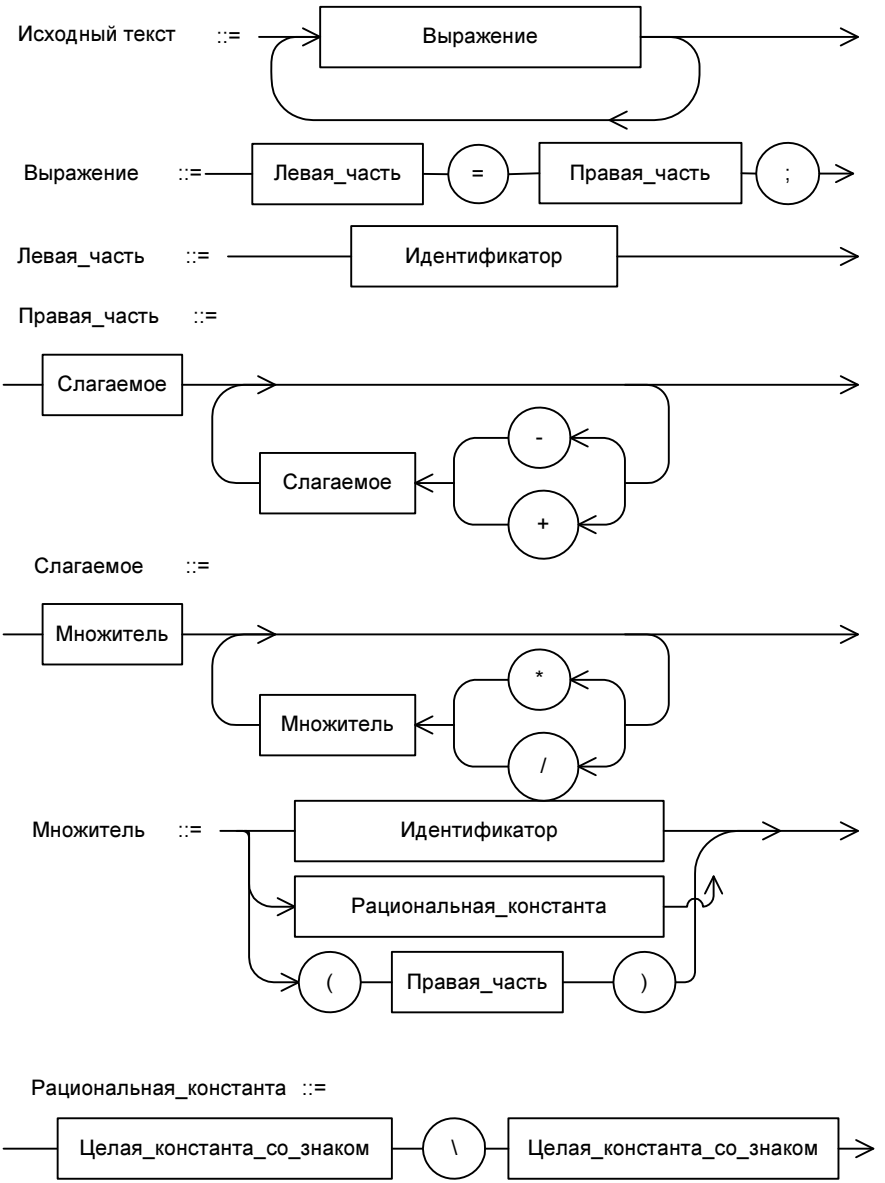


Рис. П1.1. Синтаксис исходного текста, содержащего арифметические выражения, оперирующие с целочисленными комплексными данными

Продумайте ошибки, которые могут возникать в ходе вычисления выражения, и предусмотрите генерацию исключений в случае возникновения ошибочных ситуаций. Для этого создайте простую иерархию исключений "от общего к частному".

Поместите все нелокальные имена, за исключением `main()`, в пространства имен. Обеспечьте отделение объявления пространства как общей среды для разрабатываемых функциональных элементов проекта (и их логического группирования) от объявления пространства имен для программиста, использующего элементы программы. Продумайте файловую структуру проекта в связи с использованием пространств имен.

Отчетность по заданию и подведение итогов

По результатам выполнения задания студент предъявляет преподавателю текст программы и демонстрирует ее работоспособность.

Отчет, предоставляемый преподавателю в электронном виде или в виде распечатки электронного документа, должен состоять из следующих разделов:

- ❒ техническое задание;
- ❒ текст программы;
- ❒ описание программы (требования см. ниже);
- ❒ испытания программы (требования см. ниже).

Описание программы должно включать инструкцию пользователя, спецификации форматов входных и выходных данных, а также необходимые схемы: для объектно-ориентированной части проекта — диаграмма классов, отображающая отношения определенных в программе классов; для процедурной части проекта — структурная схема, отображающая файловую и функциональную организацию проекта и отношения между функциональными блоками.

Раздел "Испытания программы" должен содержать комплекс тестовых примеров, для каждого из которых четко сформулировано его назначение (на основе анализа ожидаемого поведения программы) и ожидаемые результаты исполнения теста. Необходимо предусмотреть тесты для основных обозримых вариантов поведения программы, включая моделирование ошибочных ситуаций, обрабатываемых программой.

Рекомендуемые источники

1. Грис Д. Проектирование компиляторов для цифровых вычислительных машин / Пер. с англ. — М.: Мир, 1975. — 544 с.

2. Новиков Ф. А. Дискретная математика для программистов. Учебник. — СПб.: Питер, 2000. — 304 с., ил.
3. Страуструп Б. Язык программирования C++. Специальное издание / Пер. с англ. — М.: "Издательство БИНОМ"; СПб.: "Невский диалект", 2001. — 1099 с., ил.
4. Лекции и практикум по программированию и информатике (для учащихся СПбГПУ).

ПРИЛОЖЕНИЕ 2

Введение в потоковый ввод-вывод C++

В своей книге "Язык программирования C++" [Stroustrup, 2000] Страуструп отмечает, что проектирование и реализация средств универсального ввода-вывода для языка программирования представляют значительную трудность. Основная трудность реализации универсального ввода-вывода обусловлена, главным образом, следующими обстоятельствами:

- в программах на C++ (как, впрочем, и на других объектно-ориентированных языках) используется большое число типов, определяемых пользователем, объекты этих типов часто нужно уметь вводить и выводить;
- в качестве источника и приемника данных при вводе-выводе не обязательно фигурируют наиболее распространенные виды потоков (например, поток данных, считываемых из файла или с клавиатуры). В качестве потока данных может потребоваться представить другой объект. Примером может служить строковый объект. Вспомните, что даже в стандартной библиотеке C существуют функции типа `sprintf()` и `sscanf()`, позволяющие использовать ANSI-строку как источник или приемник данных при форматированном вводе-выводе;
- представление данных при вводе или выводе может определяться национальными и культурными особенностями (например, в разных странах используются разные форматы представления даты и времени в виде строки). Национальные особенности могут затрагивать представление чисел и символов (например, национальными особенностями определяется, что используется в качестве разделителя целой и дробной частей, какие символы считаются буквами, а какие нет и т. д.)

Не все эти факторы были учтены в первых редакциях стандартной библиотеки C++. Поэтому существуют, вообще говоря, две относительно независимые реализации потокового ввода-вывода в C++.

Первая реализация основывается на типах, объявления которых становятся доступны при использовании заголовочного файла iostream.h (их мы обсудим подробно в *разд. П2.1*). В этой реализации полностью поддерживается ввод-вывод нестандартных типов, а также частично поддержаны нестандартные потоки (в частности, обеспечивается работа со строковым объектом, трактуемым как поток).

Вторая реализация ввода-вывода в стандартной библиотеке C++ основывается на типах, объявления которых становятся доступны при использовании заголовочного файла iostream (без расширения) в стандартном пространстве имен `std`. Эта реализация основывается на шаблонных классах и их специализациях и позволяет распространить концепции ввода-вывода на произвольные типы, а также учесть национальные особенности форматов вводимых и выводимых данных *(см. разд. П2.5)*.

П2.1. Иерархия классов ввода-вывода (заголовочный файл iostream.h)

Обсуждаемая далее реализация ввода-вывода представлена в стандартной библиотеке C++ набором классов, интерфейс к которым обеспечен заголовочным файлом iostream.h. В данном приложении, которое является частью книги, ориентированной, в частности, на учебные цели, мы начинаем рассмотрение средств потокового ввода-вывода именно с изучения средств "старой" библиотеки ввода-вывода, не содержащей шаблонных классов. По мнению автора книги, такой порядок изложения поможет легче разобраться в организации иерархии шаблонных классов и их специализаций в современной версии стандартной библиотеки.

Основная причина, по которой в программах на C++ предпочтительнее использовать именно возможности потокового ввода-вывода C++, а не функции из библиотеки C (интерфейс к которым предоставляется заголовочным файлом stdio.h), заключается не только в объектно-ориентированной природе потокового ввода-вывода C++. Ведь, в конце концов, не все программы на C++ являются объектно-ориентированными.

Основная причина заключается в том, что многие функции из библиотеки C определены как функции с переменным числом параметров, что делает невозможным строгий контроль соответствия типов при обращении к ним. Отметим, что обеспечение более строгого контроля корректности преобразований типов является одной из отличительных особенностей языка C++ по отношению к языку C.

Иерархия классов ввода-вывода (рис. П2.1) является примером иерархии с множественным наследованием и виртуальными базовыми классами. Отме-

тим, что рассмотрение всех классов, связанных с обеспечением ввода-вывода, является довольно обширной и трудоемкой задачей. Не нарушая общей целостности картины, мы рассмотрим некоторое основное подмножество средств библиотеки ввода-вывода, достаточное для выполнения учебных проектов.

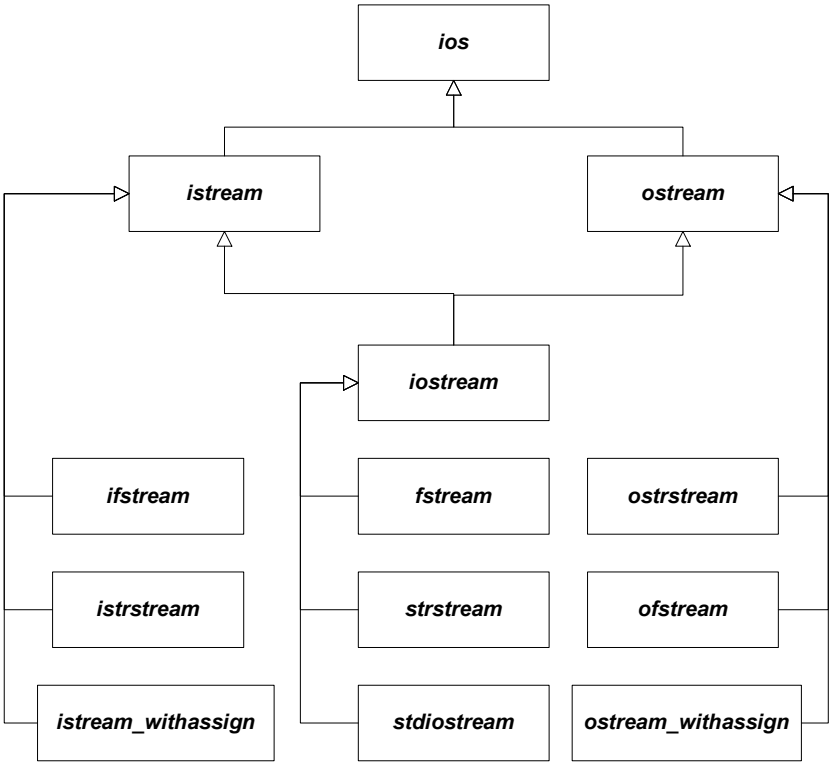


Рис. П2.1. Иерархия классов потокового ввода-вывода

Класс *ios* и производные классы

На вершине иерархии находится класс `ios`, который, несмотря на то, что технически не является абстрактным, как правило, не используется для создания объектов ввода-вывода, а используется только в качестве базового класса для остальных потоков ввода-вывода. Действительно, класс `ios` содержит единственный открытый конструктор потока, для использования которого требуется иметь предварительно созданный объект типа `streambuf`, обеспечивающий возможности буферизованного ввода или вывода. В большинстве случаев намного удобнее воспользоваться производными от `ios` классами, которые

обеспечивают более простой интерфейс для создания потоковых объектов и предоставляют дополнительную функциональность, адаптированную к задачам ввода или вывода данных.

В классе `ios` содержатся определения для общих элементов реализаций различных потоков ввода-вывода (дальнейший текст содержит наиболее существенные фрагменты объявления класса).

- Флаги состояния потока:

```
enum io_state { goodbit = 0x00,    // Флаг нормального состояния
                                   // потока
                eofbit  = 0x01,    // Бит "конец файла"
                failbit = 0x02,    // Бит сбоя
                                   // (поток работоспособен)
                badbit  = 0x04     // Бит серьезной ошибки
                                   // (дальнейшая работа с потоком
                                   //  невозможна)
};
```

- Режимы открытия файловых потоков:

```
enum open_mode{ in  = 0x01,        // Чтение
                out = 0x02,        // Запись
                                   // (ios::trunc подразумевается,
                                   //  если одновременно не указаны:
                                   //      ios::in или
                                   //      ios::ate, или
                                   //      ios::app)

                ate = 0x04,        // После открытия указатель
                                   // позиционируется на конец файла
                app = 0x08,        // Запись в конец файла
                                   // (ios::out подразумевается)
                trunc = 0x10,      // Создание с очисткой прежнего
                                   // содержимого
                nocreate = 0x20,   // Если файл не существует,
                                   // установить состояние ошибки
                noreplace = 0x40,  // Если файл существует,
                                   // установить состояние ошибки
                binary = 0x80      // Двоичный (не текстовый) поток
};
```

- Константы для обозначения базы, относительно которой осуществляется перемещение в пределах потока:

```
enum seek_dir{     beg = 0,    // Относительно начала
                   cur = 1,    // Относительно текущей позиции
                   end = 2     // Относительно конца
};
```

- Флаги форматов ввода-вывода данных:

```
enum{  skipws = 0x0001,    // Пропуск ведущих пробелов при чтении
       left   = 0x0002,    // Выравнивание по левому краю
       right  = 0x0004,    // Выравнивание по правому краю
       dec    = 0x0010,    // Чтение или запись целого числа
                           // в десятичном формате (по умолчанию)
       oct    = 0x0020,    // Чтение или запись целого числа
                           // в восьмеричном формате
       hex    = 0x0040,    // Чтение или запись целого числа
                           // в шестнадцатеричном формате
       showbase = 0x0080,  // При выводе отобразить префикс
                           // формата целого числа
                           // (например, OX для шестнадцатеричных)
       showpoint = 0x0100, // При выводе данных плавающего типа
                           // всегда отображать десятичную точку
       uppercase = 0x0200, // Выводить в верхнем регистре
                           // шестнадцатеричные цифры и
                           // литеру 'E', в экспоненциальном формате
       showpos = 0x0400,   // Для неотрицательных данных
                           // отображать знак "плюс"
       scientific = 0x0800,// Выводить данные плавающего типа
                           // в экспоненциальном формате
       fixed = 0x1000,     // Выводить данные плавающего типа
                           // в формате с фиксированной точкой
       unitbuf = 0x2000    // После каждого вывода
                           // направлять, очищать буфера
                           // (always flush)
};

static const long basefield;   // dec | oct | hex
static const long adjustfield; // left | right | internal
static const long floatfield;  // scientific | fixed
```

- Функции для проверки состояния потока:

```
inline int good() const; // возвращает 1, если нет ошибок
inline int eof() const;  // возвращает 1, если установлен бит
                         // "конец файла"
```

```
inline int fail() const; // возвращает 1, если установлен бит сбоя
inline int bad() const;  // возвращает 1, если установлен бит
                         // серьезной ошибки
```

- Перегрузка операции ! для потоковых объектов

```
inline int operator!() const { return fail() }
```

- Некоторые другие возможности.

Класс для вывода *ostream*

Класс ostream виртуально наследует классу ios и предоставляет реализацию общей функциональности последовательного вывода (sequential output) и вывода с произвольным доступом (random access output). В классе ostream многократно перегружена операция << для обеспечения вывода данных встроенных типов.

Классы для вывода *ofstream*, *ostrstream*, *ostream_withassign*

Класс ofstream обеспечивает представление абстракции выходного потока, связанного с файлом на диске.

Класс ostrstream обеспечивает представление абстракции выходного потока, в котором в качестве приемника данных используется массив символов.

Класс ostream_withassign является вариантом реализации класса ostream, допускающим присваивание потоковых объектов (см. пример использования этого класса в *разд. 7.1*).

Предопределенными выходными потоками типа ostream_withassign являются следующие объекты:

- cout — стандартный поток для вывода (консольный вывод), по умолчанию означает вывод на монитор;
- cerr — стандартный поток для вывода сообщений об ошибках (версия с блочной буферизацией — буфер очищается после каждой операции вывода), по умолчанию означает вывод на монитор;
- clog — стандартный поток для вывода сообщений об ошибках (полностью буферизованная версия).

Эти объекты могут быть присвоены другим объектным переменным типа ostream.

Класс для ввода *istream*

Класс istream виртуально наследует классу ios и предоставляет реализацию общей функциональности последовательного ввода (sequential input) и ввода

с произвольным доступом (random access input). В классе `istream` многократно перегружена операция `>>` для обеспечения ввода данных встроенных типов.

Классы для ввода *ifstream*, *istrstream*, *istream_withassign*

Класс `ifstream` обеспечивает представление абстракции входного потока, связанного с файлом на диске.

Класс `istrstream` обеспечивает представление абстракции входного потока, в котором в качестве источника данных используется массив символов.

Класс `istream_withassign` является вариантом реализации класса `istream`, допускающим присваивание потоковых объектов (см. пример использования этого класса в *разд. 7.1*).

Предопределенным входным потоком типа `istream_withassign` является объект `cin` — стандартный поток для ввода (консольный ввод), по умолчанию означает ввод с клавиатуры.

Класс для ввода и вывода *iostream*

Класс `iostream` наследует классам `istream` и `ostream` и предоставляет реализацию общей функциональности потока, используемого для последовательного ввода-вывода и ввода-вывода с произвольным доступом. Как правило, объекты класса `iostream` не создаются, а используются объекты производных от него классов.

Классы для вывода *fstream*, *strstream*, *stdiostream*

Класс `fstream` обеспечивает представление абстракции потока ввода-вывода, связанного с файлом на диске.

Класс `strstream` обеспечивает представление абстракции потока ввода-вывода, в котором в качестве приемника данных используется массив символов.

Класс `stdiostream` обеспечивает представление абстракции потока ввода-вывода, основанного на использовании стандартных системных средств ввода-вывода. Данный класс включен в библиотеку для обеспечения совместимости со старыми версиями потокового ввода-вывода. Функции-члены класса `stdiostream` могут использоваться совместно с функциями, объявленными в заголовочном файле stdio.h.

П2.2. Вывод данных

В данном разделе рассматривается вывод данных средствами объектов типа `ostream` и производных от него типов.

Вывод данных встроенных типов

Поскольку для встроенных типов обеспечена перегруженная операция <<, то вывод данных встроенных типов, не использующий средств форматирования, выглядит довольно просто:

```
#include <iostream.h>

void PrintLine( int x ) {
    cout << "x=" << x << '\n';
}
```

В приведенном примере использованы три версии перегруженной операции (для строк, для значений целого типа и для символьных значений), при этом сначала на консоль будет выведена строка "x=", а следом за ней — значение переменной x. Возможность последовательной записи цепочек выводимых данных обеспечена тем, что перегруженная операция << в качестве результата формирует ссылку на объект типа ostream.

Вывод символов и строк

Помимо возможностей, обеспеченных операцией <<, для вывода символов и строк могут оказаться полезными открытые функции-члены класса ostream со следующим интерфейсом:

```
ostream& put( char ch ); // ch — печатаемый символ

ostream& write( const char* s, int nCount );
ostream& write( const unsigned char* s, int nCount );
ostream& write( const signed char* s, int nCount );
                        // s — указатель на начало строки
                        // nCount — число печатаемых символов
```

Отметим, что перегруженные функции write() полезны при выводе в двоичные потоки.

Буферизация

Как правило, при работе с потоками ввода и вывода используется буферизация.

На примере выходных потоков рассмотрим организацию буферизации ввода-вывода более подробно.

Буферизация при выводе означает, что выводимые символы сначала попадают в буфер (область памяти, создаваемую и управляемую объектом, предна-

значенным специально для этого), а через какое-то время переписываются туда, куда нужно. Это позволяет не терять производительность приложения, поскольку, с одной стороны, быстродействие внешних устройств существенно ниже быстродействия процессора, а с другой стороны, возможны ситуации, когда по каким-то системным причинам приемник не может немедленно принять данные.

Объект, обеспечивающий буферизацию, имеет тип, производный от класса `streambuf`, который является абстрактным классом. При этом для различных типов выходных потоков используются различные версии буфера (рис. П2.2).

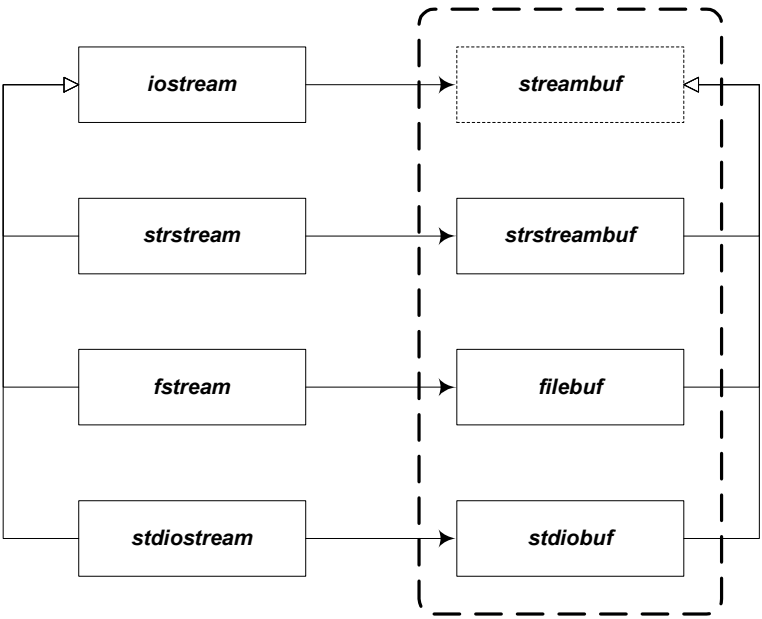


Рис. П2.2. Потоки вывода и буферизация

В некоторых случаях в ходе работы приложения нужно быть уверенным, что все данные из буфера записаны в выходной поток. Для того чтобы явно потребовать переписать содержимое буфера в выходной поток, следует воспользоваться открытой функцией-членом класса `ostream` со следующим интерфейсом:

```
ostream& flush();
```

То же самое можно реализовать с использованием специального манипулятора вывода `flush` *(см. информацию в следующем разделе)*.

Форматирование вывода. Манипуляторы

По умолчанию при использовании операции << печатается необходимое минимальное число символов. В распоряжении программиста имеются средства форматирования вывода, позволяющие представить выходные данные в требуемом виде. Этими средствами являются так называемые манипуляторы вывода, используемые в качестве аргумента операции <<. Примером наиболее распространенного манипулятора является манипулятор перехода на новую строчку при выводе. С его использованием рассмотренный выше пример можно переписать следующим образом:

```
#include <iostream.h>

void PrintLine( int x ) {
    cout << "x=" << x << endl; // Использован манипулятор endl
}
```

Существуют два основных вида манипуляторов. Манипуляторы первого вида не требуют никаких дополнительных сведений (поэтому их называют манипуляторами без аргументов). Для использования манипуляторов без аргументов не нужно подключать никаких дополнительных заголовочных файлов, поскольку все они либо являются членами класса `ostream`, либо унаследованы вместе с `ios`. Основные используемые при выводе манипуляторы без аргументов перечислены в табл. П2.1.

Таблица П2.1. *Манипуляторы без аргументов при выводе*

Манипулятор	Класс, в котором объявлен	Описание	Продолжительность действия
`dec`	`ios`	Вывод последующих целочисленных данных в десятичном формате (действует по умолчанию)	Пока не будет отменено другим манипулятором преобразования формата
`oct`	`ios`	Вывод последующих целочисленных данных в восьмеричном формате	
`hex`	`ios`	Вывод последующих целочисленных данных в шестнадцатеричном формате	
`endl`	`ostream`	Вывод символа новой строки и очистка буфера	Однократно
`flush`	`ostream`	Очистка буфера	

Использование перечисленных манипуляторов иллюстрирует следующий пример:

```
#include <iostream.h>

void PrintFormats( int x ) {
                          // Вывод:
    cout << x << endl;    // ...в текущем формате
                          //    (если до этого формат не изменялся,
                          //     то по умолчанию вывод в десятичном
                          //     формате)

    cout << dec << x << endl;  // ...в десятичном формате (явно)
    cout << oct << x << endl;  // ...в восьмеричном
    cout << hex << x << endl;  // ...в шестнадцатеричном

       cout << dec;            // Восстановление умолчания!
}
```

В ходе работы функции `PrintFormats()` на консоль в отдельных строчках будет выведено представление значения переменной `x` в десятичном, восьмеричном и шестнадцатеричном форматах. После этого восстанавливается формат по умолчанию.

Второй вид манипуляторов — это манипуляторы, требующие аргумент (их иногда называют параметризованными). Для использования таких манипуляторов необходимо подключить заголовочный файл iomanip.h. В табл. П2.2 перечислены параметризованные манипуляторы, причем для каждого указан тип параметра (`int` или `long`).

Таблица П2.2. *Параметризованные манипуляторы при выводе*

Манипулятор	**Описание**	**Продолжительность действия**
`setw(int)`	Устанавливает ширину поля вывода (по умолчанию — необходимое минимальное число символов)	Действует только на очередной вывод
`setfill(int)`	Устанавливает заполняющий символ, используемый при дополнении до ширины поля вывода (по умолчанию — пробел)	Пока не будет отменено манипулятором, устанавливающим другой заполнитель
`setprecision(int)`	Устанавливает точность вывода значений с плавающей точкой (по умолчанию — 6 разрядов после десятичной точки)	Пока не будет установлена другая точность

Таблица П2.2 *(окончание)*

Манипулятор	Описание	Продолжительность действия
`setiosflags(long)`	Устанавливает флаги форматов ввода-вывода (см. описание класса `ios`)	Каждый флаг действует до тех пор, пока не будет установлен флаг, его отменяющий
`resetiosflags(long)`	Переустанавливает флаги форматов ввода-вывода (очищает указанные флаги)	

Использование манипуляторов `setw`, `setfill` и `setprecision` достаточно очевидно и обычно не вызывает затруднений. Остановимся более подробно на остальных манипуляторах. Из прототипа манипуляторов `setiosflags` и `resetiosflags` следует, что они требуют значение типа `long` в качестве аргумента. Это значение может быть сформировано как поразрядное ИЛИ, составленное из требуемых флагов форматов ввода-вывода, перечисленных в описании класса `ios`, например:

```
cout << hex;
cout << setiosflags( ios::left | ios::showbase | ios::uppercase );
            // Выводить в шестнадцатеричном формате,
            // выравнивать по левому краю,
            // представлять выводимые числовые данные в
            // соответствии с правилами C++
            // (выводить 0X для шестнадцатеричных чисел),
            //  для литер A,B,...,F использовать верхний регистр

            // ...

cout << resetiosflags( ios::left | ios::showbase | ios::uppercase );
            // Очистить ранее установленные флаги форматов
```

Умалчиваемое значение флагов форматов ввода-вывода:

```
ios::left | ios::dec
```

Весьма полезными при управлении форматами вывода могут также оказаться функции, объявленные в классе `ios`:

```
int width( int nw ); // Устанавливает значение ширины поля вывода
                     // в соответствии с переданным аргументом
                     // (действует только на ближайший вывод)
                     // Возвращает 0
```

```
int width() const;   // Возвращает текущее значение ширины

int precision( int np ); // Устанавливает значение точности для
                         // вывода данных с плавающей точкой
                         // Возвращает прежнее значение точности

int precision() const;   // Возвращает текущее значение точности

char fill( char cFill );   // Устанавливает заполняющий символ
                           // Возвращает прежнее значение
                           // заполняющего символа

char fill() const;         // Возвращает текущее значение
                           // заполняющего символа
```

Рассмотрите приведенный далее пример функции, использующей как манипуляторы с аргументом, так и манипуляторы без аргумента.

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

void PrintFormatsManipulated( int x, double y ) {

    // Установить форматы вывода:
    //
    //    Выводить в шестнадцатеричном формате,
    //    выравнивать по левому краю,
    //    представлять выводимые числовые данные в
    //    соответствии с правилами C++
    //    (выводить 0X для шестнадцатеричных чисел),
    //     для литер A,B,...,F использовать верхний регистр

    cout << setiosflags( ios::left | ios::showbase | ios::uppercase );

    // Сохранить прежний заполняющий символ и установить
    // новый заполняющий символ - точка
    char oldFill = cout.fill( '.' );

    // Вывести значение x в десятичном формате в поле шириной
    //   10 символов
    //   NB! Действует выравнивание по левому краю
    cout << dec << setw( 10 ) << x << endl;

    // Вывести значение x в восьмеричном формате в поле шириной
    cout << oct << x << endl;

    // Вывести значение x в шестнадцатеричном формате в поле
    //   шириной 12 символов
```

```
    //   NB! Действует выравнивание по левому краю
    //   Выводится префикс 0X перед шестнадцатеричным значением
    //   Используется верхний регистр для записи цифр A..F
    cout << hex << setw( 12 ) << x << endl;

    // Сохранить прежнее значение точности и установить
    //   новое значение — 2 десятичных знака после точки
    int oldPrecision = cout.precision( 2 );

    // Вывести значение y в 20 позициях с точностью 2
    cout << setw( 20 ) << y;

    // Восстановить форматы, действовавшие до начала выполнения
    // функции PrintFormatsManipulated()
    cout << dec;
    cout << setprecision( oldPrecision );
    cout << setfill( oldFill );
    cout << resetiosflags( ios::left|ios::showbase|ios::uppercase );
}
```

Обратите внимание на то, что при установке флагов форматов ввода-вывода, а также при изменении значений точности представления и ширины поля вывода внутри функциональных блоков, как правило, следует не забывать восстанавливать состояние флагов форматов ввода-вывода во избежание влияния установленных флагов на вывод, реализуемый за пределами соответствующего функционального блока.

Вывод для типов, не встроенных в C++

Вывод для произвольных типов несложно обеспечить, перегрузив операцию <<.

Если программист имеет доступ к представлению типа, для которого реализуется перегруженная операция вывода, то в объявление класса может быть помещено объявление дружественной функции, соответствующей перегружаемой операции.

Внутри перегружаемой операции в этом случае будет доступ к закрытым членам класса, например:

```
#include <iostream.h>

class MyComplex {
    double re;
    double im;
```

```
public:
    MyComplex( double real = 0, double image = 0 ) {
        re = real;
        im = image;
    }
    // ...
    friend ostream& operator<< ( ostream&, const MyComplex& );
};

ostream& operator<< ( ostream& stream, const MyComplex& c ) {
    // Есть доступ к закрытым членам re и im
    stream << c.re << "+" << c.im << "i";
}
```

Данная версия операции вывода для данных типа `MyComplex` не идеальна: при оформлении выводимого комплексного числа не учитывается знак мнимой части. Необходимые изменения не представляют сложности и могут быть реализованы студентами самостоятельно.

Однако в некоторых случаях программист не имеет доступа к представлению класса. Тогда вместо дружественной функции реализуется обычная функция. При этом, разумеется, программист имеет возможность использовать только открытый интерфейс класса:

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

class MyComplex {
    double re;
    double im;
public:
    MyComplex( double real = 0, double image = 0 ) {
        re = real;
        im = image;
    }
    inline double getRe() { return re; }
    inline double getIm() { return im; }
    // ...
};

ostream& operator<< ( ostream& stream, const MyComplex& c ) {
    stream << c.getRe() << "+" << c.getIm() << "i";

    return stream;
}
```

Отметим, что в этом случае для представления действительной и мнимой части комплексных данных весьма пригодилась бы концепция свойств *(см. разд. 9.5)*, однако C++ не поддерживает свойств.

Напоминаем, что если в подобных перегружаемых операциях используется изменение форматов вывода, то следует не забывать восстановить исходное состояние флагов форматов, например:

```
ostream& operator<< ( ostream& stream, const MyComplex& c ) {

    cout << setiosflags( ios::showpoint );

    int oldPrecision = cout.precision( 2 );

    stream << getRe() << "+" << c.getIm() << "i";

    cout << setprecision( oldPrecision );
    cout << resetiosflags( ios::showpoint );

    return stream;
}
```

В любом случае, не следует определять перегружаемые операции >> и << в виде членов класса (несмотря на формальную легальность таких определений), например:

```
class MyComplex {
    // ...
public:
    ostream& operator<<( ostream& ); // Плохой стиль!
    istream& operator>>( istream& ); // Плохой стиль!
};
```

В этом случае при вводе-выводе объектов класса `MyComplex` придется записывать операции ввода-вывода в непривычной форме (выводимый объект — слева от операции, а потоковый объект — справа):

```
MyComplex c;
c >> cout;
c << cin;
```

Это работает, но противоречит стандартным соглашениям о записи операций ввода-вывода, и, безусловно, путает читателя программы [Meyers, 1998, Item 19].

П2.3. Ввод данных

В этом разделе рассматривается ввод данных средствами объектов типа `istream` и производных от него типов.

Ввод данных встроенных типов

Поскольку для встроенных типов обеспечена перегруженная операция >>, то ввод данных встроенных типов, не использующий средств форматирования, выглядит так же просто, как и вывод:

```
#include <iostream.h>

int ReadLine() {
    int x;

    cin >> x;
    return x;
}
```

При вводе также можно записывать цепочки вводимых данных, т. к. перегруженная операция >> в качестве результата формирует ссылку на объект типа `istream`.

Ввод символов и строк

Помимо возможностей, обеспеченных операцией >>, для ввода символов и строк могут оказаться полезными открытые функции-члены класса `istream` со следующим интерфейсом:

```
istream& get( char& ch );  // Считывает символ из потока
                           // ch — считанный символ

istream& get( char* s, int nCount, char delim = '\n' );
                           // Считывает символы в массив
                           // s — указатель на начало массива
                           // nCount — максимальное число символов
                           // delim — разделитель
                           // Добавляет нуль-символ
                           // Разделитель остается во входном потоке

istream& getline( char* s, int nCount, char delim = '\n' );
                           // То же, что и get, но разделитель
                           // считывается из входного потока
```

```
istream& read( char* s, int nCount );
                            // Считывает указанное число символов
                            // nCount или до конца файла
                            // и сохраняет их в строке s
                            // Удобна для чтения из двоичных потоков

istream& ignore( int nCount, int delim = EOF );
                            // Считывает указанное число символов
                            // nCount до разделителя delim,
                            // но нигде не сохраняет считанные символы

istream& putback ( char ch ); // Вталкивает символ обратно в поток:
                              // ch должен действительно быть последним
                              // прочитанным символом, в противном
                              // случае результат непредсказуем
```

Форматированный ввод. Манипуляторы

По умолчанию при использовании операции >> считывание производится до ближайшего разделителя. Аналогично выводу данных, форматированием ввода также можно управлять с использованием манипуляторов, например:

```
#include <iostream.h>

void ReadValue( istream& input, int& x ) {

    // Ввести из поля размером 10 позиций значение x,
    //   представленное во входном потоке в шестнадцатеричном
    //   формате
    input >> setw( 10 ) << hex << x;
}
```

Манипуляторы, используемые при вводе, приведены в табл. П2.3, П2.4.

Таблица П2.3. *Манипуляторы без аргументов при вводе*

Манипулятор	Класс, в котором объявлен	Описание	Продолжительность действия
dec	ios	Ввод целочисленных данных в десятичном формате (действует по умолчанию)	Пока не будет отменено другим манипулятором преобразования формата
oct	ios	Ввод целочисленных данных в восьмеричном формате — допускается ведущий префикс 0	

Таблица П2.3 *(окончание)*

Манипулятор	Класс, в котором объявлен	Описание	Продолжительность действия
`hex`	`ios`	Ввод целочисленных данных в шестнадцатеричном формате — допускается ведущий префикс `0X` или `0x`	
`ws`	`istream`	Пропуск пробелов, символов табуляции и переводов строк до ближайшего непробельного символа	Однократно

Таблица П2.4. *Параметризованные манипуляторы при вводе*

Манипулятор	Описание	Продолжительность действия
`setw( int)`	Устанавливает ширину поля ввода (по умолчанию — до ближайшего разделителя)	Действует только на очередной ввод
`setiosflags( long)`	Устанавливает флаги форматов ввода-вывода (см. описание класса `ios`)	Каждый флаг действует до тех пор, пока не будет установлен флаг, его отменяющий
`resetiosflags( long)`	Переустанавливает флаги форматов ввода-вывода (очищает указанные флаги)	

Ввод для типов, не встроенных в C++

Аналогично перегрузке операции вывода данных в поток, ввод для произвольных типов может быть обеспечен перегрузкой операции >>. Вариант возможной реализации ввода-вывода для типа `MyComplex` мог бы иметь, например, следующий вид:

```
#include <iostream.h>

class ReadComplexException {}; // Исключение, генерируемое при ошибке
                               // чтения комплексного данного

class MyComplex {
    double re;
    double im;
```

```
public:
    MyComplex( double real = 0, double image = 0 ) {
        re = real;
        im = image;
    }

    // ...
    friend ostream& operator<< ( ostream&, const MyComplex& );
    friend istream& operator>> ( istream&, MyComplex& );
};

// Перегружаемая операция для вывода комплексного данного
// (пример реализации)
ostream& operator<< ( ostream& output, const MyComplex& c ) {
    output << c.re << "+" << c.im << "i";

    return output;
}

// Перегружаемая операция для ввода комплексного данного
// (пример реализации)
istream& operator>> ( istream& input, MyComplex& c ) {
    char ch;
    double temp;

    // Ввод числа с плавающей точкой во временную переменную
    input >> temp;
    if( !input ) throw ReadComplexException();

    input >> ws; // Пропуск пробельных символов
    input.get( ch );
    if( !input ) {
        // Если больше символов нет, значит, число состоит
        // только из действительной части
        c.re = temp;
        c.im = 0;
        input.clear();
        return input;
    }

    if( ch == 'i' || ch == 'I' ) {
    // Если вслед за считанным значением сразу следует символ
    // 'i', значит, число состоит только из мнимой части
        c.re = 0;
        c.im = temp;
        return input;
    }
```

```
    // Проверка корректности знака операции
    if( ch!='+' && ch!='-' ) throw ReadComplexException();

    // Здесь: знак операции корректен, следовательно
    // копируем значение из временной переменной в действительную
    // часть комплексной величины
    c.re = temp;

    // Осталось прочитать мнимую часть
    input >> c.im;
    if( !input ) throw ReadComplexException();

    // Если необходимо, инвертируем знак
    if( ch=='-' ) c.im = -c.im;

    // Проверяем, есть ли символ 'i' вслед за мнимой частью
    input >> ws;
    input.get( ch );
    if( !input ) throw ReadComplexException();

    if( ch != 'i' && ch!='I' ) throw ReadComplexException();

    return input;
}
```

В данном примере предполагается, что во входном потоке комплексные данные должны задаваться в виде `a+bi` или `a-bi`, при этом допустимо использование произвольного числа пробелов между элементами записи, а также допускается отдельный знак мнимой части, т. е. запись `5+ -4i` считается корректной. В случае, когда исходный текст не удовлетворяет заданному формату представления, генерируется исключение.

П2.4. Файловые потоки

Как следует из приведенной ранее схемы иерархии классов библиотеки ввода-вывода, файловый ввод-вывод обеспечен классами `ifstream`, `ofstream` и `fstream`, которые соответственно являются производными от классов `istream`, `ostream` и `iostream`. Поэтому оказывается, что существенная часть возможностей в связи с вводом из файлового потока и выводом в файловый поток уже нами рассмотрена.

Основное отличие файловых потоков заключается в том, что такой поток нужно ассоциировать с файлом на диске, определив при этом режим использования файла, а также протокол доступа к файлу. Для файловых потоков могут использоваться некоторые дополнительные манипуляторы ввода-вывода. Указанные особенности рассматриваются в данном разделе.

Отметим, что для использования в программе файловых потоков необходимо подключить заголовочный файл fstream.h.

Ввод данных из файлового потока *ifstream*

Основными способами инициализации файлового потока для ввода являются следующие.

- Создание объекта, поддерживающего буферизованный ввод и связывание его с файлом на диске посредством конструктора класса `ifstream`:

```
#include <fstream.h>

void UsingInputFileStream1() {

    ifstream fInput( "input.txt" );
    if( !fInput ) {
        // Обработка ошибки открытия файла
        // ...
    }

    // ...
}
```

- Создание объекта типа `ifstream` с последующим вызовом функции-члена класса `open()`:

```
#include <fstream.h>

void UsingInputFileStream2() {

    ifstream fInput;
    fInput.open( "input.txt" );
    if( !fInput ) {
        // Обработка ошибки открытия файла
        // ...
    }

    // ...
}
```

Рассмотрим более подробно спецификации конструктора класса `ifstream` и функции `open()`.

Используемый в рассмотренном выше примере конструктор имеет следующий интерфейс:

```
ifstream( const char *szName, // ASCIIZ-строка: имя файла ввода
          int mode = ios::in, // Режим открытия файла:
                              //  ios::in — для чтения
                              //     (действует по умолчанию)
                              //  ios::nocreate — устанавливается
                              //     бит сбоя, если файл
                              //     не существует
                              //  ios::binary — файл открывается
                              //     в двоичном режиме
          int nProt = filebuf::openprot // Протокол доступа:
                              //  filebuf::openprot — может
                              //     открываться разными
                              //     приложениями одновременно
                              //     (действует по умолчанию)
                              //  filebuf::sh_none — может
                              //     использоваться только данным
                              //     приложением
                              //  filebuf::sh_read — разрешается
                              //     совместный доступ по чтению
                              //  filebuf::sh_write — разрешается
                              //     совместный доступ по записи
);
```

Некоторые распространенные варианты инициализации файлового объекта для ввода иллюстрирует следующий фрагмент:

```
// Открыть файл для чтения как двоичный поток
ifstream fInput1( "input1.txt", ios::in | ios::binary );
// ...

// Проверить существование файла
ifstream fInput2( "input2.txt", ios::nocreate );
if( !fInput2 ) {
    // Файл не существует
}
// ...

// Открыть для чтения как текстовый поток с возможностью
// использования в качестве разделяемого ресурса только для записи
ifstream fInput3( "input3.txt", ios::in, filebuf::sh_read );
// ...
```

Прототип функции `open()` является, по существу, калькой с только что рассмотренного конструктора, поэтому дополнительного изучения не требует.

Закрытие потока осуществляется автоматически при вызове деструктора (например, при выходе объекта типа `ifstream` за пределы области видимости). Если требуется обеспечить закрытие потока раньше, можно воспользоваться функцией-членом класса `close()`, которая имеет следующий интерфейс:

```
void close();
```

Эта функция вызывает функцию `close()`-член класса `filebuf`, которая, в свою очередь, закрывает файл и разрывает логическую связь между потоковым объектом и файлом на диске. Объект типа `filebuf` при этом не разрушается.

Вывод данных в файловый поток *ofstream*

Основными способами инициализации файлового потока для вывода являются следующие.

- Создание объекта, поддерживающего буферизованный вывод и связывание его с файлом на диске посредством конструктора класса `ofstream`:

```
#include <fstream.h>

void UsingOutputFileStream1() {

    ofstream fOutput( "output.txt" );
    if( !fOutput ) {
        // Обработка ошибки открытия файла
        // ...
    }

    // ...
}
```

- Создание объекта типа `ofstream` с последующим вызовом функции-члена класса `open()`:

```
#include <fstream.h>

void UsingOutputFileStream2() {

    ofstream fOutput;
    fOutput.open( "output.txt" );
    if( !fOutput ) {
        // Обработка ошибки открытия файла
        // ...
    }

    // ...
}
```

Рассмотрим более подробно спецификацию конструктора класса `ofstream`.

Используемый в рассмотренном выше примере конструктор имеет следующий интерфейс:

```
ofstream( const char *szName, // ASCIIZ-строка: имя файла ввода
          int mode = ios::out,// Режим открытия файла:
                              //  ios::out — для записи
                              //     (действует по умолчанию)
                              //  ios::in — прежнее содержимое не
                              //     будет очищаться
                              // Другие режимы: см. спецификацию
                              //     класса ios
          int nProt = filebuf::openprot // Протокол доступа:
                              //  filebuf::openprot — может
                              //  открываться разными приложениями
                              //  одновременно
                              //     (действует по умолчанию)
                              //  filebuf::sh_none — может
                              //  использоваться только данным
                              //  приложением
                              //  filebuf::sh_read — разрешается
                              //  совместный доступ по чтению
                              //  filebuf::sh_write — разрешается
                              //  совместный доступ по записи
);
```

Некоторые распространенные варианты инициализации файлового объекта для вывода иллюстрируют следующие фрагменты:

```
// Открыть для записи без очистки прежнего содержимого
ofstream fOutput1( "output1.txt", ios::out | ios::in );
// ...

// Проверить существование файла
ofstream fOutput2( "output2.txt", ios::nocreate );
if( !fOutput2 ) {
    // Файл не существует
}
// ...

// Открыть для записи в конец файла с возможностью
// использования в качестве разделяемого ресурса только для записи
ofstream fOutput3( "output3.txt", ios::app, filebuf::sh_write );
// ...
```

Закрытие потока осуществляется автоматически при вызове деструктора (например, при выходе объекта типа `ofstream` за пределы области видимости). Если требуется обеспечить закрытие потока раньше, можно воспользоваться функцией-членом класса `close()`, аналогично случаю использования файлового потока для ввода.

Использование потока *fstream* для реализации ввода и вывода при работе с одним и тем же файлом

Иногда необходимо обеспечить такое взаимодействие с файлом, которое предусматривало бы возможность его чтения и записи в процессе работы с одним и тем же потоком ввода вывода. Например, когда пользователь текстового редактора редактирует содержимое некоторого файла, он одновременно использует данный файл и как источник данных, и как приемник данных.

Для реализации подобных возможностей используется объект класса `fstream`. Инициализация такого объекта сходна с инициализацией объектов типа `ifstream` и `ofstream`. Некоторые отличия существуют в интерпретации значения аргумента конструктора или функции `open()`, отвечающего за режим открытия файла. Приведем фрагмент спецификации конструктора класса `fstream`:

```
fstream( const char *szName, // ASCIIZ-строка: имя файла ввода
         int mode,           // Режим открытия файла:
                             //   NB! Значения по умолчанию нет!
                             //
                             // Все режимы определяются в
                             //  спецификации класса ios
         int nProt = filebuf::openprot // Протокол доступа
);
```

Обратите внимание, что для объектов типа `fstream` интерпретация режимов открытия потоков полностью соответствует интерпретации, приведенной в описании класса `ios`. Для потока `ifstream` доступно только подмножество возможных режимов, а для потока `ofstream` предусмотрена специальная интерпретация значения `ios::in`.

В ходе работы с потоком, допускающем и чтение, и запись, в связи с позиционированием внутри потока часто используются следующие функции:

```
/*
 * При работе с потоком ввода-вывода в связи с ВВОДОМ данных
 */
```

```
long istream::tellg(); // TELL Get pointer:
                       // Функция возвращает номер байта,
                       // соответствующий текущему значению
                       // указателя файла при чтении
                       // Нумерация байтов начинается с 0

istream& istream::seekg( long pos ); // SEEK Get pointer:
                       // Функция перемещает указатель файла на
                       // требуемую позицию pos

istream& istream::seekg( long offset, ios::seek_dir direction );
                       // Функция смещает указатель на offset
                       // позиций относительно direction:
                       //   ios::beg — относительно начала потока
                       //   ios::cur — относительно текущей
                       //              позиции
                       //   ios::end — относительно конца потока

/*
 * При работе с потоком ввода-вывода в связи с ВЫВОДОМ данных
 */

long ostream::tellp(); // TELL Put pointer:
                       // Функция возвращает номер байта,
                       // соответствующий текущему значению
                       // указателя файла при записи

ostream& ostream::seekp( long pos ); // SEEK Put pointer:
                       // Функция перемещает указатель файла на
                       // требуемую позицию pos

ostream& ostream::seekp( long offset, ios::seek_dir direction );
                       // Функция смещает указатель на offset
                       // позиций относительно direction:
                       //   ios::beg — относительно начала потока
                       //   ios::cur — относительно текущей
                       //              позиции
                       //   ios::end — относительно конца потока
```

Из приведенных объявлений видно, что функции `seekg()` и `tellg()` определены в классе `istream`, а функции `seekp()` и `tellp()` — в классе `ostream`.

Так как класс `fstream` унаследован от класса `iostream`, а тот, в свою очередь, является продуктом множественного наследования от классов `istream` и `ostream`, то методы классов `istream` и `ostream` доступны и в классе `fstream`.

Заметим, что реализация названных функций в `istream` и `ostream` обеспечивает возможность конструирования классов, допускающих позиционирование

указателя файла. Эти функции не имеют эффекта, если поток не прикреплен к чему-нибудь такому, для чего позиционирование имеет смысл.

Манипуляторы в связи с файловыми потоками

При работе с файловыми потоками могут быть использованы все рассмотренные ранее манипуляторы ввода-вывода. Дополнительно для файловых потоков могут применяться следующие манипуляторы (табл. П2.5).

Таблица П2.5. *Манипуляторы в связи с файловыми потоками*

Манипулятор	Класс, в котором объявлен	Описание	Продолжительность действия
`text`	`ios`	Устанавливает текстовый режим потока (действует по умолчанию)	Пока не будет отменено другим манипулятором
`binary`	`ios`	Устанавливает двоичный режим потока	

Относительно этих манипуляторов в документации по классу `ios` говорится, что их можно использовать с любым потоком, для которого установлена связь с буфером `filebuf`.

П2.5. Потоковый ввод-вывод в пространстве имен стандартной библиотеки C++

Как уже было сказано выше, альтернативная реализация ввода-вывода представлена в стандартной библиотеке C++ иерархией шаблонных классов и их специализаций, интерфейс к которым обеспечен заголовочным файлом iostream (без расширения). Указанные классы объявлены в пространстве имен `std`. Подробное описание этих средств можно почерпнуть из книги Страуструпа [Stroustrup, 2000, гл. 21]. Здесь мы рассмотрим основные принципы использования библиотеки ввода-вывода в пространстве имен `std`.

Примечание

Заметим, что программы, использующие заголовочный файл iostream.h, во многих случаях без изменений перекомпилируются с заголовочным файлом iostream, поскольку соответствующие специализации объявленных в iostream шаблонных классов поддерживают в качестве подмножества основную часть интерфейса библиотеки ввода-вывода, описанной в *разд. П2.1–П2.4* (для учеб-

ных целей этих средств может оказаться вполне достаточно). Впрочем, следует иметь в виду, что некоторые программы, использующие "старую" версию библиотеки потокового ввода-вывода C++, могут оказаться несовместимыми с интерфейсом, определяемым заголовочным файлом iostream (без расширения), см., например, фрагменты программ, представленных в листингах 7.4—7.6.

Таким образом, в распоряжении программиста, поддерживающего код, использующий "старую" версию библиотеки, имеются две альтернативы. Первая состоит в том, чтобы исключить из кода заголовочные файлы, предоставляющие интерфейс к средствам "старой" библиотеки (fstream.h, iomanip.h, ios.h, iostream.h, istream.h, ostream.h, streamb.h и strstrea.h). Как уже было отмечено выше, часть кода при этом, возможно, придется переписать. Если же поддерживаемая программа очень сильно зависит от использования ресурсов старой версии, то, возможно, наилучшим решением для обеспечения работоспособности кода будет вторая альтернатива, заключающаяся в отказе от использования обновленных средств ввода-вывода. При этом следует иметь в виду, что невозможно совместное применение ресурсов двух версий библиотек ввода-вывода в пределах одной программы.

Иерархия классов стандартной библиотеки ввода-вывода

В этом разделе мы рассмотрим организацию взаимодействия классов, обеспечивающих средства ввода-вывода, определенные в пространстве имен `std`. При этом мы подробно остановимся на определениях классов, обеспечивающих важнейшие общие свойства потоков ввода и вывода, оставляя читателю изучение деталей реализации в качестве самостоятельной работы.

Класс *ios_base*

На вершине иерархии классов стандартной библиотеки ввода-вывода (рис. П2.3) находится класс `ios_base`, обеспечивающий представление общих как для потоков ввода, так и для потоков вывода свойств потоковых объектов и функций-членов класса, которые не зависят от типа элементов данных, с которыми работает поток. Поэтому класс `ios_base` не является шаблонным классом.

Класс `ios_base` не предназначен для создания самостоятельных объектов. Объекты класса `ios_base` обычно являются подобъектами производных от `ios_base` типов. Эти подобъекты хранят информацию, которая является общей для обслуживания как процессов ввода данных из потока, так и процессов вывода данных в поток. В частности, объект `ios_base` хранит следующую информацию о форматировании объектов ввода-вывода:

- флаги форматирования как экземпляр типа `fmtflags`;
- информацию о состоянии потока как экземпляр типа `iostate`;
- значение ширины при форматировании данных (переменная типа `int`);

- поле отображаемой точности при форматировании данных (переменная типа `int`);
- объект класса `locale`, обеспечивающий представление информации о форматировании в связи с национальными особенностями.

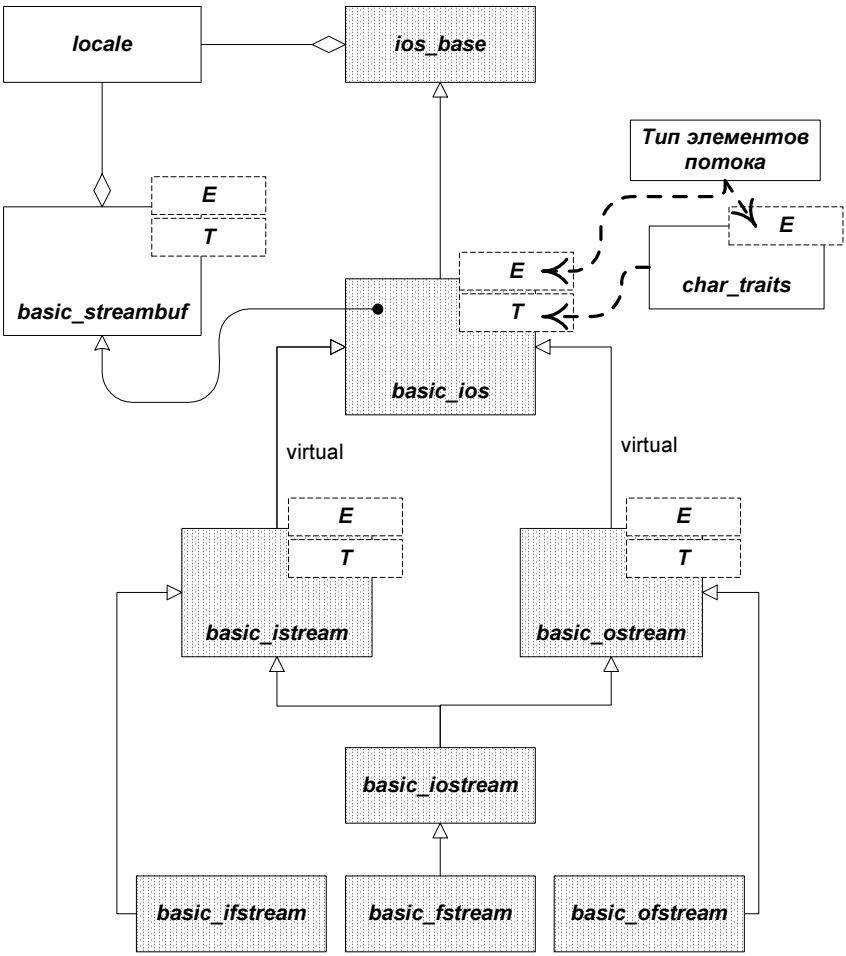


Рис. П2.3. Иерархия классов стандартной библиотеки ввода-вывода в пространстве имен *std*

Ниже приведены фрагменты определения класса `ios_base` в связи с обсуждаемой структурой информации о потоке ввода-вывода.

```
class ios_base {
public:
    //...
```

```
// Флаги форматирования
// (ср. с флагами, определенными в классе ios — см. разд. П2.1)
enum fmflags {
    skipws = 0x0001,      // Пропускать пробелы перед считыванием
    unitbuf = 0x0002,     // Производить очистку буфера после каждого
                          // вывода
    uppercase = 0x0004,   // Выводить в верхнем регистре
    showbase = 0x0008,    // Выводить префикс представления
                          // при выводе не в десятичной системе
                          // счисления
    showpoint = 0x0010,   // Всегда выводить десятичную точку при
                          // выводе плавающих данных
    showpos = 0x0020,     // Выводить знак + при представлении
                          // неотрицательных данных
    left = 0x0040,        // Выравнивать по левому краю поля вывода
    right = 0x0080,       // Выравнивать по правому краю поля вывода
    internal = 0x0100,    // Выравнивать, вставляя заполнящие символы
                          // между знаком и значением
                          // (только при выводе числовых данных)
    dec = 0x0200,         // Использовать представление в десятичном
                          // формате при вводе-выводе
    oct = 0x0400,         // Использовать представление в восьмеричном
                          // формате при вводе-выводе
    hex = 0x0800,         // Использовать представление в шестнадца-
                          // теричном формате при вводе-выводе
    scientific = 0x1000, // Выводить плавающие данные в представлении
                          // с мантиссой и порядком
    fixed = 0x2000,       // Выводить плавающие данные в представлении
                          // с целой и дробной частью
    boolalpha = 0x4000,   // Вводить и выводить булевские значения
                          // (true и false) как строки
    adjustfield = 0x01c0,// Маска для битов, управляющих
                          // выравниванием
    basefield = 0x0e00,   // Маска для битов, управляющих
                          // системой счисления
    floatfield = 0x3000   // Маска для битов, управляющих выводом
                          // плавающих значений
};
// Биты состояния потока
enum iostate {
    goodbit = 0x0,  // Нет ошибок (биты не установлены)
    eofbit = 0x1,   // Бит ситуации "конец файла"
    failbit = 0x2,  // Бит сбоя
```

```
        badbit = 0x4    // Бит серьезной ошибки
    };
    // Режимы открытия
    enum openmode {
        in = 0x01,      // Открыть для ввода
        out = 0x02,     // Открыть для вывода
        ate = 0x04,     // После открытия переместиться на конец файла
        app = 0x08,     // Открыть для записи в конец файла
        trunc = 0x10,   // Создать с очисткой прежнего содержимого
        binary = 0x20   // Открыть как двоичный поток
    };
    // Константы в связи с представлением базы, относительно которой
    // происходит перемещение в потоке
    enum seekdir {
        beg = 0, // Относительно начала
        cur = 1, // Относительно текущей позиции
        end = 2  // Относительно конца
    };
    // ...
private:
    // Закрытые данные
    // (приведены для иллюстрации основных данных, хранящихся
    //  в объекте ios_base)
    iostate _State;  // Состояние потока
    fmtflags _Fmtfl; // Установленные флаги форматирования
    int _Prec;       // Установленное значение точности
    int  _Wide;      // Установленное значение ширины
    locale _Loc;     // Объект в связи с представлением национальных
                     // особенностей
    // ...
};
```

Как явствует из приведенного фрагмента, многие элементы определения класса `ios_base` сходны с соответствующими определениями класса `ios`, изученного в *разд. П2.1*.

Класс *basic_ios* и структура *char_traits*

Шаблонный класс `basic_ios` обеспечивает представление данных и функциональности, общих как для потоков ввода (`basic_istream`), так и для потоков вывода (`basic_ostream`), которые зависят от значений параметров шаблона. Этих параметров два (см. рис. П2.3). Первым параметром шаблона `E` является тип элементов данных, хранящихся в потоке. Примером аргумента шаблона может служить тип `char`, используемый при определении наиболее распро-

страненной разновидности потоков — символьных потоков. Именно (и только) с такими потоками имеет дело "старая" версия стандартной библиотеки ввода-вывода. Вторым параметром шаблона `T` является тип, определяющий свойства элементов потока. В качестве аргумента, подставляемого вместо `T` по умолчанию, используется шаблонный класс `char_traits`, инстанцированный типом `E`. Для символьных потоков это соответствует специализации `char_traits<char>`. Класс `char_traits` определяет различные свойства элементов потока, такие как семантика присваивания, сравнения и копирования элементов типа `E`, определение целочисленного кода, соответствующего экземпляру типа `E`, семантика состояния конца файла для потоков, оперирующих данными пользовательского типа `E` и т. д. Используя терминологию Страуструпа, можно сказать, что основное назначение типа `char_traits` состоит в установлении "связи между закорючками с общепонятными значениями, так называемыми буквами, и числовыми кодами". Класс `char_traits` предоставляет механизм, позволяющий программисту использовать в строках любой символьный набор, при этом сам термин "символ" может трактоваться довольно широко (см. информацию о представлении, кодировании и распознавании символов в литературе по теории формальных грамматик и трансляции). Информация, определяемая в классе `char_traits`, используется различными классами библиотеки ввода-вывода (см. рис. П2.3), а также классом `basic_string` для реализации корректного управления последовательностями из элементов символьного набора.

Замечание

В этой книге мы не рассматриваем подробности работы со строками с использованием средств стандартной библиотеки. Отметим лишь, что реализация стандартных библиотечных возможностей основана на использовании шаблонного класса `basic_string`, определенного в заголовочном файле string в пространстве имен `std`. Используемый в ряде примеров этой книги класс `string` является не чем иным, как специализацией класса `basic_string` с заданием в качестве аргумента шаблона типа `char`. Стандартная библиотека содержит и специализацию для работы со строками символов в формате Unicode — класс `wstring`.

Наиболее распространенными в современной проектной практике являются, по крайней мере, два символьных набора: символьный набор, обеспечивающий представление однобайтовых символов в коде ASCII (на уровне языка поддерживается типом `char`), и символьный набор, обеспечивающий представление двухбайтовых символов в формате Unicode (поддерживается типом `wchar_t`). Для этих двух символьных наборов стандартная библиотека содержит специализированные версии класса `char_traits`:

- тип `char_traits<char>` является явной специализацией шаблона `char_traits` для определения свойств символов типа `char`;

- тип `char_traits<wchar_t>` является явной специализацией шаблона `char_traits` для определения свойств символов типа `wchar_t`.

Итак, объект класса `basic_ios` обеспечивает управление потоком ввода-вывода, содержащем элементы типа `E`, свойства которых определяются в соответствующей специализации класса `char_traits`.

Представленный ниже фрагмент определения класса `basic_ios` иллюстрирует некоторые важнейшие элементы открытого интерфейса класса:

```
template <class E, class T = char_traits<E> >
class basic_ios : public ios_base {
    // Функции для проверки состояния потока
    int good() const; // возвращает 1, если нет ошибок
    int eof() const;  // возвращает 1, если установлен бит
                      // "конец файла"
    int fail() const; // возвращает 1, если установлен бит сбоя
    int bad() const;  // возвращает 1, если установлен бит
                      // серьезной ошибки
    iostate rdstate() const; // возвращает информацию о текущем состоянии
                             // потока

    // Функции для управления состоянием потока
    void clear( iostate state = goodbit );
                // информация о состоянии потока замещается значением
                // ( state | (rdbuf()!=0 ? goodbit : badbit) )
    void setstate( iostate state );
                // устанавливается состояние state посредством вызова
                // функции clear с аргументом ( state | rdstate() )

// Перегрузка операции ! для потоковых объектов
    inline int operator!(); // возвращает 1, если установлен бит сбоя
};
```

Отметим, что объект класса `basic_ios` (обыкновенно выступающий в роли подобъекта объектов классов `basic_istream` и `basic_ostream`) содержит указатель на буфер потока — экземпляр класса `basic_streambuf` (см. рис. П2.3).

Классы *basic_istream* и *istream*

Шаблонный класс `basic_istream` виртуально наследует классу `basic_ios` и описывает свойства и функциональность объекта, обеспечивающего считывание информации из буфера потока. Определение класса содержит набор определений перегруженной операции >> для форматированного ввода данных различных типов, а также комплект методов класса, обеспечивающих считывание информации без форматирования:

```
template <class E, class T = char_traits<E> >
    class basic_istream : virtual public basic_ios<E, T> {
public:
    class sentry;
    explicit basic_istream(basic_streambuf<E, T> *sb);
    virtual ~istream();
    bool ipfx(bool noskip = false);
    void isfx();
    basic_istream& operator>>(basic_istream& (*pf)(basic_istream&));
    basic_istream& operator>>(basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
    basic_istream& operator>>(ios_base<E, T>& (*pf)(ios_base<E, T>&));
    basic_istream& operator>>(basic_streambuf<E, T> *sb);
    basic_istream& operator>>(bool& n);
    basic_istream& operator>>(short& n);
    basic_istream& operator>>(unsigned short& n);
    basic_istream& operator>>(int& n);
    basic_istream& operator>>(unsigned int& n);
    basic_istream& operator>>(long& n);
    basic_istream& operator>>(unsigned long& n);
    basic_istream& operator>>(void *& n);
    basic_istream& operator>>(float& n);
    basic_istream& operator>>(double& n);
    basic_istream& operator>>(long double& n);
    streamsize gcount() const;
    int_type get();
    basic_istream& get(E& c);
    basic_istream& get(E *s, streamsize n);
    basic_istream& get(E *s, streamsize n, E delim);
    basic_istream& get(basic_streambuf<E, T> *sb);
    basic_istream& get(baiic_streambuf<E, T> *sb, E delim);
    basic_istream& getline(E *s, streamsize n)E
    basic_istream& getline(E *s, streamsize n, E delim);
    basic_istream& ignore(streamsize n = 1,
                          int_type delim = T::eof());
    int_type peek();
    basic_istream& read(E *s, streamsize n);
    streamsize readsome(E *s, streamsize n);
    basic_istream& putback(E c);
    basic_istream& unget();
    pos_type tellg();
    basic_istream& seekg(pos_type pos);
    basic_istream& seekg(off_type off, ios_base::seek_dir way);
    int sync();
};
```

Анализируя определения класса, можно заметить, что внешний интерфейс и назначение большинства методов и перегруженных операций сходны с методами и операциями, рассмотренными в *разд. П2.1, П2.3* применительно к классу `istream` из "старой" библиотеки, поэтому изучение средств, предоставляемых классом `basic_istream`, не составляет большого труда.

Стандартная библиотека содержит явную специализацию класса `basic_istream` для работы с потоками, использующими элементы типа `char`, свойства которых определяются в классе `char_traits<char>`. При этом имя `istream` является синонимом типа-специализации:

```
typedef basic_ыistream<char, char_traits<char> > istream;
```

Это позволяет использовать привычное пользователям прежней версии библиотеки ввода-вывода имя `istream` вместо громоздкого имени явной специализации.

Классы *basic_ostream* и *ostream*

Шаблонный класс `basic_ostream` виртуально наследует классу `basic_ios` и описывает свойства и функциональность объекта, обеспечивающего запись информации в буфер потока. Определение класса содержит набор определений перегруженной операции << для форматированного вывода данных различных типов, а также комплект методов класса, обеспечивающих вывод информации без форматирования:

```
template <class E, class T = char_traits<E> >
    class basic_istream : virtual public basic_ios<E, T> {
public:
    class sentry;
    explicit basic_ostream(basic_streambuf<E, T> *sb);
    virtual ~ostream();
    bool opfx();
    void osfx();
    basic_ostream& operator<<(basic_ostream& (*pf)(basic_ostream&));
    basic_ostream& operator<<(basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
    basic_ostream& operator<<(ios_base<E, T>& (*pf)(ios_base<E, T>&));
    basic_ostream& operator<<(basic_streambuf<E, T> *sb);
    basic_ostream& operator<<(const char *s);
    basic_ostream& operator<<(char c);
    basic_ostream& operator<<(bool n);
    basic_ostream& operator<<(short n);
    basic_ostream& operator<<(unsigned short n);
    basic_ostream& operator<<(int n);
    basic_ostream& operator<<(unsigned int n);
```

```
    basic_ostream& operator<<(long n);
    basic_ostream& operator<<(unsigned long n);
    basic_ostream& operator<<(float n);
    basic_ostream& operator<<(double n);
    basic_ostream& operator<<(long double n);
    basic_ostream& operator<<(void * n);
    basic_ostream& put(E c);
    basic_ostream& write(E *s, streamsize n);
    basic_ostream& flush();
    pos_type tellp();
    basic_ostream& seekp(pos_type pos);
    basic_ostream& seekp(off_type off, ios_base::seek_dir way);
};
```

И в этом случае внешний интерфейс и назначение большинства методов и перегруженных операций сходны с методами и операциями, рассмотренными в *разд. П2.1, П2.2* применительно к классу `ostream` из "старой" библиотеки, поэтому изучение средств, предоставляемых классом `basic_ostream`, также не составит труда.

Стандартная библиотека содержит явную специализацию класса `basic_ostream` для работы с потоками, использующими элементы типа `char`, свойства которых определяются в классе `char_traits<char>`. При этом имя `ostream` является синонимом типа-специализации:

```
typedef basic_ostream<char, char_traits<char> > ostream;
```

Классы *basic_iostream* и *iostream*

Шаблонный класс `basic_iostream` наследует классам `basic_istream` и `basic_ostream`. Класс `basic_iostream` описывает свойства и функциональность объекта, обеспечивающего запись информации посредством использования подобъекта базового класса `basic_ostream` и считывание информации посредством использования подобъекта базового класса `basic_istream`.

Стандартная библиотека содержит явную специализацию класса `basic_iostream` для работы с потоками, использующими элементы типа `char`, свойства которых определяются в классе `char_traits<char>`. При этом имя `iostream` является синонимом типа-специализации:

```
typedef basic_iostream<char, char_traits<char> > iostream;
```

Организация работы с файловыми потоками

Общее устройство библиотеки ввода-вывода, основанной на использовании шаблонных классов и их явных специализаций, должно быть понятно из

предшествующих разделов. Организация работы с файловыми потоками следует тем же проектным принципам, что и организация работы с потоками общего вида. Например, для работы с файловым потоком ввода необходимо инстанцировать шаблон `basic_ifstream`, являющийся классом, производным от `basic_istream`. Для работы с символьными файловыми потоками в стандартной библиотеке предусмотрена явная специализация класса `basic_ifstream`, а также доступен синоним `ifstream`:

```
typedef basic_ifstream<char, char_traits<char> > ifstream;
```

Для работы с файловым потоком вывода необходимо инстанцировать шаблон `basic_ofstream`, являющийся классом, производным от `basic_ostream`. Для работы с символьными файловыми потоками в стандартной библиотеке предусмотрена явная специализация класса `basic_ofstream`, а также доступен синоним `ofstream`:

```
typedef basic_ofstream<char, char_traits<char> > ofstream;
```

Наконец, для работы с файловым потоком ввода-вывода необходимо инстанцировать шаблон `basic_fstream`, являющийся классом, производным от `basic_iostream`. Для работы с символьными файловыми потоками в стандартной библиотеке предусмотрена явная специализация класса `basic_fstream`, а также доступен синоним `fstream`:

```
typedef basic_fstream<char, char_traits<char> > fstream;
```

Резюме

Система ввода-вывода в том виде, в каком она реализована в составе стандартной библиотеки C++, ориентирована на использование в наиболее употребительных случаях организации ввода-вывода (таких, как ввод-вывод символов и строк) и дополнительно позволяет распространить ввод-вывод на работу с типами, определенными пользователем. Таким образом, программист может использовать стандартные решения для реализации нестандартных систем ввода-вывода, работающих с другими разновидностями потоков, учитывающие национальные особенности представления информации при вводе и выводе, использующие другие стратегии буферизации, правила форматирования и т. п.

ПРИЛОЖЕНИЕ 3

Описание компакт-диска

На компакт-диске приведены все пронумерованные листинги примеров программ, рассматриваемых в книге, а также некоторые дополнительные программы, необходимые для тестирования анализируемых примеров.

В корневом каталоге диска находятся файлы, обеспечивающие автоматический запуск диска, HTML-страницы, облегчающие работу с диском, а также некоторые вспомогательные файлы.

Файл *readme.txt* содержит данный текст.

В каталоге *Samples* содержатся тексты примеров программ, организованные в соответствии с порядком следования глав и нумерацией листингов в книге. Этот каталог включает следующие подкаталоги:

- *Samples\02* — с примерами программ к *главе 2 "Элементы объектной модели"*;
- *Samples\03* — с примерами программ к *главе 3 "Класс как основной механизм абстракции"*;
- *Samples\04* — с примерами программ к *главе 4 "Наследование и иерархии классов"*;
- *Samples\05* — с примерами программ к *главе 5 "Объекты классов и полиморфизм"*;
- *Samples\06* — с примерами программ к *главе 6 "Обработка ошибок на основе использования механизма исключений"*;
- *Samples\07* — с примерами программ к *главе 7 "Множественное наследование и интерфейсы"*;
- *Samples\08* — с примерами программ к *главе 8 "Пространства имен в связи с модульностью и иерархией"*;

- *Samples\09* — с примерами программ к *главе 9 "Введение в обобщенное программирование"*;
- *Samples\10* — с примерами программ к *главе 10 "Организация вычислительного процесса в управляемых средах"*;
- *Samples\11* — с примерами программ к *главе 11 "Развитие объектно-ориентированной модели управления типами"*;
- *Samples\12* — с примерами программ к *главе 12 "Введение в компонентное программирование"*.

В каждом каталоге, соответствующем главе книги, располагается набор каталогов, организованный в соответствии с нумерацией листингов, например, в каталоге Samples\08\Listing_8_02 содержатся файлы, относящиеся к примеру, представленному в листинге 8.2.

Если для работы с примерами программ требуется выполнение определенной последовательности действий, или в том случае, когда каталог листинга содержит дополнительные файлы или подкаталоги, в каталог листинга помещается файл readme.txt, содержащий необходимую дополнительную информацию.

Например, файл readme.txt, расположенный в каталоге Samples\08 \Listing_8_02, содержит следующий текст:

```
Описание содержимого этого каталога

  Каталог org\myserver\mypackage содержит следующие файлы:
  HelloDate.jar - скомпилированный Java-пакет, в котором
                  определяется класс HelloDate
  HelloDate.java - Текст определения класса HelloDate на Java
  (см. раздел 8.3 "Пространства имен в Java")

  Каталог HelloDateApp содержит следующие файлы:
  MyApp.java - клиентская программа, использующая класс HelloDate;
  jcwrap.bat - BATCH-файл, устанавливающий значение переменной окружения
               CLASSPATH и запускающий компилятор Java
  jwrap.bat  - BATCH-файл, устанавливающий значение переменной окружения
               CLASSPATH и запускающий виртуальную машину Java
               (на компьютере должны быть установлены средства разработки
                Java, например, Sun Java Developer's Kit)

  Порядок работы

  1. Скопируйте содержимое этого каталога на жесткий диск.

  2. В каталоге HelloDateApp исполните batch-файл для компиляции
     Java-программы в байт-код:
     jcwrap.bat MyApp.java
```

3. Убедитесь, что в каталоге HelloDateApp появился файл MyApp.class

4. Запустите интерпретатор байт-кода Java:
   jwrap.bat HelloDateApp

5. Убедитесь, что программа печатает приветствие, текущую дату и время.

Кроме примеров программ, на компакт-диске содержится информация об авторе этой книги. В каталоге Images содержатся файлы изображений, использованных на HTML-страницах.

Для удобства просмотра представленной информации, при загрузке диска автоматически запускается HTML-страница default.html.

Список источников

Ссылки на зарубежные источники даются по возможности в их оригинальном наименовании. Одновременно приводятся библиографические сведения о переводном издании, если оно имеется и известно автору.

1. **Austern, 1999**

   Austern, Matthew H. Generic Programming and the STL: Using and Extending the C++ Standard Template Library. — Addison-Wesley, 1999.

   Остерн М. Обобщенное программирование и STL: Использование и наращивание стандартной библиотеки шаблонов / Пер. с англ. — СПб.: Невский диалект, 2004. — 544 с., ил.

2. **Booch, 1994**

   Booch, Grady. Object-Oriented Analysis and Design with Applications. 2nd edition. Addison Wesley, 1994.

   Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. 2-е изд. / Пер. с англ. — М.: "Издательство Бином", СПб.: "Невский диалект", 2000. — 560 с., ил.

3. **Booch, Rumbaugh, Jacobson, 1999**

   Booch, Grady. Rumbaugh, James. Jacobson, Ivar. The Unified Modeling Language. User Guide. Addison Wesley, 1999.

   Буч Г., Рамбо Дж., Якобсон А. Язык UML. Руководство пользователя / Пер. с англ. — М.: ДМК Пресс, 2000. — 432 с., ил. (Серия "Для программистов")

4. **Brooks, 1995**

   Brooks, Frederick P., Jr. The Mythical Man-Month. Essays on Software Engineering. Anniversary Edition. Addison-Wesley, 1995.

Брукс Ф. Мифический человеко-месяц, или как создаются программные системы / Пер. с англ. — СПб.: Символ, 2000. — 304 с., ил.

5. **Chappell**, 2001

   Chappell, David. Understanding .NET's Common Type System. Inform-IT Columns. 14/12/2001.

6. **Dahl, Dijkstra, Hoare, 1972**

   Dahl O.-J., Dijkstra E.W., Hoare C.A.R. Structured Programming. Academic Press. London and New York, 1972.

   Дал У., Дейкстра Э., Хоор К. Структурное программирование / Пер. с англ. — М.: Мир, 1975. — 248 с.

7. **Eckel, 2000-1**

   Eckel, Bruce. Thinking in C++. 2nd ed. Volume One: Introduction to Standard C++. Prentice Hall Inc., 2000.

   Эккель Б. Философия C++. Введение в стандартный C++. 2-е изд. / Пер. с англ. — СПб: Питер, 2004. — 572 с., ил.

8. **Eckel, 2000-2**

   Eckel, Bruce. Thinking in Java. 2nd ed. Prentice Hall Inc., 2000.

   Эккель Б. Философия Java / Пер. с англ. — СПб.: Питер, 2001. — 880 с., ил.

9. **Eckel, Allison, 2003**

   Eckel, Bruce. Allison, Chuck. Thinking in C++. 2nd ed. Volume Two: Practical Programming. Prentice Hall Inc., 2003.

   Эккель Б., Эллисон Ч. Философия C++. Практическое программирование / Пер. с англ. — СПб: Питер, 2004. — 608 с., ил.

10. **Ellis, Stroustrup, 1990**

    Ellis, Margaret A. and Stroustrup, Bjarne. The Annotated C++ Reference Manual. Addison-Wesley. Reading, Mass, 1990.

    Эллис М., Страуструп Б. Справочное руководство по языку C++ с комментариями / Пер. с англ. — М.: Мир, 1992. — 445 с., ил.

11. **Forsyth, 1984**

    Expert Systems. Principles and case studies. Edited by Richard Forsyth. London. New York. Chapman and Hall, 1984.

    Экспертные системы. Принципы работы и примеры: Пер. с англ. / А. Брукинг, П. Джонс, Ф. Кокс и др.; под ред. Р. Форсайта. — М.: Радио и связь, 1987. — 224 с., ил.

12. **Fowler, 1999**

Fowler, Martin. Beck, Kent. Refactoring. Improving the Design of Existing Code. Addison-Wesley Pub. Co., 1999.

Фаулер М. Рефакторинг. Улучшение существующего кода / Пер. с англ. — М.: Символ-Плюс, 2003. — 432 с.

13. **Franz, 1998**

Franz, Michael. Code Generation On-the-Fly: A Key for Portable Software. Zurich, 2000.

Франц М. Динамическая кодогенерация: ключ к разработке переносимого программного обеспечения / ИнфоАрт, 1998: http://www.uni-vologda.ac.ru /oberon/infoart/otherfly.htm.

14. **Graham, Knuth, Patashnik, 1994**

Graham R., Knuth D., Patashnik O. Concrete Mathematics. A Foundation for Computer Science. Addison-Wesley, 1994.

Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики / Пер. с англ. — М.: Мир, 1998. — 703 с., ил.

15. **Gunnerson, 2000**

Gunnerson, Eric. A Programmer's Introduction to C#. Apress, 2000.

Гуннерсон Э. Введение в C#. Библиотека программиста / Пер. с англ. — СПб.: Питер, 2001. — 304 с., ил.

16. **Holub, 1995**

Holub, Allen I. Enough Rope to Shoot Yourself in the Foot. Rules for C and C++ Programming. McGraw-Hill, 1995.

Голуб А. И. C и C++. Правила программирования / Пер. с англ. — М.: Восточная Книжная Компания, 1996. — 272 с.

17. **King, 1997**

King, Gary Warren. Object-Oriented is Really Better than Structured, 1997: www-eksl.cs.umass.edu/~gwking/whyoop.htm.

Кинг Г. Объектно-ориентированный подход действительно лучше структурного / Пер. с англ. А. И. Легалова, 1997: www.softkraft.ru/design /woop/woop01.shtml.

18. **Mesarovic, Takahara, 1989**

Mesarovic M.D., Takahara Y. Abstract Systems Theory. Springer Verlag, 1989.

19. **Meyer, 1992**

Meyer, Bertran. Eiffel the Language. Prentice-Hall, Englewood Cliffs, NJ., 1992.

20. **Meyer, 2001**

Meyer, Bertran. Software Engineering in the Academy // Computer, May 2001. IEEE Computer Society, 2001.

Мейер Б. Программная инженерия как предмет обучения // "Открытые системы", № 7—8, 2001.

21. **Meyers, 1996**

Meyers, Scott. More Effective C++: 35 New Ways To Improve Your Programs and Designs. Addison-Wesley, 1996.

Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших проектов / Пер. с англ. — М.: ДМК Пресс, 2000. — 304 с.

22. **Meyers, 1998**

Meyers, Scott. Effective C++: 50 Specific Ways To Improve Your Programs and Designs. Addison-Wesley, 1998.

Мейерс С. Эффективное использование C++. 50 рекомендаций по улучшению ваших проектов / Пер. с англ. — М.: ДМК, 2000. — 240 с.

23. **Meyers, 2000**

Meyers, Scott. How Non-Member Functions Improve Encapsulation // C/C++ Users Journal, Feb. 2000: www.cuj.com/articles/2000/0002.

Мейерс С. Как функции, не являющиеся методами, улучшают инкапсуляцию / Пер. с англ. А. И. Легалова: www.softcraft.ru/coding/sm/sm01.shtml.

24. **Microsoft, 1999**

Desktop Applications with Microsoft Visual C++ 6.0. Training Kit. Hands-On, Self-Paced Training. Microsoft Press, 1999.

Разработка приложений на Microsoft Visual C++ 6.0. Учебный курс. Сертификационный экзамен 70-016: официальное пособие Microsoft для самостоятельной подготовки / Пер. с англ. — М.: Издательско-торговый дом "Русская редакция", 2000. — 576 с., ил.

25. **Minsky, 1967**

Minsky, Marvin L. Finite and Infinite Machines. Prentice-Hall, Inc., 1967.

Минский М. Вычисления и автомат / Пер. с англ. — М.: Мир, 1971. — 364 с.

26. **Naughton, Schildt, 1999**

Naughton, Patric. Schildt, Herbert. Java 2: The Complete Reference, Third Edition. Osborne / McGraw Hill, 1999.

Ноутон П., Шилдт Г. Java 2 / Пер. с англ. — СПб.: БХВ-Петербург, 2001. — 1072 с., ил.

27. **Platt, 2001**

Platt, David S. Introducing Microsoft .NET. Microsoft Press, 2001.

Платт Д. Знакомство с Microsoft .NET / Пер. с англ. — М.: Издательско-торговый дом "Русская редакция", 2001. — 240 с., ил.

28. **Pohl, 1997**

Pohl, Ira. Object-Oriented Programming Using C++. 2nd edition. Addison Wesley, 1997.

Пол И. Объектно-ориентированное программирование на C++. 2-е изд. / Пер. с англ. — М.: "Издательство Бином", СПб.: "Невский диалект", 1999. — 462 с., ил.

29. **Richter, 2002**

Richter, Jeffrey. Applied Microsoft .Net Framework Programming. Microsoft Press, 2002.

Рихтер Дж. Программирование на платформе Microsoft .Net Framework / Пер. с англ. — М.: Издательско-торговый дом "Русская редакция", 2002. — 512 с., ил.

30. **Rogerson, 1997**

Rogerson, Dale. Inside COM. Microsoft Press, 1997.

Роджерсон Д. Основы COM / Пер. с англ. — М.: Издательско-торговый дом "Русская редакция", 2000. — 400 с., ил.

31. **Sebesta, 2002**

Sebesta, Robert W. Concepts of Programming Languages. Addison-Wesley Inc., 2002.

Себеста Р. У. Основные концепции языков программирования / Пер. с англ. — М.: Издательский дом "Вильямс", 2001. — 672 с., ил.

32. **Solomon, Russinovitch, 2000**

Solomon, David A. Russinovich, Mark E. Inside for Microsoft Windows 2000. 3rd Ed. Microsoft Press, 2000.

Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. Мастер-класс. / Пер. с англ. — СПб.: Питер; М.: Издательско-торговый дом "Русская редакция", 2001. — 752 с., ил.

33. **Stroustrup, 1994**

Stroustrup, Bjarne. The Design and Evolution of C++. Addison-Wesley, 1994.

Страуструп Б. Дизайн и эволюция языка C++ / Пер. с англ. — М.: ДМК Пресс, 2000. — 448 с., ил. (Серия "Для программистов").

34. **Stroustrup, 1998**

Stroustrup, Bjarne. An Overview of the C++ Programming Language. Handbook of Object Technology. CRC Press, 1998.

35. **Stroustrup, 2000**

Stroustrup, Bjarne. The C++ Programming Language. Special Edition. Addison-Wesley, 2000.

Страуструп Б. Язык программирования C++. Специальное изд. / Пер. с англ. — М.: "Издательство Бином"; СПб.: "Невский диалект", 2001. — 1099 с., ил.

36. **Weinberg, 1988**

Weinberg, Gerald. Understanding the Professional Programmer. Dorset House Publishing, 1988.

37. **Young, 2000**

Young, Michael. Microsoft XML: Step by step. Microsoft Press, 2000.

Янг М. Microsoft XML. Шаг за шагом: Практическое пособие / Пер. с англ. — М.: Издательство ЭКОМ, 2001. — 384 с., ил.

38. **Богатырев, 1997**

Богатырев Р. Уроки аварии ракеты-носителя Ariane 5 // Computer Week-Moscow, № 38, 1997.

39. **Борк, 2001**

Борк Дж. Будущее сетевых приложений // Computer World/Russia, № 37, 2001.

40. **Власовских, Пышкин, 2003**

Власовских А. С., Пышкин Е. В. Механизмы отделения логической структуры информации от ее представления в электронном документообороте // XXXI Неделя науки СПбГПУ. Ч. VII: Материалы межвузовской научной конференции. — СПб.: Изд-во СПбГПУ, 2003.

41. **Давыдов, 2001**

Давыдов В. Г. Теория и технология программирования. Конспект лекций (ч. 2). — СПб.: Изд-во СПбГТУ, 2000. — 228 с., ил.

42. **Давыдов, 2002**

Давыдов В. Г. Теория и технология программирования. Часть 1: Учебное пособие. — СПб.: Изд-во СПбГТУ, 2002. — 318 с., ил. (Серия "Современные компьютеризированные образовательные технологии").

43. **Давыдов, 2003**

Давыдов В. Г. Программирование и основы алгоритмизации: Учебное пособие. — М.: Высшая школа, 2003. — 447 с., ил.

44. **Добрынин, 2003**

Добрынин В. Ю. Технологии компонентного программирования / Учебно-методическое пособие. — СПб.: Изд-во СПбГУ, 2003: http://oasis.apmath.spbu.ru/~vdobr/COM/.

45. **Дубова, 1999**

Дубова Н. COM или CORBA? Вот в чем вопрос // "Открытые системы", № 3, 1999.

46. **Дубовцев, 2004**

Дубовцев А. В. Microsoft .NET в подлиннике / Под ред. Е. В. Пышкина. — СПб.: БХВ-Петербург, 2004. — 704 с., ил.

47. **Зуев, Кротов, Сухомлин, 1996**

Зуев Е. А., Кротов А. Н., Сухомлин В. А. Язык программирования C++: этапы эволюции и современное состояние // Тезисы доклада Первой российской конференции "Индустрия программирования '96". — М.: 1996. (см. также версию на CIT Forum, 1997).

48. **Зуев, 1997**

Зуев Е. А. Редкая профессия // PC Magazine/Russian Edition. Спецвыпуск № 5 (75), 1997.

49. **Легалов, 2000**

Легалов А. И. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? — Красноярск: 2000. Деп. рук. № 622-В00. Деп. в ВИНИТИ 13.03.2000. — 43 с.

50. **Легалов, 2001**

Легалов А. И. О стрельбе по множественному наследованию // "Открытые системы", № 5—6, 2001.

51. **Легалов, 2002**

Легалов А. И. Эволюция мультиметодов при процедурном подходе (2002): www.softcraft.ru/coding/evp/evp.shtml.

52. **Лекарев, 1997**

Лекарев М. Ф. Визуальный формализм для разработки программного обеспечения. — СПб.: Изд-во СПбГТУ, 1997. — 95 с.

53. **Лекарев, Давыдов, 2002**

Лекарев М. Ф., Давыдов В. Г. Проектирование Windows-приложений: Учебное пособие. — СПб.: Изд-во СПбГТУ, 2002. — 224 с., ил.

54. **Одинцов, 2002**

Одинцов И. С. Профессиональное программирование. Системный подход. — СПб.: БХВ-Петербург, 2002. — 512 с., ил.

55. **Попов и др., 1990**

Искусственный интеллект: В 3 кн. Кн. 1. Системы общения и экспертные системы: Справочник / Под ред. Э. В. Попова. — М.: Радио и связь, 1990. — 464 с., ил.

56. **Пышкин, 2003-1**

Pychkine, Evgueni V. Understanding the Object Model. Prerequisites for the crash course of software engineering. Ajankohtaista-aktuellt. Central Ostrobothnia Polytechnics, 2003. — 29 p.

57. **Пышкин, 2003-2**

Пышкин Е. В. Основные концепции объектно-ориентированного программирования: Учебное пособие. — СПб: Изд-во СПбГПУ, 2003. — 302 с., ил.

58. **Пышкин, 2004-1**

Pychkine, Evgueni. An Introduction to the Microsoft .Net Framework Architecture and Tools. Supplementary materials for the crash course of software engineering. Central Ostrobothnia Polytechnic, 2004. — 63 p.

59. **Пышкин, 2004-2**

Пышкин Е. В. Функциональные особенности электронных документов и модели информационного обмена в электронном документообороте // Вычислительные, измерительные и управляющие системы: Сборник научных трудов аспирантов и молодых ученых факультета технической кибернетики / Под. ред. Ю. Б. Сениченкова. — СПб.: Изд-во СПбГПУ, 2004. — с. 79—83.

60. **Рассохин, 1993**

Рассохин Д. От Си к Си++. — М.: Изд-во "Эдель", 1993. — 128 с., ил.

61. **Труб, 2001**

Труб И. О проблемах множественного наследования // "Открытые системы", № 2, 2001.

62. **Фролов, 1987**

Фролов Г. В., Олюнин В. Ю. Практический курс программирования на языке PL/1. — М.: Наука, Гл. ред. физ.-мат. лит., 1986. — 384 с.

63. **Шалыто, 1998**

Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. — СПб.: Наука, 1998. — 628 с.

64. **Шамгунов, Корнеев, Шалыто, 2004**

Шамгунов Н. Н., Корнеев Г. А., Шалыто А. А. State Machine. Новый паттерн объектно-ориентированного программирования // "Информационно-управляющие системы", № 5, 2004.

Предметный указатель

A

API (Application Programming Interface) 457
ATL (Active Template Library) 542

C

CCI (Common Compiler Infrastructure) 22
CLR (Common Language Runtime) 418, 419, 546
CLS (Common Language Specification) 545, 554
COM (Component Object Model) 484, 539
CORBA (Common Object Request Broker Architecture) 542
CTS (Common Type System) 545, 551

D

DLL (Dynamic Link Library) 334, 486, 513

E

EJB (Enterprise JavaBeans) 542

F

FCL (Framework Class Library) 467, 555

G

GAC (Global Assembly Cache) 550
GC (Garbage Collection) 417, 432
GUID (Global Unique Identifier) 497, 540

I

IDL (Interface Description Language) 531
IL (Intermediate Language) 418

J

JNI (Java Native Interface) 412

M

MFC (Microsoft Foundation Classes) 457

R

RTTI (Run Time Type Information) 204

S

SOAP (Simple Object Access Protocol) 545
STL (Standard Template Library) 8, 368, 386

U

UML (Unified Modeling Language) 7

V

VCL (Visual Component Library) 459

X

XML (eXtensible Markup Language) 481

А

Автоматное программирование 20
Атрибуты *См.* Специализированные атрибуты

Б

Безопасный код 408, 418
◊ верифицируемый код 422
◊ код, безопасный по отношению к типам 422

В

Ввод-вывод (потоки C++) 573
◊ библиотека iostream 600
◊ библиотека iostream.h 574
◊ буферизация 580, 594, 596
◊ ввод данных встроенных типов 589
◊ ввод данных пользовательских типов 591
◊ ввод символов и строк 589
◊ версии библиотеки 573, 600
◊ вывод данных встроенных типов 580
◊ вывод данных пользовательских типов 586
◊ вывод символов и строк 580
◊ иерархия шаблонных классов 600
◊ класс basic_ios 604
◊ класс basic_iostream и его специализации 609
◊ класс basic_istream и его специализации 606
◊ класс basic_ostream и его специализации 608
◊ класс char_traits 605
◊ класс ios 575, 600
◊ класс ios_base 601
◊ класс istream и производные 579
◊ класс ostream и производные 578
◊ манипуляторы ввода 590, 600
◊ манипуляторы вывода 582, 600
◊ режимы открытия потоков 576, 602
◊ файловый ввод-вывод 579, 593, 610
◊ флаги состояния потока 576, 577, 602
◊ форматирование 577, 582, 590, 602
Виртуальные функции 150, 177, 436
◊ версии виртуальных функций 439
◊ виртуальный деструктор 193
◊ вызов виртуальной функции 197
◊ вызов невиртуального метода вместо виртуального 200
◊ определение 177
◊ спецификатор new (C#) 438
◊ таблица виртуальных функций *См.* Динамическое (позднее) связывание
◊ функции-не члены класса, подобные виртуальным 201
◊ чисто виртуальная функция 193
Встроенные функции (inline) 105

Д

Делегаты 450
◊ класс Delegate 453
◊ создание экземпляра делегата 463
◊ цепочка делегатов 453
Деструктор класса 88, 433
Динамическая компоновка 486
◊ версии динамических библиотек 551
Динамическое (позднее) связывание 177, 435
◊ определение типа во время выполнения 204
◊ таблица виртуальных функций 185

Документация 478
◊ автоматизация создания и модификации 479
◊ встроенный XML 481
◊ документирующие комментарии 479, 481
◊ соответствие кода и документации 479

И

Интерфейс 163, 193, 307, 489
◊ IUnknown 491
◊ запрос интерфейса у компонента 489, 491, 493
◊ идентификация интерфейсов компонента COM 497
◊ интерфейс COM 491
◊ множественное наследование интерфейсов 312, 316
◊ наследование с интерфейсами 308
◊ определение интерфейса 311
◊ понятие 47
◊ реализация интерфейсов 308, 312
◊ учет ссылок на интерфейсы 498
Исключения 234, 549
◊ bad_alloc 10
◊ блок finally 272
◊ в деструкторах 264
◊ в конструкторах 258
◊ встроенные типы языка 242
◊ генерация исключения 236
◊ группирование исключений 249
◊ копирование объектов исключений 238
◊ обработка исключений 236, 252
◊ ошибки резервирования динамической памяти 257
◊ повторная генерация 237, 252
◊ раскрутка стека 265
◊ спецификация исключений 268, 273
◊ ссылка на объект исключения 240, 250
◊ типы, определяемые пользователем 244

К

Класс:
◊ абстрактный класс 192, 307, 314
◊ вложенный класс 121
◊ дружественный доступ 111, 119
◊ закрытые члены класса 68, 70
◊ класс как область видимости 96
◊ конкретный тип 167
◊ константы и константные функции 91
◊ контейнерный класс 364
◊ локальный класс 121
◊ методы класса 66
◊ определение класса 75
◊ открытые члены класса 68, 70
◊ понятие 27
◊ спецификация и реализация класса 76
◊ статические члены класса 93, 99
◊ управление доступом 67
◊ управление доступом к членам класса 158
◊ шаблонный класс 359
Компонент программного обеспечения 483, 489
◊ построение компонента COM 514
◊ построение сборки .NET 546
◊ реализация интерфейса 491
◊ создание экземпляра компонента 491, 499, 517, 528, 548
Компонентная архитектура 485, 488, 493, 506
◊ Microsoft .NET 546
Компонентное программирование 483
◊ архитектура клиент-сервер 489
◊ модель COM 487, 513, 539
◊ модель Microsoft .NET 544, 545
Конструктор класса 72
◊ в производном классе 154
◊ закрытый конструктор 84
◊ инициализаторы 97, 304
◊ конструктор как неявное преобразование 84
◊ конструктор по умолчанию 83, 98
◊ копирующий конструктор 126
◊ перегрузка конструкторов 81
◊ спецификатор explicit 86
◊ статический конструктор 100
Контейнеры 364
◊ итераторы 365, 369
◊ обработка контейнеров (алгоритмы) 385
◊ специализированные контейнеры 364
◊ стандартные контейнеры 368

Л

Логическое программирование 20

М

Метаданные 418, 464, 475

Н

Наследование 36, 51, 142
◊ виртуальные базовые классы 300
◊ виртуальные функции 54
◊ иерархия классов ввода-вывода C++ 574, 601
◊ иерархия классов 156, 458
◊ многократное использование интерфейса 163
◊ множественное наследование 283
◊ наследование интерфейса 540
◊ наследование реализации 540
◊ наследование с интерфейсами 48
◊ неизменные методы и классы 167
◊ однокоренная иерархия классов 164, 388
◊ переопределение методов 151
◊ повышающее приведение типа 148
◊ проблема хрупкого базового класса 439
◊ ромбовидное наследование 299
◊ сокрытие имен 152, 165
◊ спецификаторы доступа 159
◊ структура объектов 40, 146
◊ шаблонные классы 390, 600

О

Обобщенное программирование 352
◊ в стиле C 352
◊ инстанцирование 358
◊ стандартная библиотека шаблонов 386
◊ шаблонные классы 359
◊ шаблонные функции 356
Обработка ошибок 219
◊ анализ кода HRESULT в компонентном программировании 491
◊ анализ кода ошибки 221
◊ глобальные переменные состояния 224
◊ использование функций обратного вызова 229
◊ многоуровневая обработка 233
◊ обработка исключений
См. Исключения
◊ функции-обработчики 228
Общеязыковая среда исполнения (CLR) 418, 419
Объект класса:
◊ запрет копирования 133
◊ инстанцирование шаблона 362
◊ копирование объектов 124
◊ подсчет ссылок 93
◊ полиморфное поведение 55, 173, 177, 206, 436
◊ размещение в динамической памяти 29
◊ размещение в стеке 29
◊ свойства 27
Объектная модель 15, 26
◊ абстрагирование 27
◊ инкапсуляция 30
◊ модульность 58, 321, 546
Объектно-ориентированное программирование 15, 20
Операторы и операции 11
Отношения классов 35
◊ агрегирование 50
◊ ассоциация 41
◊ зависимость 46
◊ композиция 50
◊ обобщение 36
◊ отношение обобщения 142
◊ реализация 47
Отражение (рефлексия) 465, 468, 474
◊ вызов методов 477
◊ извлечение информации о типе 474
◊ создание экземпляров типов 476

П

Перегрузка методов класса 106
◊ выбор метода 107
Перегрузка операций 108, 112
◊ друзья класса 111
◊ операции new и delete 118
◊ операции ввода-вывода 201
◊ операции-не члены класса 110
◊ операции-члены класса 108
◊ операция "запятая" 109

◊ операция присваивания 126
◊ перегрузка арифметических операций 108
◊ перегрузка логических операций 109
Преобразование типа:
◊ const_cast 213
◊ dynamic_cast (динамическое преобразование) 204, 210, 212
◊ reinterpret_cast 213
◊ static_cast (статическое преобразование) 211
◊ небезопасное статическое преобразование 175
◊ перекрестное приведение типа 206
◊ понижающее преобразование 204
◊ преобразование производного типа к базовому 148
Приложение Windows:
◊ обработка сообщений 456
◊ объектно-ориентированная архитектура 457
◊ первичный поток 456
◊ поток 456
◊ функция WinMain 456
ПриложениеWindows 456
Промежуточное программное обеспечение 542
Пространства имен 321
◊ using-директива 326
◊ using-объявление 326, 331
◊ глобальное пространство имен 321
◊ конфликты имен 329
◊ объединение пространств имен 331
◊ определение пространства имен 324
◊ пакеты Java 341
◊ псевдонимы 330
◊ реализация в платформе .NET 550
◊ сборки (платформа .NET) 347
◊ совместимость со старым кодом 337
◊ стандартное пространство имен 337, 574
◊ управление версиями 333

Р

Реестр Windows 523
◊ регистрация библиотеки типов 536
◊ регистрация компонента COM 525
Рефлексия *См.* Отражение

С

Сборка мусора 417, 432
◊ метод finalize() 433
Свойства 445, 485
◊ метод-аксессор get 446
◊ метод-аксессор set 446
◊ свойства с параметром (индексаторы) 447
События 455, 459, 485
◊ обработка сообщений 455
◊ привязка событий к обработчикам 458, 459
Специализированные атрибуты 464
◊ атрибут Obsolete 466
◊ атрибуты, определяемые пользователем 471
◊ доступ в ходе выполнения программы 468
Ссылочные типы 425
Стандартное соглашение о вызове функции 489
Статическое (раннее) связывание 173
Строки 10
◊ ANSI C 10
◊ string 10
◊ неизменяемые строки 133
◊ проектирование модельного класса 127
Структурное программирование 18

Т

Типы данных 424
◊ встроенные (элементарные) типы 425, 428
◊ размерные типы 425

У

Указатель this 101
Управляемый код 408, 423
◊ взаимодействие с неуправляемым кодом 412
◊ компиляция 410
◊ метаданные *См.* Метаданные
◊ модель Java 409
◊ модель Microsoft .NET 418
Продолжение рубрики см. на с. 628

Управляемый код *(прод.)*:
◊ ограничения 417
◊ сборки .NET 546, 550
Устойчивый код 408

Ф

Фабрика класса 517, 530
Функциональное программирование 20
Функционально-иерархическая декомпозиция 19

Я

Язык программирования 16
◊ генеалогия языков 23