Философия C++

Введение в стандартный C++

БРЮС ЭККЕЛЬ

2-е ИЗДАНИЕ

ПИТЕР

BRUCE ECKEL

Thinking in C++

Second Edition. Volume One: Introduction to Standard C++

БРЮС ЭККЕЛЬ

Философия C++

Введение в стандартный C++

2-е ИЗДАНИЕ

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2004

ББК 32.973.2-018
УДК 681.3.06
Э38

Эккель Б.

Э38 Философия С++. Введение в стандартный С++. 2-е изд. — СПб.: Питер, 2004. — 572 с.: ил.

ISBN 5-94723-763-6

В книге «Философия С++» последовательно и методично излагаются вопросы использования объектно-ориентированного подхода к созданию программ. Автор не просто описывает различные проблемы и способы их решения, он раскрывает перед читателем особый образ мышления, не владея которым невозможно комфортно чувствовать себя в объектно-ориентированной среде.

Это одна из тех книг, которые обязательно должен прочесть каждый, кто всерьез занимается разработкой программного обеспечения в С++.

ББК 32.973.2-018
УДК 681.3.06

Права на издание получены по соглашению с Prentice Hall, Inc. Upper Sadle River, New Jersey 07458.

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

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

ISBN 0-13-979809-9 (англ.)
ISBN 5-94723-763-6

© 2000 by Bruce Eckel, MindView, Inc.
© Перевод на русский язык, ЗАО Издательский дом «Питер», 2004
© Издание на русском языке, оформление, ЗАО Издательский дом «Питер», 2004

Краткое содержание

Предисловие ... 18
Глава 1. Знакомство с объектами ... 28
Глава 2. Создание и использование объектов ... 67
Глава 3. Элементы C в языке C++ ... 92
Глава 4. Абстрактное представление данных ... 167
Глава 5. Скрытие реализации ... 196
Глава 6. Инициализация и зачистка ... 213
Глава 7. Перегрузка функций и аргументы по умолчанию ... 231
Глава 8. Константы ... 248
Глава 9. Подставляемые функции ... 276
Глава 10. Механизм контроля имен ... 299
Глава 11. Ссылки и копирующий конструктор ... 331
Глава 12. Перегрузка операторов ... 356
Глава 13. Динамическое создание объектов ... 401
Глава 14. Наследование и композиция ... 427
Глава 15. Полиморфизм и виртуальные функции ... 459
Глава 16. Знакомство с шаблонами ... 503
Приложение А. Стиль программирования ... 545
Приложение Б. Рекомендации по программированию ... 553
Алфавитный указатель ... 565

Содержание

Предисловие ... 18
Что нового во втором издании ... 18
Что попало во второй том ... 19
Что должен знать читатель ... 19
Изучение C++ ... 19
Цели ... 20
Главы ... 21
Упражнения ... 24
Исходные тексты ... 25
Языковые стандарты ... 25
Поддержка ... 25
Благодарности ... 26
От издателя перевода ... 27

Глава 1. Знакомство с объектами ... 28
Развитие абстрактных представлений ... 29
Интерфейс объекта ... 30
Скрытая реализация ... 32
Повторное использование реализации ... 33
Наследование и повторное использование интерфейса ... 34
Точное и приблизительное подобие ... 37
Взаимозаменяемость объектов и полиморфизм ... 38
Создание и уничтожение объектов ... 42
Обработка исключений ... 43
Анализ и проектирование ... 44
Фаза 0 — составление плана ... 46
Фаза 1 — что делать ... 47

Фаза 2 — как делать ... 49
Фаза 3 — построение ядра ... 53
Фаза 4 — итеративный перебор сценариев ... 53
Фаза 5 — эволюция ... 54
О пользе планирования ... 55
Экстремальное программирование ... 56
Начинайте с написания тестов ... 56
Парное программирование ... 57
Причины успеха C++ ... 58
Улучшенный язык C ... 59
Быстрое обучение ... 59
Эффективность ... 60
Простота описания и понимания системы ... 60
Максимальная интеграция с библиотеками ... 60
Шаблоны и повторное использование исходных текстов ... 60
Обработка ошибок ... 61
Масштабное программирование ... 61
Стратегии перехода ... 61
Рекомендации ... 62
Организационные трудности ... 63
Итоги ... 65
Глава 2. Создание и использование объектов ... 67
Процесс трансляции ... 68
Интерпретаторы ... 68
Компиляторы ... 68
Процесс компиляции ... 69
Средства раздельной компиляции ... 71
Объявления и определения ... 71
Компоновка ... 75
Использование библиотек ... 76
Первая программа на C++ ... 77
Использование классов библиотеки Iostream ... 78
Пространства имен ... 78
Базовые сведения о строении программы ... 79
Программа «Hello, world!» ... 80
Запуск компилятора ... 81
О потоках ввода-вывода ... 81
Конкатенация символьных массивов ... 82
Чтение входных данных ... 82
Запуск других программ ... 83

Знакомство со строками ... 83
Чтение и запись файлов ... 84
Знакомство с векторами ... 86
Итоги ... 90
Упражнения ... 91

Глава 3. Элементы С в языке С++ ... 92
Создание функций ... 92
Возвращаемое значение функции ... 94
Использование библиотеки функций С ... 95
Разработка собственных библиотек ... 95
Управляющие конструкции ... 96
Значения true и false ... 96
Цикл if-else ... 96
Цикл while ... 97
Цикл do-while ... 98
Цикл for ... 98
Ключевые слова break и continue ... 99
Команда switch ... 100
Ключевое слово goto ... 101
Рекурсия ... 102
Знакомство с операторами ... 103
Приоритет ... 103
Инкремент и декремент ... 103
Знакомство с типами данных ... 104
Встроенные типы ... 104
Ключевые слова bool, true и false ... 105
Спецификаторы ... 106
Знакомство с указателями ... 107
Модификация внешних объектов ... 110
Знакомство со ссылками С++ ... 112
Указатели и ссылки как модификаторы ... 113
Видимость ... 114
Определение переменных непосредственно перед использованием ... 115
Распределение памяти ... 117
Глобальные переменные ... 117
Локальные переменные ... 118
Статические переменные ... 119
Внешние переменные ... 120
Константы ... 121
Квалификатор volatile ... 123

Операторы и их использование ... 123
Присваивание ... 123
Математические операторы ... 124
Операторы отношения ... 125
Логические операторы ... 125
Поразрядные операторы ... 126
Операторы сдвига ... 126
Унарные операторы ... 129
Тернарный оператор ... 129
Оператор запятой ... 130
Характерные ошибки при использовании операторов ... 130
Операторы приведения типов ... 131
Явное приведение типов в C++ ... 132
Оператор sizeof ... 135
Ключевое слово asm ... 136
Синонимы операторов ... 136
Создание составных типов ... 136
Определение псевдонимов ... 137
Объединение переменных в структуры ... 137
Перечисляемые типы ... 140
Экономия памяти при использовании объединений ... 141
Массивы ... 142
Рекомендации по отладке ... 150
Флаги отладки ... 150
Преобразование переменных и выражений в строки ... 152
Макрос assert() языка C ... 153
Адреса функций ... 153
Определение указателя на функцию ... 154
Сложные определения и объявления ... 154
Использование указателя на функцию ... 155
Массивы указателей на функции ... 156
Утилита make и управление раздельной компиляцией ... 156
Команда make ... 157
Make-файлы данной книги ... 160
Пример make-файла ... 160
Итоги ... 162
Упражнения ... 162
Глава 4. Абстрактное представление данных ... 167
Маленькая библиотека в стиле C ... 168
Динамическое распределение памяти ... 171
Неверные предположения ... 174

Конфликты имен ... 175
Базовый объект ... 176
Понятие объекта ... 181
Абстрактные типы данных ... 182
Подробнее об объектах ... 183
Этикет использования заголовочных файлов ... 184
О важности заголовочных файлов ... 184
Проблема многократного объявления ... 186
Директивы препроцессора #define, #ifdef и #endif ... 187
Стандартное устройство заголовочного файла ... 187
Пространства имен в заголовках ... 188
Использование заголовков в проектах ... 188
Вложенные структуры ... 189
Глобальная видимость ... 192
Итоги ... 192
Упражнения ... 192

Глава 5. Скрытие реализации ... 196
Установка ограничений ... 196
Управление доступом в C++ ... 197
Защищенные члены ... 198
Друзья ... 199
Вложенные друзья ... 201
Покушение на «чистоту» языка ... 203
Строение объекта ... 203
Класс ... 204
Версия Stash с ограничением доступа ... 206
Версия Stack с ограничением доступа ... 206
Классы-манипуляторы ... 207
Скрытие реализации ... 207
Лишние перекомпиляции ... 208
Итоги ... 210
Упражнения ... 210

Глава 6. Инициализация и зачистка ... 213
Гарантированная инициализация с помощью конструктора ... 214
Гарантированная зачистка с помощью деструктора ... 215
Исключение блока определений ... 217
Циклы for ... 218
Выделение памяти ... 219
Класс Stash с конструктором и деструктором ... 220
Класс Stack с конструктором и деструктором ... 223
Агрегатная инициализация ... 225

Конструктор по умолчанию 227
Итоги 228
Упражнения 229

Глава 7. Перегрузка функций и аргументы по умолчанию 231
Снова об украшении имен 232
Перегрузка по типу возвращаемого значения 233
Безопасность типов при компоновке 233
Пример перегрузки 234
Объединения 237
Аргументы по умолчанию 239
Заполнители в списке аргументов 241
Выбор между перегрузкой и аргументами по умолчанию 241
Итоги 245
Упражнения 246

Глава 8. Константы 248
Подстановка значений 248
Константы в заголовочных файлах 249
Константы и защита данных 250
Агрегаты 251
Отличия от языка С 251
Указатели 253
Указатель на константу 253
Константный указатель 254
Формат определений 254
Присваивание и проверка типов 255
Константность символьных массивов 255
Аргументы функций и возвращаемые значения 256
Передача констант по значению 256
Константное возвращаемое значение 257
Передача и возвращение адресов 259
Классы 261
Ключевое слово const в классах 262
Константы времени компиляции в классах 264
Константные объекты и функции классов 267
Ключевое слово volatile 271
Итоги 272
Упражнения 273

Глава 9. Подставляемые функции 276
Недостатки препроцессорных макросов 276
Макросы и доступ 279

Подставляемые функции ... 279
Подставляемые функции внутри классов ... 280
Функции доступа ... 281
Функции чтения и записи ... 282
Классы Stash и Stack с подставляемыми функциями ... 286
Подставляемые функции и компилятор ... 289
Ограничения ... 289
Опережающие ссылки ... 290
Скрытые операции в конструкторах и деструкторах ... 291
Вынесение определений из класса ... 292
Другие возможности препроцессора ... 293
Вставка лексем ... 293
Средства диагностики ... 294
Итоги ... 296
Упражнения ... 297

Глава 10. Механизм контроля имен ... 299
Статические элементы в языке C ... 299
Статические переменные в функциях ... 300
Статические объекты в функциях ... 301
Управление связыванием ... 303
Другие определители класса хранения ... 305
Пространства имен ... 305
Создание пространств имен ... 306
Использование пространств имен ... 308
Управление пространствами имен ... 311
Статические члены в C++ ... 312
Определение статических переменных классов ... 312
Вложенные и локальные классы ... 315
Статические функции классов ... 316
Порядок инициализации статических объектов ... 318
Первое решение ... 320
Второе решение ... 322
Альтернативные спецификации компоновки ... 325
Итоги ... 326
Упражнения ... 326

Глава 11. Ссылки и копирующий конструктор ... 331
Указатели в C++ ... 331
Ссылки в C++ ... 332
Ссылки в функциях ... 333
Рекомендации по передаче аргументов ... 335

Копирующий конструктор 335
Передача и возврат по значению 335
Конструирование копий 340
Копирующий конструктор по умолчанию 344
Альтернативы 346
Указатели на члены классов 347
Функции классов 349
Пример 350
Итоги 351
Упражнения 352

Глава 12. Перегрузка операторов 356
Предупреждение 356
Синтаксис 357
Перегружаемые операторы 358
Унарные операторы 358
Бинарные операторы 362
Аргументы и возвращаемые значения 371
Особые операторы 373
Неперегружаемые операторы 379
Операторы, не являющиеся членами классов 380
Базовые рекомендации 381
Перегрузка присваивания 382
Поведение функции operator= 383
Указатели в классах 384
Подсчет ссылок 386
Автоматическое создание функции operator= 390
Автоматическое приведение типа 391
Приведение типа с использованием конструктора 391
Оператор приведения типа 392
Пример приведения типа 394
Ошибки при автоматическом приведении типа 396
Итоги 397
Упражнения 398

Глава 13. Динамическое создание объектов 401
Создание объекта 402
Динамическое выделение памяти в языке C 403
Оператор new 404
Оператор delete 405
Простой пример 405
Затраты на управление памятью 406

Переработка ранних примеров ... 406
Вызов delete void* как вероятный источник ошибки ... 407
Зачистка при наличии указателей ... 408
Класс Stash с указателями ... 408
Операторы new и delete для массивов ... 412
Указатели и массивы ... 413
Нехватка памяти ... 413
Перегрузка операторов new и delete ... 414
Перегрузка глобальных операторов ... 415
Перегрузка операторов для класса ... 417
Перегрузка операторов для массивов ... 419
Вызовы конструктора ... 421
Операторы new и delete с дополнительными аргументами ... 422
Итоги ... 424
Упражнения ... 424

Глава 14. Наследование и композиция ... 427

Синтаксис композиции ... 427
Синтаксис наследования ... 429
Список инициализирующих значений конструктора ... 430
Инициализация объектов внутри класса ... 431
Встроенные типы в списке инициализирующих значений ... 431
Объединение композиции с наследованием ... 432
Автоматический вызов деструктора ... 433
Порядок вызова конструкторов и деструкторов ... 433
Скрытие имен ... 435
Функции, которые не наследуются автоматически ... 438
Наследование и статические функции классов ... 441
Выбор между композицией и наследованием ... 442
Выделение подтипов ... 443
Закрытое наследование ... 445
Защищенность ... 446
Защищенное наследование ... 447
Перегрузка операторов и наследование ... 447
Множественное наследование ... 448
Пошаговая разработка ... 449
Повышающее приведение типа ... 449
Понятие повышающего приведения типа ... 450
Повышающее приведение типа и копирующий конструктор ... 451
Снова о композиции и наследовании ... 453
Повышающее приведение типа указателей и ссылок ... 454
Проблемы ... 455

Итоги 455
Упражнения 455

Глава 15. Полиморфизм и виртуальные функции 459
Эволюция программирования на C++ 459
Повышающее приведение типа 460
Проблема 461
Связывание вызовов функций 461
Виртуальные функции 462
Расширяемость 463
Позднее связывание в C++ 465
Хранение информации о типе 466
Механизм вызова виртуальных функций 467
Внутренние механизмы 469
Инициализация указателя VPTR 470
Повышающее приведение типа и объекты 470
Виртуальные функции и эффективность 471
Абстрактные базовые классы и чисто виртуальные функции 472
Определения чисто виртуальных функций 476
Наследование и таблицы виртуальных функций 477
Расщепление объектов 479
Перегрузка и переопределение 481
Изменение типа возвращаемого значения 482
Виртуальные функции и конструкторы 484
Порядок вызова конструкторов 485
Поведение виртуальных функций внутри конструкторов 485
Деструкторы и виртуальные деструкторы 486
Чисто виртуальные деструкторы 488
Виртуальные функции в деструкторах 490
Создание однокоренной иерархии 490
Перегрузка операторов 493
Понижающее приведение типа 495
Итоги 498
Упражнения 499

Глава 16. Знакомство с шаблонами 503
Контейнеры 503
Потребность в контейнерах 505
Общие сведения о шаблонах 506
Многократное использование кода в C и Smalltalk 506
Решение с применением шаблонов 508

Синтаксис шаблонов ... 509
Определения неподставляемых функций ... 510
Класс IntStack в виде шаблона ... 511
Константы в шаблонах ... 512
Шаблоны для классов Stack и Stash ... 514
Шаблон класса PStash для указателей ... 516
Управление принадлежностью ... 520
Хранение объектов по значению ... 522
Знакомство с итераторами ... 524
Класс Stack с итераторами ... 531
Класс PStash с итераторами ... 533
Ценность итераторов ... 538
Шаблоны функций ... 540
Итоги ... 541
Упражнения ... 542

Приложение А. Стиль программирования ... 545

Общие правила ... 545
Имена файлов ... 546
Начальные и конечные комментарии ... 546
Отступы, круглые и фигурные скобки ... 547
Имена идентификаторов ... 550
Порядок включения заголовочных файлов ... 551
Защита заголовков ... 551
Использование пространств имен ... 551
Функции require() и assure() ... 552

Приложение Б. Рекомендации по программированию ... 553

Алфавитный указатель ... 565

Моим родителям, сестре и брату

Предисловие

Язык С++, как и любой из человеческих языков, предназначен для выражения смысловых концепций. Если правильно использовать это выразительное средство, оно оказывается гораздо более удобным и гибким, чем его альтернативы (особенно с увеличением объема и сложности задач).

С++ нельзя рассматривать как механический набор функциональных возможностей, многие из которых сами по себе не имеют смысла. Если речь идет о *проектировании*, а не о простом кодировании, необходимо использовать всю совокупность этих возможностей. Но чтобы понять С++ с этой точки зрения, нужно представлять себе основные проблемы С и программирования в целом. В этой книге мы рассмотрим проблемы программирования, выясним, почему они являются проблемами и как С++ подходит к их решению. Таким образом, материал каждой главы выбирается на основании моего представления о решении определенного круга проблем с помощью языка. Надеюсь, при таком подходе мне удастся постепенно провести вас от знания С к той точке, где «родным» для вас станет С++.

В этой книге я постараюсь сформировать у читателя такую мысленную модель, которая бы давала глубокое понимание языка вплоть до самых основ. Встретив непонятную ситуацию, вы используете эту модель и получаете ответ. Я попытаюсь поделиться с читателем теми соображениями, которые изменили мой стиль мышления и помогли мне «думать на С++».

Что нового во втором издании

По сравнению с первым изданием книга была основательно переработана. В ней были отражены все изменения, появившиеся с выходом окончательной версии стандарта С++, а также то, что я узнал с момента выхода первого издания. Весь текст первого издания был проанализирован и переписан. Из книги постепенно исключались некоторые старые примеры, изменялись существующие, добавлялись новые. Также появилось много новых упражнений. Материал был значительно переупорядочен, что объясняется появлением новых вспомогательных инструмен-

тов и расширением моего понимания того, как люди изучают C++. Была добавлена новая глава, которая кратко представляет основные концепции C и базовый синтаксис C читателям, не обладающим опытом работы на C.

Итак, на вопрос: «что же нового во втором издании?» — можно ответить так: одна часть материала была написана заново, а другая существенно изменилась (иногда до такой степени, что вы не узнаете исходные примеры и материал).

Что попало во второй том

С завершением работы над стандартом C++ в стандартную библиотеку C++ был добавлен целый ряд важных новых библиотек, в том числе класс string, контейнеры и алгоритмы, а также поддержка шаблонов. Их описания были перенесены во второй том[1] вместе с такими нетривиальными темами, как множественное наследование, обработка исключений, паттерны проектирования, построение и отладка стабильных систем.

Что должен знать читатель

Первое издание книги писалось из расчета, что читатель знает C или, по крайней мере, способен прочитать программу на C. При этом я старался упростить то, что считал более сложным: язык C++. В этом издании появилась глава с кратким введением в C, но по-прежнему предполагается, что читатель обладает некоторым опытом программирования. Подобно тому, как мы легко запоминаем новые слова, встречая их в романах, о языке C можно многое узнать из контекста его использования в книге.

Изучение C++

Я взялся за изучение C++ точно в таком же положении, в котором, вероятно, находятся многие читатели, крайне серьезно относясь к своей профессии и стремясь во всем докопаться до самой сути. К сожалению, мой личный опыт относился к области программирования систем на аппаратном уровне, где C часто считался языком высокого уровня и недостаточно эффективным инструментом для низкоуровневых операций с битами. Позднее выяснилось, что я даже не был хорошим программистом C и просто скрывал свое невежество в области структур, функций malloc() и free(), setjmp() и longjmp(), а также других «сложных» концепций. А когда эти темы всплывали в разговоре, я малодушно уходил в сторону, вместо того чтобы овладевать новыми знаниями.

Когда я начал осваивать C++, существовала только одна книга — знаменитый учебник Бьярна Страуструпа[2], который характеризовался как «руководство для экспертов». Таким образом, разбираться в основных концепциях C++ мне приходилось самостоятельно. В результате была написана моя первая книга по C++[3], которая фактически стала прямым отражением моего личного опыта.

---

[1] Выход второго тома запланирован на 2004 год. — *Примеч. изд.*

[2] «The C++ Programming Language», Bjarne Stroustrup, Addison-Wesley, 1986 (первое издание).

[3] «Using C++», Osborne/McGraw-Hill, 1989.

В ней я попытался познакомить читателя одновременно с С и С++. Оба издания книги были тепло встречены читателями.

Почти одновременно с публикацией книги «Using С++» я начал преподавать С++ на семинарах и презентациях. Обучение С++ (а позднее и Java) стало моей профессией; с 1989 года я вижу кивающие головы, непонимающие и озадаченные лица в аудиториях по всему миру. Потом я начал проводить закрытые лекции для небольших групп и обнаружил нечто любопытное. Даже те люди, которые улыбались и кивали, слабо разбирались во многих вопросах. Я создал и в течение многих лет возглавлял секции С++ и Java на конференции разработчиков программного обеспечения и за это время убедился, что я и другие докладчики преподносят материал в слишком высоком темпе. В результате из-за различий как в уровне подготовки аудитории, так и в способах подачи часть материала не воспринимается. Возможно, я хочу слишком многого, но так как я принадлежу к числу людей, плохо воспринимающих традиционные лекции (которые, как мне кажется, обычно просто скучны), мне хотелось бы, чтобы все слушатели успевали за мной.

Однажды мне довелось подготовить несколько разных презентаций в течение короткого промежутка времени. Так я пришел к использованию экспериментального и итеративного метода (кстати, этот способ также хорошо работает при проектировании программ С++). Со временем я разработал учебный курс, который включал в себя все, что я узнал в ходе своей преподавательской деятельности. В нем процесс обучения делился на отдельные легко усваиваемые шаги, а для самостоятельной работы (наиболее эффективный способ обучения) каждая презентация сопровождалась упражнениями.

Первое издание этой книги создавалось в течение двух лет, а представленный в нем материал был опробован во многих формах на многих семинарах. На основании отзывов, полученных на семинарах, я изменял способ подачи материала, пока не почувствовал, что он подходит для образовательных целей. Но не стоит полагать, что книга содержит краткое изложение программы семинаров; я постарался включить в нее как можно больше информации и структурировать ее для перехода к следующей теме. Книга в первую очередь призвана служить читателю, который в муках осваивает новый язык программирования.

Цели

Работая над книгой, я поставил себе следующие цели.

- Поэтапное изложение материала, чтобы читатель прочно усваивал одну концепцию, а потом переходил к другим.
- Как можно более короткие и простые примеры. Это часто не позволяло мне использовать в книге «реальные» задачи, но я убедился, что для начинающих доступность примера обычно важнее, чем грандиозность решаемой проблемы. Кроме того, объем программного кода, воспринимаемого в процессе обучения, ограничен. Меня иногда критикуют за «игрушечность» примеров, но я готов смириться с этим — важнее, чтобы они были поучительными.
- Тщательно продуманная последовательность изложения, чтобы читатель не сталкивался с незнакомыми темами. Конечно, это удавалось не всегда; в таких ситуациях дается краткое вводное описание.

- Книга должна содержать материал, важный для понимания языка (а не все, что я знаю). На мой взгляд, информация не равноценна по своей важности. Некоторые темы 95 % программистов никогда не понадобятся, лишь запутают их и повысят субъективную сложность языка. Для примера обратимся к C: если выучить таблицу приоритета операторов (чего лично я никогда не делал), вы сможете создавать более короткие выражения. Но если *вам* приходится задумываться над написанием выражения, то оно наверняка запутает читателя программы. Так что забудьте о приоритетах, а в неоднозначных ситуациях используйте круглые скобки. То же относится к некоторым аспектам языка C++, которые, как мне кажется, больше интересуют разработчиков компиляторов, а не программистов.
- Материал каждого раздела должен быть посвящен конкретной теме. Это не только повышает активность и заинтересованность читателей, но и создает у них ощущение продвижения к цели.
- Книга должна заложить прочную основу, чтобы читатель хорошо понял изложенный материал и мог перейти к более сложным курсам и книгам (в частности, ко второму тому этой книги).
- Обучение не должно ориентироваться на определенную версию C++, потому что для нас важны не подробности конкретной реализации, а сам язык. Как правило, производители предоставляют вполне приличную документацию с описанием особенностей реализации своего компилятора.

Главы

В языке C++ новые возможности строились на базе существующего синтаксиса (поэтому C++ часто называют *гибридным* объектно-ориентированным языком). По мере того как все больше людей успешно проходят обучение, мы начинаем лучше представлять, в какой последовательности программисты осваивают разные аспекты C++. Как выяснилось, они следуют по более или менее одинаковому пути, естественному для программиста с подготовкой в области процедурного программирования. Поэтому я попытался понять этот путь, пройти его и ускорить его, ответив на те вопросы, которые возникали у меня при изучении языка, а также те, которые мне задавались в различных аудиториях во время преподавания.

Этот курс готовился с одной целью: рационализировать процесс изучения C++. Отклик аудитории помог мне разобраться в том, какие части трудны для понимания и требуют дополнительных пояснений. Иногда меня охватывало чрезмерное рвение, и я пытался представить слишком много материала сразу. Однако я на собственном опыте выяснил, что все новые возможности необходимо хорошо объяснить, а это часто приводит учеников в замешательство. В итоге после долгой внутренней борьбы я постарался свести к минимуму количество представляемых концепций языка, в идеале — до одной серьезной концепции на каждую главу.

Таким образом, в каждой главе я стремился представить одну концепцию (или небольшую группу взаимосвязанных концепций), по возможности без упоминания того, что еще не рассматривалось. Для этого мне пришлось вспоминать о механизмах C несколько дольше, чем хотелось бы. Преимущество такого подхода —

в том, что читатель не будет путаться при виде незнакомых конструкций C++, которые еще не рассматривались в книге. Знакомство с языком пройдет спокойно и примерно в той же последовательности, в которой вы бы сами изучали его самостоятельно.

Ниже приводится краткое описание глав.

Глава 1. Знакомство с объектами. Когда проекты стали слишком большими и сложными для нормального сопровождения, наступил «кризис программирования»; программисты говорили: «Мы не можем довести свои проекты до конца, а если и доведем — они обойдутся слишком дорого!» Для решения возникших проблем появились новые решения, которые рассматриваются в этой главе вместе с основными идеями объектно-ориентированного программирования (ООП) и его местом в преодолении кризиса. Кроме того, в этой главе рассматриваются достоинства и недостатки языка, а также даются рекомендации по поводу перехода на C++.

Глава 2. Создание и использование объектов. Глава посвящена процессу разработки программ с использованием компиляторов и библиотек. Мы рассмотрим первую программу на C++ и выясним, как строятся и компилируются программы. Далее будут представлены некоторые базовые библиотеки объектов, входящие в стандарт C++. К концу этой главы вы будете хорошо представлять себе, как написать программу на C++ с помощью готовой библиотеки объектов.

Глава 3. Элементы C в языке C++. В этой главе приводится краткий обзор средств языка C, используемых в C++, а также некоторых базовых возможностей, присутствующих только в C++. Также в ней представлена утилита make, часто используемая при программировании; в частности, именно она применялась для подготовки всех примеров книги. Материал главы построен в предположении, что у вас имеется солидный опыт программирования на каком-либо процедурном языке, например Pascal, C или даже на каком-нибудь из диалектов BASIC.

Глава 4. Абстрактное представление данных. Многие средства C++ так или иначе связаны с возможностью создания новых типов данных. Абстракция не только улучшает структуру программ, но и закладывает основу для других, более мощных средств ООП. В главе показано, как эта идея реализуется простым перемещением функций внутрь структур, подробно описывается, как это происходит и какой код при этом создается. Также будет представлена оптимальная организация программ с делением на заголовочные файлы и файлы реализации.

Глава 5. Скрытие реализации. Вы можете решить, что некоторые данные и функции структур должны быть недоступны пользователям нового типа, и объявить их закрытыми (private). Такой подход отделяет внутреннюю реализацию от интерфейса, используемого прикладным программистом, и позволяет легко изменить реализацию, не нарушая работы клиентского кода. Читатель узнает, что ключевое слово class всего лишь описывает новый тип данных, и поймет, что загадочный термин «объект» просто обозначает особую переменную.

Глава 6. Инициализация и зачистка. Многие ошибки в языке C возникали из-за использования неинициализированных переменных. *Конструкторы* C++ гарантируют, что переменные новых типов данных («объекты классов») всегда пройдут инициализацию. Если для объектов требуются некоторые завершающие действия (зачистка), вы всегда можете рассчитывать на ее выполнение *деструктором* C++.

Глава 7. Перегрузка функций и аргументы по умолчанию. Язык C++ помогает создавать большие сложные проекты. Вы можете задействовать множество биб-

лиотек, содержащих одноименные функции, и даже использовать одно и то же имя с разной смысловой нагрузкой в одной библиотеке. В C++ эта задача легко решается благодаря *перегрузке функций*, позволяющей употреблять одно имя для функций с разными списками аргументов. С другой стороны, аргументы по умолчанию позволяют по-разному вызывать одну и ту же функцию с автоматической передачей значений по умолчанию для некоторых аргументов.

Глава 8. Константы. Глава посвящена ключевым словам const и volatile, которые имеют особый смысл в C++ (особенно в классах). Вы узнаете, как ключевое слово const применяется к определениям указателей. Также в этой главе будет показано, как изменяется смысл ключевого слова const при использовании его вне и внутри классов и как определять в классах константы времени компиляции.

Глава 9. Подставляемые функции. Препроцессорные макросы избавляют программы от издержек, связанных с вызовами функций, но при этом препроцессор подавляет работу полезного механизма проверки типов. Подставляемые функции обладают всеми достоинствами препроцессорных макросов, сохраняя все преимущества «настоящих» вызовов функций. В этой главе подробно рассматривается реализация и использование подставляемых функций.

Глава 10. Механизм контроля имен. Именование является одной из важнейших операций в программировании. В очень больших проектах нередко возникает нехватка свободных имен. C++ обеспечивает высокую степень контроля над именами: их созданием, использованием, размещением в памяти и компоновкой. В этой главе продемонстрированы принципы двух способов контроля имен в C++. В первом варианте ключевое слово static управляет видимостью и типом связывания имен (также рассматривается его особый смысл в классах). Гораздо более удобный механизм контроля имен на глобальном уровне основан на использовании ключевого слова namespace, позволяющего разделять глобальное пространство имен на изолированные части.

Глава 11. Ссылки и копирующий конструктор. Указатели C++ похожи на указатели C, но при этом они обладают дополнительными преимуществами благодаря сильной типизации C++. В C++ также существует другой механизм для работы с адресами: из Algol и Pascal позаимствован механизм *ссылок*, позволяющий компилятору передавать адрес вместо значения при использовании обычной записи. Также в этой главе представлены копирующие конструкторы, которые применяются при передаче и возврате объектов функциями по значению. Глава завершается описанием указателей на члены классов.

Глава 12. Перегрузка операторов. Возможность перегрузки операторов иногда относится к категории «синтаксического сахара» — она упрощает синтаксис пользовательских типов, позволяя задействовать стандартные операторы наряду с функциями. Как показано в этой главе, перегрузка операторов всего лишь является другой формой вызова функции. Вы научитесь писать собственные операторы, ориентироваться в неочевидных аспектах использования аргументов и возвращаемых типов, а также выбирать между оформлением оператора в виде функции класса или в виде дружественной функции.

Глава 13. Динамическое создание объектов. Сколькими самолетами придется управлять системе управления воздушным движением? Сколько геометрических фигур будет задействовано в работе системы автоматизированного проектирования? В общем случае нам неизвестно количество, продолжительность жизненного

цикла и/или тип объектов, необходимых для вашей программы. В этой главе будет показано, как элегантно эта проблема решается в C++ при помощи новых операторов new и delete. Также продемонстрированы разнообразные варианты перегрузки операторов new и delete, позволяющие управлять выделением и освобождением памяти.

Глава 14. Наследование и композиция. Абстрактное представление данных позволяет создавать новые типы «с нуля», но путем композиции и наследования новые типы можно создавать на базе существующих типов. В случае композиции новый тип собирается из компонентов, которые представляют собой объекты других типов, тогда как при наследовании создается специализированная версия существующего типа. В этой главе рассматривается синтаксис переопределения функций, а также подчеркивается важная роль конструкторов и деструкторов при наследовании и композиции.

Глава 15. Полиморфизм и виртуальные функции. Чтобы самостоятельно изучить и хорошо понять этот краеугольный камень ООП, вам понадобится полгода, если не больше. На основании небольших простых примеров мы разберемся, как создать семейство типов путем наследования и как работать с объектами этого семейства через общий базовый класс. Ключевое слово virtual позволяет работать со всеми классами семейства на общем уровне вне зависимости от фактического типа объекта. В результате улучшается расширяемость программ и, соответственно, упрощается и удешевляется их разработка и сопровождение.

Глава 16. Знакомство с шаблонами. Наследование и композиция позволяют заново задействовать объектный код, однако это не решает всех проблем многократного использования. Шаблоны дают возможность многократно использовать *исходные тексты* программ и автоматизировать подстановку имен типов в тело класса или функции. Эта возможность обеспечивается библиотеками *контейнерных классов* — важным инструментом быстрой и надежной разработки объектно-ориентированных программ (стандартная библиотека C++ содержит обширную подборку контейнерных классов). Последняя глава закладывает прочную основу для изучения этой важнейшей темы.

Другие темы (а также более сложные аспекты C++) рассматриваются во втором томе книги.

Упражнения

Я выяснил, что самостоятельная работа исключительно важна для полноценного понимания темы, поэтому каждая глава завершается набором упражнений. По сравнению с первым изданием количество упражнений заметно увеличилось.

Многие упражнения относительно просты, чтобы их можно было выполнить за разумный промежуток времени в классе или лаборатории под наблюдением инструктора и убедиться в том, что все учащиеся усвоили материал. Некоторые упражнения имеют повышенную сложность и ориентированы на более опытных учащихся. И все же в большинстве случаев упражнения решаются быстро и предназначаются для проверки полученных знаний. Предполагается, что более сложные задачи вы найдете самостоятельно или, что более вероятно, они сами найдут вас.

Исходные тексты

Исходные тексты программ, приводимых в книге, распространяются бесплатно, но с соблюдением авторских прав. В соответствии с авторскими правами запрещается воспроизведение кода в печатных изданиях без разрешения, но разрешается его использование во многих других ситуациях.

Код распространяется в виде архива, который может быть распакован на любой платформе утилитой zip (то есть на большинстве платформ; если на вашем компьютере эта утилита не установлена, поищите версию для своей платформы в Интернете). В каталоге, выбранном для распаковки архива, находится файл с текстом уведомления об авторских правах. Вы можете использовать эти программы в своих проектах и при проведении учебных занятий при условии соблюдения перечисленных условий.

Языковые стандарты

Говоря о соответствии стандарту ISO C, я буду просто говорить «C». Различия между стандартом C и старыми версиями, предшествовавшими появлению стандарта, будут проводиться только там, где это необходимо.

На момент написания книги комитет по стандартизации C++ завершил работу над языком. Используя термин «стандарт C++», я имею в виду стандартизированный язык. Если я просто упоминаю о C++, это тоже означает соответствие стандарту.

По поводу названий комитета по стандартизации C++ и самого стандарта существует некоторая путаница. Стив Кламадж (Steve Clamage), председатель комитета, разъясняет ситуацию так.

Стандартизацией C++ занимаются два комитета ++: комитет NCITS (ранее X3) J16 и комитет ISO JTC1/SC22/WG14. ANSI предоставляет NCITS право формировать технические комитеты для разработки американских национальных стандартов.

В 1989 году комитету J16 было поручено создать американский стандарт C++. Примерно в 1991 году комитету WG14 было поручено создать международный стандарт. Проект J16 был преобразован в международный проект и передан в ведение стандартизационных проектов ISO.

Два комитета проводят заседания в одно время и в одном месте, и право голоса J16 трактуется WG14 как голос американской стороны. WG14 поручает J16 техническую работу и голосует по поводу ее результатов.

Стандарт C++ первоначально создавался как стандарт ISO. Позднее институт ANSI проголосовал (по рекомендации J16) за принятие стандарта ISO C++ в качестве американского стандарта C++.

Таким образом, при ссылках на стандарт C++ правильнее говорить о стандарте ISO.

Поддержка

Возможно, ваш компилятор не поддерживает некоторые из возможностей, рассматриваемых в книге (особенно в случае старой версии компилятора). Реализация такого языка, как C++, — титанический труд, поэтому последовательное введение

поддержки тех или иных возможностей оправданно. Если при подготовке одного из примеров компилятор выдаст многочисленные сообщения об ошибках, это не обязательно свидетельствует об ошибке компилятора; возможно, соответствующие возможности просто не реализованы в вашем конкретном компиляторе.

Благодарности

Прежде всего, я благодарен всем пользователям Интернета, которые поделились со мной своими замечаниями и рекомендациями. Они оказали огромную помощь при доработке книги, и без них у меня бы ничего не вышло. Отдельное спасибо Джону Куку (John Cook).

Идеи и свое понимание языка для этой книги я черпал из многих источников. Прежде всего, это мои друзья Чак Эллисон (Chuck Allison), Андреа Провальо (Andrea Provaglio), Дэн Сакс (Dan Saks), Скотт Мейерс (Scott Meyers), Чарльз Петцольд (Charlez Petzold) и Майкл Уилк (Michael Wilk); такие первопроходцы языка, как Бьярн Страуструп (Bjarne Stroustrup), Эндрю Кениг (Andrew Koenig) и Роб Мюррей (Rob Murray); члены комитета по стандартизации C++ Натан Майерс (Nathan Myers), особенно щедрый на полезные советы, Билл Плаугер (Bill Plauger), Рег Чарни (Reg Charney), Том Пенелло (Tom Penello), Том Плам (Tom Plum), Сэм Друкер (Sam Druker) и Уве Стайнмюллер (Uwe Steinmueller); люди, с которыми я беседовал в секции C++ на конференции разработчиков программного обеспечения, и другие посетители моих семинаров, которые задавали мне вопросы, столь необходимые, чтобы материал стал еще более понятным.

Огромное спасибо моему другу Гэну Киёка (Gen Kiyooka), чья компания Digigami предоставила в мое распоряжение веб-сервер.

Я изучал C++ вместе со своим другом Ричардом Хейлом Шоу (Richard Hale Shaw), замечания и поддержка которого оказали мне огромную помощь. Я также благодарен КоЭнн Викорен (KoAnn Vikoren), Эрику Фауроту (Eric Faurot), Дженнифер Джессап (Jennifer Jessup), Таре Эрровуд (Tara Arrowood), Марко Парди (Marco Pardi), Николь Фримен (Nicole Freeman), Барбаре Ханском (Barbara Hanscome), Регине Ридли (Regina Ridley), Алексу Данну (Alex Dunne) и всем остальным сотрудникам MFI.

Выражаю отдельную благодарность всем своим учителям и ученикам (которые тоже меня многому научили).

Спасибо всем моим любимым писателям, я искренне признателен им за их работу: Джон Ирвинг, Нил Стивенсон, Робертсон Дэвис (нам тебя будет не хватать), Том Роббинс, Уильям Гибсон, Ричард Бах, Карлос Кастанеда и Джин Вольф.

Спасибо Гвидо ван Россуму (Guido van Rossum) за то, что он изобрел Python и бескорыстно отдал его миру. Твой дар обогатил мою жизнь.

Я благодарен работникам Prentice Hall: Алану Эпту (Alan Apt), Ане Терри (Ana Terry), Скотту Дисанно (Scott Disanno), Тони Хольму (Tony Holm) и редактору электронного варианта Стефани Инглиш (Stephanie English), а также Брайану Гамбрелу (Brian Gambrel) и Дженни Бургер (Jennie Burger) из отдела маркетинга.

Сонда Донован (Sonda Donovan) помогала с подготовкой компакт-диска, а Дэниел Уилл-Харрис (Will-Harris) создал для него рисунок.

Спасибо всем замечательным людям из «Гребнистого Холма», превратившим его в такое замечательное место; спасибо Элу Смиту (Al Smith), создателю пре-

красного кафе «Camp4», моим соседям Дэйву и Эрике, Марше из книжного магазина «Heg's Place», Пат и Джону из «Teocalli Tamale», Сэму из «Bakers Cafe» и Тиллеру за помощь в исследованиях. Благодарю всех прекрасных людей, которые посещают Camp4 и делают мое утро более интересным.

Ниже перечислены лишь некоторые из моих друзей, оказывавших мне поддержку в работе: Зак Урлокер (Zack Urlocker), Эндрю Бинсток (Andrew Binstock), Нейл Рубенкинг (Neil Rubenking), Крейг Брокшмидт (Kraig Brockschmidt), Стив Синофски (Steve Sinofsky), JD Хильдебрандт (JD Hildebrandt), Брайан Мак-Элинни (Brian McElhinney), Бринкли Барр (Brinkley Barr), Ларри О'Брайен (Larry O'Brien), Билл Гейтс (Bill Gates) из «Midnight Engineering Magazine», Ларри Константайн (Larry Constantine), Люси Локвуд (Lucy Lockwood), Том Кеффер (Tom Keffer), Дэн Путтерман (Dan Putterman), Джин Ванг (Gene Wang), Дэйв Майер (Dave Mayer), Дэвид Интерсимон (David Intersimone), Клэр Сойерс (Claire Sawyers), итальянцы Андреа Провальо (Andrea Provaglio), Роселла Джиоя (Rossella Gioia), Лаура Фаллай (Laura Fallai), Марко и Лелла Канту (Marco & Lella Cantu), Коррадо (Corrado), Ильза и Кристина Джустоцци (Ilsa & Christina Giustozzi), Крис и Лора Стрэнд (Chris and Laura Strand) и Паркер (Parker), Элмквисты (Almquists), Брэд Джербик (Brad Jerbic), Мэрилин Цветанич (Marylin Cvitanic), семейства Мэбри (Mabrys), Хафлингер (Haflingers) и Поллок (Pollocks), Питер Винчи (Peter Vinci), семейства Роббинс (Robbins) и Мелтер (Moelters), Дэйв Стонер (Dave Stoner), Лори Адамс (Laurie Adams), семейство Крэнстон (Cranstons), Ларри Фогг (Larry Fogg), Майк и Карен Секвейра (Mike and Karen Sequiera), Гэри Энтсмингер (Gary Entsminger) и Эллисон Броди (Allison Brody), Кевин (Kevin), Сонда (Sonda) и Элла Донован (Ella Donovan), Честер (Chester) и Шеннон Андерсен (Shannon Andersen), Джо Лорди (Joe Lordi), Дэйв (Dave) и Бренда Бартлетт (Brenda Bartlett), семейство Рентшлер (Rentshlers), Линн (Lynn) и Тодд (Todd) и их семьи. И конечно, папа с мамой.

От издателя перевода

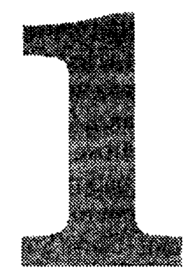
Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

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

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

Знакомство с объектами



Компьютерная революция начиналась с машин, поэтому многие языки программирования ориентированы на машины.

Однако компьютер — не столько машина, сколько инструмент, расширяющий возможности человеческого разума, — «велосипед для разума», как выражается Стив Джобс (Steve Jobs), и новая разновидность выразительных средств. В результате этот инструмент все меньше напоминает машины и все больше — новые стороны нашего разума и такие выразительные средства, как живопись, скульптура, мультипликация или кино. Объектно-ориентированное программирование стало одним из направлений на пути превращения компьютера в выразительное средство.

В этой главе читатель познакомится с основными концепциями объектно-ориентированного программирования (ООП), включая краткий обзор методов ООП-разработки. Настоящая глава (как и книга в целом) предполагает, что у читателя имеется опыт работы на процедурных языках программирования — не обязательно на языке C.

Приведенный материал носит общий, подготовительный характер. Некоторые читатели чувствуют себя неуверенно, если перед погружением в мир объектно-ориентированного программирования они не увидят «общую картину». Для таких читателей и был подготовлен содержательный обзор ООП. С другой стороны, многие читатели просто не воспринимают «общую картину», пока не узнают, как «это» работает на практике; они вязнут в общих описаниях, пока не увидят конкретный программный код. Если вы относитесь ко второй категории и желаете как можно скорее начать практическое знакомство с языком, спокойно пропустите эту главу — это не помешает изучать язык или писать собственные программы. Впрочем, когда-нибудь в будущем к этой главе было бы полезно вернуться. Так вы сможете понять, почему объекты настолько важны и как проектировать программы на объектной основе.

Развитие абстрактных представлений

Любой язык программирования основан на абстрактных представлениях. Более того, сложность решаемых задач напрямую зависит от типа и качества этой абстракции. Под «типом» подразумевается ответ на вопрос: «Что представляет данная абстракция?» Например, ассемблер можно рассматривать как маленькую абстракцию того компьютера, на котором он работает. Многие так называемые «массовые» языки, появившиеся позднее (например, Fortran, BASIC и C), могли рассматриваться как абстракции языка ассемблера. Они были гораздо совершеннее, но их первичная абстракция по-прежнему заставляла программиста мыслить в контексте структуры компьютера, а не структуры решаемой задачи. Программисту приходилось сопоставлять машинную модель («пространство решения», то есть контекст, в котором моделируется задача, — в данном случае компьютер) и модель решаемой задачи («пространство задачи», в котором определяется поставленная задача). Такое сопоставление требовало определенных усилий и было чужеродным по отношению к языку программирования. Это затрудняло разработку программ и повышало затраты на их сопровождение; в результате возникла целая отрасль «методологии программирования».

Наряду с моделированием компьютера также существует другое решение — моделирование решаемой задачи. В ранних языках типа LISP и APL было выбрано особое представление мира («Любая задача сводится к обработке списков» или «Любая задача алгоритмизируется»). В языке PROLOG задача интерпретировалась как цепочка принимаемых решений. Создавались специальные языки программирования, основанные на системах ограничений или на манипуляциях с графическими условными обозначениями (впрочем, последние слишком сильно ограничивали свободу действий программиста). Все эти подходы хорошо соответствовали конкретному классу задач, для которых они разрабатывались, но за пределами этой области были весьма неудобными.

Объектно-ориентированный подход не ограничивается узкой специализацией. Он дает программисту средства представления элементов в пространстве задачи. Представление носит достаточно обобщенный характер, чтобы программист не ограничивался конкретным типом задач. Элементы пространства задачи и их представления в пространстве решения называются *объектами* (конечно, наряду с ними существуют другие объекты, не имеющие аналогов в пространстве задачи). Идея заключается в том, чтобы программа могла адаптироваться к формулировке задачи за счет добавления новых типов объектов. Таким образом, читая программный код с описанием решения, вы также будете читать словесную формулировку задачи. По своей гибкости и мощи эта языковая абстракция превосходит все, что существовало до нее. Итак, ООП позволяет описать задачу в контексте задачи, а не в контексте того компьютера, на котором она будет решаться. Впрочем, определенная связь с компьютерами все же сохранилась. Каждый объект отдаленно напоминает маленький компьютер — он тоже обладает состоянием и поддерживает набор операций, которые вы можете выполнять. С другой стороны, нетрудно провести аналогию с объектами реального мира, которые также обладают характеристиками и определенным поведением.

Некоторые проектировщики языков посчитали, что самого по себе объектно-ориентированного программирования недостаточно для простого решения всех задач программирования, и выступили за объединение разных подходов в *мультипарадигменных* языках программирования.

Алан Кей (Alan Kay) кратко сформулировал пять основных концепций SmallTalk — первого успешного объектно-ориентированного языка, который стал одним из прототипов C++. Эти концепции представляют «чистый» подход к объектно-ориентированному программированию.

- *Любая сущность представляется объектом.* Объект можно рассматривать как усовершенствованную переменную: он содержит данные, но ему также можно направлять запросы на выполнение операций «с самим собой». Теоретически любой концептуальный компонент решаемой задачи (собака, здание, некоторая операция и т. д.) может быть представлен в программе в виде объекта.
- *Программа представляет собой совокупность объектов, которые общаются друг с другом посредством отправки сообщений.* Чтобы обратиться с запросом к объекту, следует отправить ему сообщение. Точнее, сообщение может рассматриваться как запрос на вызов функции, принадлежащей конкретному объекту.
- *Каждый объект владеет собственным блоком памяти, состоящим из других объектов.* Иначе говоря, новые объекты в программе создаются как «оболочки» для существующих объектов. Это позволяет усложнять программу, скрывая ее за простотой отдельного объекта.
- *Каждый объект обладает типом.* В терминологии ООП каждый объект является *экземпляром класса*, где термин «класс» является синонимом термина «тип». Важнейшая отличительная характеристика класса формулируется так: «Какие сообщения принимает данный класс?»
- *Все объекты одного типа могут получать одни и те же сообщения.* Как вы вскоре убедитесь, это утверждение не совсем корректно. Поскольку объект типа «круг» также является объектом типа «геометрическая фигура», объект-круг заведомо принимает сообщения, предназначенные для объектов-фигур. Следовательно, программа для работы с геометрическими фигурами может автоматически обрабатывать все объекты, которые являются частными случаями геометрической фигуры. Данная особенность является одной из самых сильных сторон ООП.

Интерфейс объекта

Вероятно, Аристотель был первым, кто серьезно изучал концепцию *типов*: он говорил о «классе птиц и классе рыб». Принцип, в соответствии с которым все объекты, несмотря на свою уникальность, также принадлежат к некоторому классу, обладающему и общими характеристиками и поведением, был напрямую воплощен в первом объектно-ориентированном языке Simula-67 — именно там появилось основополагающее ключевое слово class, которое вводило новый тип в программу.

Язык Simula, как нетрудно догадаться по его названию, создавался для имитации — например, для решения знаменитой «задачи кассира». В этой задаче задействовано множество кассиров, клиентов, счетов, операций и денежных единиц — короче говоря, множество «объектов». Объекты, различающиеся только по текущему состоянию во время выполнения программы, группируются в «классы»; отсюда и произошло ключевое слово class. Создание абстрактных типов данных (классов) является одной из основополагающих концепций объектно-ориентированного программирования. Абстрактные типы данных работают почти так же, как встроенные типы: программист определяет переменные этого типа (в терминологии ООП такие переменные называются *объектами*, или *экземплярами*) и работает с ними при помощи *сообщений*, или *запросов*. Программист отправляет запрос, а объект сам решает, как следует обработать этот запрос. Объекты, относящиеся к одному классу, обладают общими характеристиками: так, у каждого счета имеется баланс, каждый кассир может принимать деньги от клиентов и т. д. В то же время каждый объект обладает собственным состоянием: на счетах хранятся разные суммы, кассиры обладают разными именами и т. д. Таким образом, любой объект — кассир, клиент, счет, операция — представляется в программе отдельной сущностью (объектом), причем каждый объект принадлежит к конкретному классу, определяющему его характеристики и поведение.

Итак, хотя в объектно-ориентированном программировании фактически определяются новые типы данных, практически во всех объектно-ориентированных языках используется термин «класс». Если вы видите термин «тип», читайте «класс» — и наоборот[1].

Поскольку класс определяет совокупность объектов с одинаковыми характеристиками (элементы данных) и поведением (функциональность), встроенные типы данных тоже можно рассматривать как классы — скажем, вещественные числа тоже обладают набором характеристик и поведением. Различие состоит в том, что программист определяет класс в соответствии со спецификой задачи, а не использует готовый тип данных, представляющий машинную единицу хранения данных. Он расширяет язык программирования и включает в него новые типы, соответствующие его потребностям. Система программирования принимает новые классы, обеспечивает для них всю обработку и проверку типов наравне со встроенными типами.

Область применения объектно-ориентированного подхода не ограничивается имитацией. Согласны вы с тем, что любая программа имитирует проектируемую систему, или нет, применение методов ООП помогает сводить трудные задачи к простым решениям.

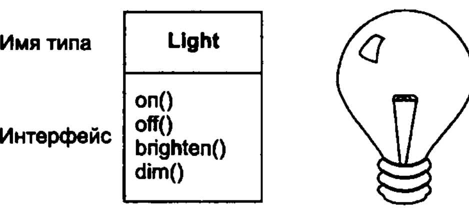
После определения класса вы можете создать сколько угодно объектов этого класса и работать с ними как с элементами пространства решения задачи. Одна из основных трудностей объектно-ориентированного программирования как раз и заключается в создании однозначного соответствия между элементами пространства задачи и объектами в пространстве решения.

Но как заставить объект сделать нечто полезное? Необходимо передать объекту запрос на выполнение некоторой операции, например зачислить деньги на счет,

---

[1] Некоторые авторы различают эти два понятия. Они утверждают, что тип определяет интерфейс, а класс — конкретную реализацию этого интерфейса.

нарисовать что-нибудь на экране или повернуть выключатель. Каждый объект обрабатывает только конкретные разновидности запросов. Совокупность этих запросов образует *интерфейс*, который определяется типом объекта. Рассмотрим простой пример — электрическую лампочку:



Интерфейс определяет лишь то, *какие* запросы обрабатываются данным объектом. Где-то внутри объекта должен находиться программный код, который эти запросы обрабатывает. Этот код вместе со скрытыми данными образует *реализацию*. С позиций процедурного программирования в этой схеме нет ничего сложного. В типе с каждым возможным запросом ассоциируется некоторая функция, которая вызывается при получении запроса. При описании этого процесса обычно говорят, что программист «отправляет сообщение» («обращается с запросом») к объекту, а объект решает, что делать с этим сообщением (какой фрагмент кода выполнить).

В приведенном примере имеется тип/класс с именем Light и конкретный объект класса Light с именем It. К объекту Light можно обратиться со следующими запросами: включиться (on()), выключиться (off()), прибавить или убавить яркость (brighten() и dim()). При создании объекта класса Light указывается имя этого объекта (It). Чтобы отправить сообщение объекту, вы указываете имя объекта и отделяете его от сообщения точкой (.). С точки зрения пользователя готового класса, в этом и заключается все программирование с применением объектов.

Приведенная выше диаграмма оформлена по стандарту *UML* (Unified Modeling Language). Каждый класс представлен отдельным блоком; в верхней части блока находится имя класса, в средней — все переменные класса, которые вы желаете документировать, а в нижней части — *функции класса* (функции, которые принадлежат данному объекту и обрабатывают получаемые сообщения). Довольно часто на UML-диаграммах отображаются только имя и открытые функции класса, в этом случае средняя часть блока отсутствует. А если важно только имя класса, то и нижняя часть блока становится необязательной.

Скрытая реализация

Для удобства всех программистов можно условно разделить на *создателей классов* (тех, кто определяет новые типы данных) и *прикладных программистов* («потребителей», использующих типы данных в своих приложениях). Прикладные программисты собирают библиотеки классов и используют их для ускорения разработки приложения. Создатель класса стремится построить класс, который предоставляет прикладному программисту доступ лишь к тому, что необходимо для его работы, и скрывает все остальное. Почему? Скрытые аспекты класса не могут использоваться прикладными программистами, поэтому создатель класса может свободно

изменять их, не беспокоясь о том, как эти изменения отразятся на других. Скрытая часть класса обычно представляет чувствительные «внутренности» объекта, которые могут быть легко испорчены прикладным программистом по неосторожности или от недостатка информации, поэтому скрытие реализации уменьшает количество ошибок в программе. Скрытие реализации — одна из важнейших концепций ООП, ее важность трудно переоценить.

В любых отношениях важно установить границы, которые должны соблюдаться всеми участниками этих отношений. При создании библиотеки вы устанавливаете отношения с прикладным программистом, который использует вашу библиотеку в своих приложениях — возможно, для построения более крупной библиотеки.

Если все члены класса будут доступны всем желающим, то прикладной программист сможет делать с классом все что угодно и обеспечить соблюдение правил будет в принципе невозможно. Даже если прямое обращение к некоторым членам классов нежелательно, его не удастся предотвратить без ограничения доступа.

Итак, существует несколько причин для ограничения доступа. Во-первых, некоторые аспекты класса должны быть защищены от вмешательства прикладных программистов. Речь идет об аспектах, задействованных во внутренней работе типа данных, но не входящих в его интерфейс, применяемый для решения практических задач пользователей. На самом деле ограничение доступа не мешает, а помогает прикладному программисту выделить то, что действительно существенно для его работы.

Во-вторых, ограничение доступа позволяет проектировщику библиотеки изменить внутренний механизм работы класса, не беспокоясь о том, как эти изменения отразятся на прикладных программистах. Допустим, в первой версии для ускорения разработки использовалась упрощенная реализация, а потом вдруг выяснилось, что класс необходимо переписать заново для ускорения его работы. Четкое отделение интерфейса от реализации позволяет легко решить эту задачу, после чего пользователю останется лишь перекомпоновать программу.

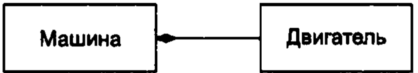
Для определения уровней доступа в C++ используются три ключевых слова: public, private и protected. Их смысл весьма прямолинеен — эти *спецификаторы доступа* указывают, кто может работать с объявлениями членов класса, следующими за ними. Спецификатор public означает, что следующие объявления доступны для всех. С другой стороны, спецификатор private означает, что доступ к следующим объявлениям возможен только из функций данного типа. Фактически он разделяет «зоны ответственности» создателя класса и прикладного программиста. При попытке обратиться извне к закрытому (private) члену класса происходит ошибка компиляции. Спецификатор protected аналогичен private с одним отличием: производные классы могут обращаться к защищенным (protected) членам класса, но доступ к закрытым (private) членам для них запрещен. О наследовании и производных классах мы поговорим чуть позже.

Повторное использование реализации

Созданный и протестированный класс должен (в идеале) быть достаточно самостоятельным, чтобы его можно было использовать повторно в других программах. Впрочем, на практике добиться этого сложнее, чем хотелось бы; качественное проектирование классов требует опыта и хорошей интуиции. Но если такой класс

будет создан, было бы обидно не задействовать его. Возможность повторного использования кода относится к числу важнейших преимуществ объектно-ориентированного программирования.

В простейшем случае вы просто напрямую создаете объект соответствующего класса, однако существует и другой способ — включить объект этого класса внутрь нового класса. Это называется «созданием вложенного объекта». Новый класс может содержать сколько угодно объектов произвольных типов в любой комбинации, необходимой для достижения желаемого поведения нового класса. Методика создания нового класса из существующих классов называется ***композицией*** (также встречается более общий термин — ***агрегирование***). О композиции также часто говорят как об «отношении принадлежности» по принципу «у машины есть двигатель».



(На этой UML-диаграмме композиция обозначена закрашенным ромбом, который указывает, что двигатель принадлежит только одной машине. Автор обычно изображает такие связи в упрощенном виде — только линией без ромба.)

Композиция обладает чрезвычайно гибкими возможностями. Вложенные объекты нового класса обычно объявляются закрытыми, что делает их недоступными для прикладных программистов, работающих с классом. С другой стороны, создатель класса может изменять эти объекты, не нарушая работы существующего клиентского кода. Кроме того, замена вложенных объектов на стадии выполнения программы позволяет динамически изменять ее поведение. Механизм наследования (см. ниже) такой гибкостью не обладает, поскольку для производных классов устанавливаются ограничения, проверяемые на стадии компиляции.

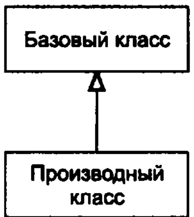
Наследование играет особую роль в объектно-ориентированном программировании. Многие авторы особо подчеркивают его важность, и у неопытных программистов может сложиться впечатление, что наследование требуется в любых ситуациях. Это часто приводит к появлению громоздких, чрезмерно усложненных архитектур. При создании новых классов сначала подумайте, нельзя ли воспользоваться более простым и гибким механизмом композиции. А когда у вас появится практический опыт, станет вполне очевидно, в каких случаях лучше подходит наследование.

Наследование и повторное использование интерфейса

Сама по себе идея объекта чрезвычайно удобна. Она позволяет объединить данные и функциональность на ***концептуальном*** уровне, чтобы программист работал в контексте пространства решения задачи и не ограничивался идиоматическими представлениями базового компьютера. Эти концепции выражаются на уровне базового синтаксиса языка программирования ключевым словом class.

Но было бы обидно потрудиться над созданием класса, а потом заново определять новый класс, обладающий похожими возможностями. Гораздо удобнее взять

существующий класс, скопировать его, а затем внести дополнения и изменения в копию. В сущности, именно это происходит при наследовании, но с одним исключением: изменения, вносимые в исходный класс (также называемый *базовым классом, родительским классом* или *суперклассом*), будут отражены и в его «копии» (называемой *производным классом, дочерним классом* или *субклассом*).

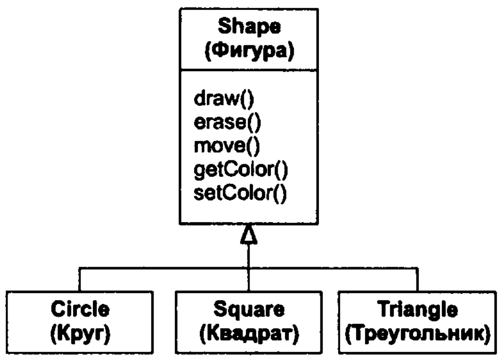


(Стрелка на приведенной UML-диаграмме направлена от производного класса к базовому. Как будет показано ниже, один базовый класс может иметь несколько производных классов.)

Тип не ограничивается определением общих ограничений для набора объектов; он также участвует в системе отношений с другими типами. Два типа могут обладать общими характеристиками и поведением, но один из них может обладать дополнительными характеристиками или обрабатывать дополнительные сообщения (или обрабатывать их иным образом). В механизме наследования сходство между типами выражается концепциями базовых и производных типов. Базовый тип имеет все характеристики и аспекты поведения всех типов, производных от него. Программист создает базовый тип для представления важнейших особенностей объектов системы. Затем от базового типа наследуются производные типы, выражающие разные варианты реализации этих важнейших особенностей.

Предположим, вы программируете мусороуборочную машину для сортировки разных типов мусора. В этом случае базовый тип «мусор» определяет общие характеристики (вес, ценность и т. д.) и общие операции (измельчение, переработка и т. д.). Из базового типа (мусора) производятся конкретные типы отходов, обладающие дополнительными характеристиками (например, цветом стекла у бутылки) или аспектами поведения (скажем, возможностью прессовки алюминиевых банок). Кроме того, некоторые аспекты производного типа могут отличаться от аспектов базового прототипа (например, ценность бумаги определяется в зависимости от ее плотности и состояния). Путем наследования программист строит иерархию типов, которая должна выражать решаемую задачу в контексте этих типов.

Рассмотрим другой, классический пример (возможно, взятый из системы автоматизированного проектирования или компьютерной игры). Базовый тип «геометрическая фигура» определяет свойства, общие для всех фигур: размер, цвет, положение и т. д. С фигурами выполняются различные операции: вывод, стирание, перемещение, закраска и т. д. Из этого базового типа производятся типы конкретных видов фигур (кругов, квадратов, треугольников и т. д.), обладающих дополнительными характеристиками и поведением. Например, к некоторым фигурам может применяться операция зеркального отражения. В иерархии типов воплощаются сходства и различия между разными фигурами.

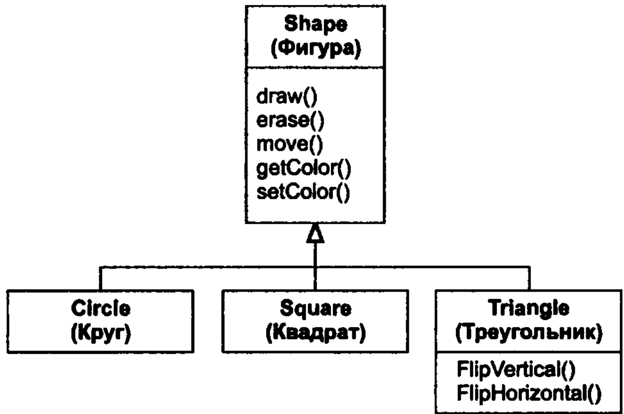


Формулировка решения в контексте задачи приносит огромную пользу, поскольку она избавляет от применения промежуточных моделей для перехода от описания задачи к описанию решения. При использовании объектов иерархия типов становится основной моделью, что позволяет перейти от описания системы в реальном мире к описанию системы на уровне программного кода. Оказывается, концептуальная простота объектно-ориентированного проектирования даже сбивает с толку некоторых программистов. Разум, привыкший к поиску сложных решений, приходит в замешательство от этой простоты.

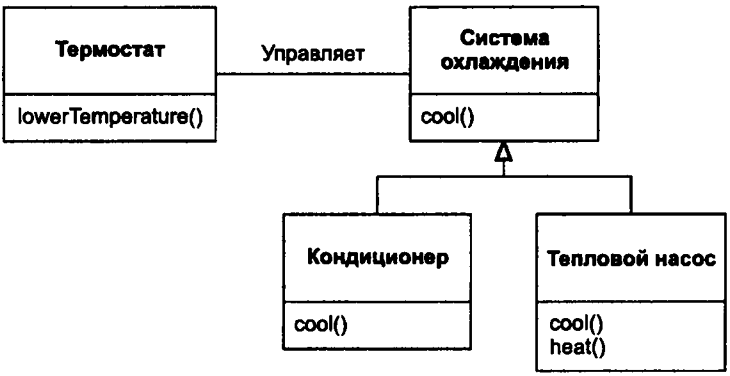
Объявляя новый тип производным от существующего типа, вы создаете новый тип. Этот новый тип не только содержит все члены существующего типа (хотя члены со спецификатором private скрыты и недоступны для производного класса), но что еще важнее — он также воспроизводит интерфейс базового класса. Иначе говоря, все сообщения, которые принимались объектами базового класса, также будут приниматься объектами производного класса. Поскольку мы знаем, что класс определяется теми сообщениями, которые он принимает, можно сказать, что *производный класс является частным случаем базового класса*. Так, в предыдущем примере это означает, что «круг является частным случаем геометрической фигуры». Эквивалентность типов на базе наследования — одно из важнейших положений, необходимых для понимания сути объектно-ориентированного программирования.

Базовый и производный классы обладают общим интерфейсом, поэтому было бы естественно предположить, что вместе с интерфейсом совместно используется некоторая реализация. При получении сообщения объект выполняет некоторый фрагмент кода. Если просто объявить класс производным и не делать ничего более, методы интерфейса базового класса перейдут в производный класс. В этом случае объекты производного класса не только являются частным случаем базового класса, но и ведут себя точно так же, что не особенно интересно.

Существуют два способа модификации производных классов по отношению к базовым. Первый способ тривиален: в производный класс просто добавляются новые функции, не входящие в интерфейс базового класса. Если базовый класс не обладает возможностями, необходимыми для некоторого частного случая, он расширяется в производном классе. Такое простое и даже примитивное применение наследования часто оказывается идеальным решением проблемы, но при расширении следует хорошо подумать, не понадобятся ли эти дополнительные функции в базовом классе. При объектно-ориентированном программировании рекомендуется периодически производить итеративный анализ и переработку структур классов.



Хотя при наследовании интерфейсы часто дополняются новыми функциями, это не обязательно. Второй, более важный, способ модификации производных классов основан на *переопределении*, то есть *изменении поведения* существующих функций базового класса.



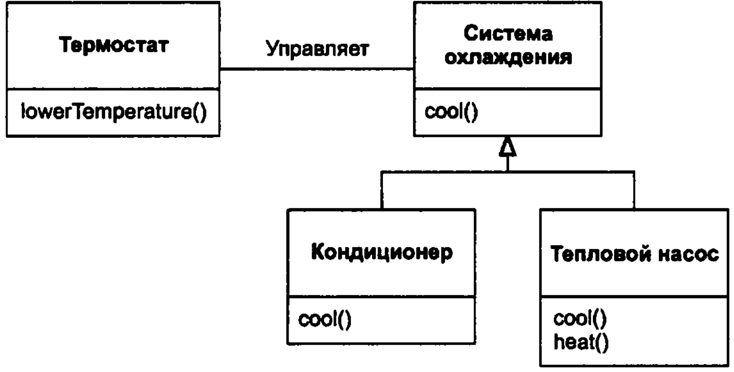
Чтобы переопределить функцию, достаточно создать в производном классе новое определение функции. В сущности, это означает: «Мы используем ту же интерфейсную функцию, но в производном типе она будет делать нечто иное».

Точное и приблизительное подобие

В отношении наследования возникает один важный вопрос: должно ли наследование ограничиваться переопределением функций базового класса (без добавления новых функций, отсутствующих в базовом классе)? В этом случае производный тип в *точности* эквивалентен базовому классу, поскольку он обладает точно таким же интерфейсом. Следовательно, объекты производного класса могут свободно использоваться вместо объектов базового класса. Иногда это называется *чистой подстановкой*, или *принципом подстановки*. В каком-то смысле такой подход к наследованию может считаться идеальным. Базовый и производный классы связаны *отношением точного подобия*, когда вы можете сказать, что круг *является* геометрической фигурой. На этом факте основан один из способов проверки иерархии

наследования: попробуйте сформулировать отношение точного подобия между классами и посмотрите, насколько осмысленно оно выглядит.

Иногда в производные типы приходится добавлять новые элементы интерфейса, что приводит к расширению интерфейса и созданию нового типа. Новый тип по-прежнему может использоваться вместо базового, однако эта подстановка не идеальна, потому что новые функции остаются недоступными для базового типа. Такая ситуация описывается как *отношение приблизительного подобия*; новый тип поддерживает интерфейс старого типа, но он также содержит другие функции, поэтому говорить о точном совпадении интерфейсов не приходится. Представьте, что в вашем доме установлена аппаратура (интерфейс) для управления охлаждающими системами. Допустим, кондиционер сломался и был заменен тепловым насосом, который может как охлаждать, так и подогревать воздух. Тепловой насос *является аналогом* кондиционера, но он способен на большее. Поскольку аппаратура рассчитана только на охлаждение, она сможет взаимодействовать только с подсистемой охлаждения нового объекта. Интерфейс объекта расширился, но существующая система «знает» только об исходном интерфейсе.



Конечно, при первом взгляде на эту иерархию становится ясно, что базовый класс «система охлаждения» ограничен и его следовало бы заменить «системой терморегулирования» с поддержкой подогрева, — в этом случае принцип подстановки заработает нормально. Тем не менее приведенная выше диаграмма дает пример того, что нередко происходит при проектировании иерархий классов и в реальном мире.

При первом знакомстве с принципом подстановки часто кажется, что этот подход (чистая подстановка) является единственно верным. Если вам удастся ограничиться этим способом — что ж, превосходно. Но на практике включение новых функций в интерфейс производного класса часто оказывается вынужденной мерой. В результате тщательного анализа оба случая выявляются достаточно очевидно.

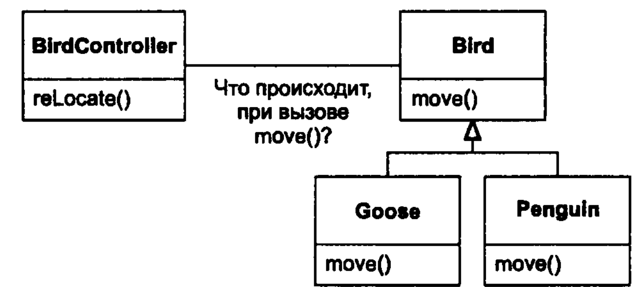
Взаимозаменяемость объектов и полиморфизм

При использовании иерархий типов часто требуется интерпретировать объект не с фактическим, а с базовым типом. Это позволяет программировать независимо от конкретных типов. Так, в примере с геометрическими фигурами функции работа-

ют с обобщенными фигурами независимо от того, являются ли они кругами, квадратами, треугольниками и т. д. Нарисовать, стереть или переместить можно любую фигуру, поэтому эти функции просто передают соответствующее сообщение объекту геометрической фигуры; они не заботятся о том, как объект обработает это сообщение.

Такой код не зависит от введения новых типов — самого распространенного способа расширения объектно-ориентированных программ для обработки новых ситуаций. Например, определение новой разновидности геометрической фигуры (пятиугольник) не повлияет на работу функций, работающих с обобщенными фигурами. Возможность простого расширения программ посредством объявления новых производных типов играет важную роль, поскольку она существенно улучшает архитектуру программы с одновременным сокращением затрат на ее сопровождение.

Тем не менее интерпретация объектов производных типов в контексте базового типа (круг как геометрическая фигура, велосипед как средство передвижения, ворона как птица и т. д.) не обходится без проблем. Если функция должна приказать обобщенной геометрической фигуре нарисовать себя, обобщенному средству передвижения — повернуть, обобщенной птице — лететь и т. д., на стадии компиляции еще неизвестно, какой фрагмент кода должен при этом выполняться. Собственно, в этом и заключается вся соль — при отправке сообщения программист *не желает* знать, какой фрагмент кода будет выполняться. Функция вывода может применяться к кругам, квадратам и треугольникам, а объект должен выполнить правильный код в зависимости от фактического типа. Если вам не нужно знать, какой именно фрагмент кода будет выполняться, то при добавлении нового подтипа выполняемый код будет выбран без изменения вызова функции. Итак, если компилятор не может заранее узнать, какой фрагмент кода будет выполняться, что же ему делать? Например, на следующей диаграмме объект BirdController работает с обобщенными объектами Bird (птица) и не знает их точного типа. С точки зрения BirdController это удобно, поскольку этому классу не нужно определять конкретный подтип Bird, с которым он работает, или проверять поведение этого подтипа. Так как же при вызове функции move() без определения конкретного типа Bird выбирается правильное поведение? Ведь гусь (Goose) летает, плавает и бегает, а пингвин (Penguin) только плавает и бегает.



Ответ на этот вопрос кроется в одном из трюков объектно-ориентированного программирования. Компилятор не может сгенерировать вызов функции в традиционном смысле. В вызовах функций, сгенерированных обычным компилятором,

применяется так называемое *раннее связывание*. Возможно, многим читателям этот термин еще не встречался, потому что они и не подозревали о существовании другой схемы связывания. Термин означает, что компилятор генерирует вызов функции с конкретным именем, а компоновщик преобразует этот вызов в абсолютный адрес выполняемой функции. В ООП программа не может определить адрес функции до выполнения программы, поэтому при отправке сообщения обобщенному объекту необходима другая схема.

Для решения этой проблемы в объектно-ориентированных языках применяется *позднее*, или *динамическое, связывание*. При отправке сообщения объекту функция, которой должно быть передано управление, не определяется до времени выполнения программы. Компилятор всего лишь убеждается в том, что функция существует, проверяет типы аргументов и возвращаемого значения (языки, в которых эта проверка не выполняется, называются *языками со слабой типизацией*), но не знает, куда именно будет передано управление.

Для выполнения позднего связывания компилятор C++ заменяет абсолютный вызов функции специальным фрагментом кода, который вычисляет адрес тела функции по информации, хранящейся в объекте (эта тема подробно рассматривается в главе 15). Таким образом, разные объекты по-разному ведут себя в зависимости от этого фрагмента. При получении сообщения объект сам решает, как его обработать.

Виртуальные функции (то есть функции, наделенные гибкими возможностями позднего связывания) объявляются с ключевым словом virtual. Чтобы использовать виртуальные функции в программах, не обязательно понимать механику их вызова, но без виртуальных функций объектно-ориентированное программирование на C++ невозможно. В C++ необходимо помнить о добавлении ключевого слова virtual, потому что по умолчанию функции классов *не используют* динамическое связывание. В виртуальных функциях проявляются различия в поведении классов, принадлежащих к одному семейству. Эти различия заложены в основу полиморфизма.

Вернемся к примеру с фигурами. Диаграмма с семейством классов, использующих единый интерфейс, была приведена выше в этой главе. Для демонстрации полиморфизма мы напишем фрагмент кода, который будет игнорировать особенности конкретных типов и взаимодействовать только с базовым классом. Такой код изолируется от информации конкретных типов, он проще в программировании и сопровождении. А если в иерархию будет добавлен новый производный тип, например многоугольник (Hexagon), то этот фрагмент будет работать с новой разновидностью типа Shape так же, как он работал с существующими типами. Тем самым обеспечивается *расширяемость* программы.

На C++ эта функция выглядит примерно так:

```
void doStuff(Shape& s) {
  s.erase();
  // ...
  s.draw();
}
```

Функция работает с любыми объектами типа Shape и не зависит от конкретного типа выводимых и стираемых объектов (суффикс & означает «Получить адрес объекта, передаваемого функции doStuff()», но вам пока не обязательно разбираться в этих тонкостях). Вызов функции doStuff() в другой части программы выглядит так:

```
Circle c;
Triangle t;
Line l;
doStuff(c);
doStuff(t);
doStuff(l);
```

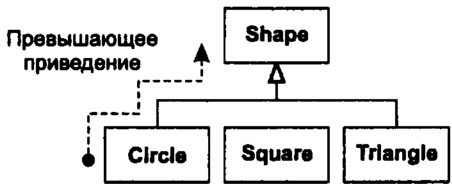
Вызовы doStuff() автоматически работают правильно независимо от фактического типа объекта.

На самом деле это очень интересный факт. Рассмотрим следующую строку:

```
doStuff(c);
```

Здесь происходит следующее: функции, которая ожидает получить объект типа Shape, передается объект типа Circle. Поскольку класс Circle является частным случаем Shape, он может интерпретироваться как таковой функцией doStuff(). Иначе говоря, любое сообщение, которое функция doStuff() может отправить объекту Shape, будет успешно принято также и объектом Circle. Все предельно безопасно и логично.

Интерпретация объекта производного типа так, как если бы он относился к базовому типу, называется *повышающим приведением типа*. Термин «повышающее» происходит от типичной структуры диаграмм наследования, при которой базовый тип расположен сверху, а производные классы выстроены внизу. Таким образом, приведение к базовому типу соответствует перемещению вверх на диаграмме наследования.



В объектно-ориентированных программах обычно в том или ином виде задействовано повышающее приведение типа, потому что оно избавляет программиста от необходимости знать конкретный тип объектов, с которыми он работает. Вспомните функцию doStuff():

```
s.erase();
// ...
s.draw();
```

Обратите внимание: мы не говорим «для объекта Circle сделать то, для объекта Square — сделать это». Программа, анализирующая все возможные разновидности Shape, получится слишком громоздкой, вдобавок ее придется изменять при каждом добавлении новой разновидности Shape. В данном случае мы говорим: «Фигура, я знаю, что ты умеешь самостоятельно обрабатывать вызовы erase() и draw(); сделай это и позаботься обо всех деталях».

Интересно, что код функции doStuff() каким-то образом делает именно то, что требуется. При вызове draw() для Circle будет выполнен совсем не тот код, который выполняется при вызове draw() для Square или Line, а при отправке сообщения draw() неизвестной разновидности Shape правильное поведение выбирается автоматически в соответствии с фактическим типом объекта. И это выглядит совершенно

удивительно, потому что, как упоминалось выше, при компиляции функции doStuff() компилятор не обладает информацией о типах, с которыми он работает. Следовательно, в обычной ситуации можно было бы предположить, что будут вызваны версии функций erase() и draw() класса Shape, а не конкретных подтипов Circle, Square и Line. Тем не менее благодаря полиморфизму все работает правильно. Обо всех деталях позаботятся компилятор и система времени выполнения. Программист должен знать лишь то, что этот механизм работает, и уметь применять его при проектировании. Если функция класса объявлена виртуальной, то при получении сообщения объект автоматически выполнит правильные действия, даже если при этом потребуется повышающее приведение типа.

Создание и уничтожение объектов

С технической точки зрения основными проявлениями ООП считаются абстрактные типы данных, наследование и полиморфизм, но существуют и другие, не менее важные аспекты. В этом разделе приводится их краткий обзор.

Особого внимания заслуживают операции создания и уничтожения объектов. Где хранятся данные объекта и как определяется продолжительность его жизненного цикла? В разных языках программирования используются разные решения. В подходе, принятом в C++, важнейшим фактором считается контроль над эффективностью, поэтому программисту предоставляется выбор. Если он стремится добиться максимальной скорости выполнения, то место хранения и продолжительность жизненного цикла объектов определяются на стадии написания программы; для этого объекты размещаются в стеке или в статической памяти. *Стеком* называется область памяти, которая напрямую используется процессором во время выполнения программы. Переменные, хранящиеся в стеке, также называются *автоматическими* переменными. *Статической памятью* называется фиксированный блок памяти, выделяемый перед началом выполнения программы. При хранении данных в стеке или статической памяти на первое место ставится скорость выделения и освобождения памяти, очень существенная в некоторых ситуациях. Однако за нее приходится расплачиваться гибкостью, так как программист должен точно знать количество, жизненный цикл и тип объектов *во время* написания программы. Если задача плохо поддается прогнозированию (системы управления складом, авиадиспетчерские системы и т. д.), это приводит к чрезмерным ограничениям.

Во втором подходе объекты создаются динамически в специальном пуле памяти, называемом *кучей*. В этом случае программист до момента выполнения программы не знает, сколько объектов потребуется, каким будет их жизненный цикл или фактический тип. Эти решения принимаются «на месте» во время выполнения программы. Если программисту понадобится новый объект, он просто создает его в куче при помощи ключевого слова new. Когда объект становится ненужным, он ликвидируется при помощи ключевого слова delete.

Поскольку операции с кучей производятся динамически на стадии выполнения, выделение памяти занимает гораздо больше времени, чем при создании объектов в стеке (выделение памяти в стеке часто сводится к одной команде процессора для смещения указателя стека). Динамический подход основан на разумном допущении, что для достаточно сложных объектов дополнительные затраты на по-

иск области памяти и ее освобождение не оказывают заметного влияния на общую продолжительность операции. Кроме того, более гибкий характер динамического выделения играет важную роль при решении общих задач программирования.

Однако наряду с этими факторами существует и другой — продолжительность жизни объекта. Если объект создается в стеке или в статической памяти, продолжительность его существования определяет компилятор, автоматически уничтожая объект. Но если объект создается в куче, компилятор не располагает информацией о сроке его существования. В C++ программист должен самостоятельно уничтожить объект в своей программе при помощи ключевого слова delete. Впрочем, существует и другой вариант — в среде выполнения может поддерживаться механизм *уборки мусора*, который автоматически обнаруживает неиспользуемые объекты и уничтожает их. Конечно, уборка мусора существенно упрощает программирование, но, с другой стороны, она приводит к дополнительным затратам ресурсов. Этот факт не соответствовал требованиям, принятым при проектировании C++, поэтому механизм уборки мусора не стал частью языка (хотя для C++ существуют уборщики мусора, разработанные сторонними производителями).

Обработка исключений

Со времен появления первых языков программирования обработка ошибок вызывала массу проблем. Разработать хорошую схему обработки ошибок чрезвычайно трудно, поэтому многие языки просто игнорировали эту проблему и перепоручали ее проектировщикам библиотек; в свою очередь, те предлагали компромиссные решения, которые часто работали, но во многих ситуациях легко обходились (как правило, их можно было просто игнорировать). Главным недостатком большинства схем обработки ошибок была их зависимость от того, насколько прилежно программистом выполнялись определенные правила, соблюдение которых не обеспечивалось на уровне языка. Если программист не проявлял должного усердия (как это часто бывает в спешке), все эти схемы становились бесполезными.

Механизм *обработки исключений* переводит обработку ошибок на уровень языка программирования, а иногда даже на уровень операционной системы. Исключение представляет собой объект, который «генерируется» в месте ошибки и «перехватывается» соответствующим *обработчиком исключения*, предназначенным специально для обработки этого конкретного типа ошибки. Со стороны все выглядит так, словно при возникновении ошибки выполнение программы идет по другой, параллельной ветви. А это означает, что код обработки ошибок может быть отделен от кода нормального выполнения. Подобная изоляция упрощает программу, поскольку программисту не приходится постоянно проверять все возможные ошибки. Кроме того, сгенерированное исключение принципиально отличается от возвращаемого численного кода или флага, устанавливаемого функцией для обозначения ошибки, — эти признаки могут просто игнорироваться. Исключения не игнорируются, поэтому рано или поздно они заведомо будут обработаны. Наконец, исключения обеспечивают восстановление после аварийных ситуаций. Вместо завершения программы нередко удается исправить ошибку и продолжить работу, что значительно повышает устойчивость системы.

Учтите, что обработка исключений не является объектно-ориентированным средством, хотя в объектно-ориентированных языках исключения обычно представляются объектами. Механизм обработки исключений появился раньше первых объектно-ориентированных языков.

Анализ и проектирование

Объектно-ориентированная парадигма требует принципиально нового подхода к программированию, и многие программисты поначалу даже не представляют, с чего начать работу над ООП-проектом. Свыкнувшись с мыслью, что все данные должны быть объектами, и научившись мыслить в объектно-ориентированном ключе, вы постепенно начнете создавать «хорошие» архитектуры, в полной мере использующие все преимущества ООП.

Методологией называется совокупность процессов и эвристических подходов, применяемых для упрощения сложных задач программирования. С момента появления объектно-ориентированного программирования было предложено немало разных методологий. В этом разделе вы получите представление о том, чего же мы пытаемся добиться при использовании методологий в своей работе.

В области методологии (и ООП в частности) проводится множество экспериментов, поэтому вы должны хорошо понять, какие проблемы призван решить тот или иной метод, прежде чем переходить на него. Это особенно справедливо по отношению к C++ — языку, который должен был упростить выражение программных концепций по сравнению с C. Иногда даже можно обойтись без более сложных методологий. Простыми средствами в C++ решается гораздо более широкий круг задач, чем с помощью простых методологий в процедурных языках.

Также следует понимать, что термин «методология» нередко оказывается слишком пышным и обещает слишком много. Все, что вы делаете во время проектирования и написания программы, является методом. Возможно, вы сами изобрели этот метод и даже не отдаете себе отчета в его применении, однако это и есть тот самый процесс, через который вы проходите в процессе творения. Если процесс получился эффективным, возможно, для его воплощения на C++ достаточно минимальной доработки. Если вас не устраивает производительность вашего труда и то, какими получаются ваши программы, подумайте о переходе на другую формальную методику или поищите нужные составляющие из нескольких формальных методов.

В процессе разработки самое важное правило звучит так: «Не заблудитесь». Такое происходит очень часто. Многие методы из области аналитики и проектирования ориентированы на решение глобальных задач. Помните, что большинство проектов не относится к этой категории, поэтому обычно успешный анализ и проектирование удается провести на базе небольшого подмножества того, что рекомендует некоторый метод. И все же некий упорядоченный подход, сколь бы ограниченным он ни был, обычно помогает взяться за дело гораздо лучше, чем если вы просто начнете программировать.

Другая опасность — «аналитический паралич», когда программист считает, что двигаться дальше можно только после досконального анализа всех деталей на текущей стадии. Помните: даже при самом тщательном анализе в системе непременно отыщутся какие-нибудь факторы, которые проявятся лишь на стадии проектирования. Еще больше непредвиденных факторов обнаружится во время програм-

мирования, а то и тогда, когда программа будет запущена в эксплуатацию. Из-за этого желательно побыстрее пройти стадии анализа и проектирования и перейти к тестированию предполагаемой системы.

Последнее обстоятельство стоит подчеркнуть особо. Опыт использования процедурных языков подсказывает, что тщательное планирование и понимание мельчайших деталей перед выходом на проектирование и реализацию весьма похвальны. Конечно, при создании СУБД необходимо добросовестно проанализировать все потребности клиента. Однако СУБД принадлежит к классу хорошо формализуемых и понятных задач; во многих программах такого рода основной проблемой является *выбор структуры* базы данных. В этой главе речь идет о совершенно ином круге задач, решение которых не может быть получено простой переработкой хорошо известного решения, так как содержит непредсказуемые факторы — элементы, не имеющие очевидных готовых решений и требующие исследований[1]. Попытки всестороннего анализа перед переходом к проектированию и реализации в задачах такого рода вызывают «аналитический паралич», потому что на стадии анализа вы не располагаете достаточной информацией. Решение таких задач требует итераций по всему рабочему циклу, а это связано с определенными рисками (впрочем, это логично — вы пытаетесь сделать нечто новое, поэтому потенциальный выигрыш тоже увеличивается). Может показаться, что ускоренный переход к предварительной реализации лишь увеличивает этот риск, но на самом деле происходит обратное: риск уменьшается, потому что вы на ранней стадии узнаете о неприемлемости тех или иных подходов. Разработка программных продуктов требует грамотного принятия решений в условиях риска.

Часто предлагается «создать первый образец, выкинуть его и начать заново». ООП позволяет выкинуть *часть* первого образца, но поскольку код инкапсулируется в классах, даже на первой итерации вы неизбежно создадите некоторые полезные структуры классов и концепции системной архитектуры, которые стоит оставить. Таким образом, первый, ускоренный «заход на цель» не только предоставляет бесценную информацию для последующего анализа, проектирования и реализации, но и закладывает кодовую базу для дальнейших итераций.

Если методология, с которой вы знакомитесь, содержит огромное количество деталей, состоит из множества этапов и поясняется массой документов, вам все равно будет трудно понять, в какой момент следует остановиться. Помните, что вы ищете ответы всего на два вопроса.

- Какие объекты будут задействованы в проекте? (Как разбивать проект на составляющие?)
- Какими интерфейсами они будут обладать? (Какие сообщения должны отправляться каждому объекту?)

Даже если у вас не будет ничего, кроме представлений об объектах и их интерфейсах, можно переходить к написанию программы. По различным причинам вам могут потребоваться другие описания и документы, но меньшим обойтись невозможно.

---

[1] Эмпирическое правило оценки проектов такого рода, придуманное автором, гласит: если проект содержит более одного непредсказуемого фактора, даже не пытайтесь планировать затраты или сроки завершения проекта до создания рабочего прототипа. Слишком большое количество возможных вариантов развития событий все равно не позволит этого сделать.

Процесс условно делится на пять фаз с дополнительной нулевой фазой, на которой просто принимается решение об использовании той или иной структуры.

Фаза 0 — составление плана

Сначала следует решить, из каких этапов будет состоять процесс. На первый взгляд звучит просто (собственно, *все* формулировки такого рода будут звучать просто), но нередко подобные решения принимаются уже после начала программирования. Если вы руководствуетесь планом «Поскорее взяться за программирование, а дальше ориентироваться по обстановке» — что ж, замечательно (при решении очевидных проблем иногда достаточно и этого). По крайней мере, осознайте, что ваш план звучит именно так.

На данном этапе также можно решить, что процесс нуждается в дополнительном структурировании, но увлекаться тоже не стоит. Вполне естественно, что некоторые программисты любят работать в «отпускном режиме», когда процесс разработки не подчиняется никаким структурным ограничениям: «Когда сделаем, тогда и сделаем». Но на собственном опыте автор убедился, что наличие промежуточных контрольных точек помогает сосредоточиться и направляет усилия программиста к этим контрольным точкам, не позволяя ему зациклиться на одной цели — «завершении проекта». Кроме того, проект делится на части, с которыми удобнее работать и которые выглядят не столь устрашающе (не говоря уже о том, что прохождение контрольной точки является хорошим поводом для застолья).

Когда автор решил изучить структуру литературного произведения (чтобы когда-нибудь написать роман), поначалу не воспринималась сама идея структурирования — казалось, нужно просто переносить на страницу те мысли, которые рождаются в голове. Позднее стало ясно, что по крайней мере в книгах по программированию структура достаточно очевидна и о ней можно не думать. Тем не менее свои работы автор все равно структурирует, пусть даже неформально и полусознательно. Следовательно, даже если ваш план сводится к тому, чтобы поскорее взяться за программирование, все равно стоит пройти следующие фазы, задать и ответить на некоторые вопросы.

Формулировка задачи

Любая система, над которой вы работаете, независимо от сложности имеет некоторую фундаментальную цель — ту основную потребность, которую она должна удовлетворять. Если заглянуть за пользовательский интерфейс, за аппаратно- или системно-зависимые детали, за алгоритмы и проблемы эффективности, вы в итоге придете к истинной сути задачи, простой и прямолинейной. Она, как и «сверхконцепции» голливудских фильмов, формулируется в одной-двух фразах. Именно это «отфильтрованное» описание должно стать отправной точкой для решения задачи.

Сверхконцепция, то есть формулировка задачи, весьма важна, потому что она задает тон для всего проекта. Не всегда удается правильно сформулировать задачу с первого раза (возможно, она станет окончательно ясной лишь на более поздней стадии проекта), но, по крайней мере, продолжайте попытки до тех пор, пока результат вас не удовлетворит. Например, в системе управления воздушным движением можно начать с формулировки, ориентированной на конкретную систему, над которой вы работаете: «Диспетчерская вышка отслеживает перемещения самолетов». Но подумайте, что произойдет, если уменьшить систему до очень маленького аэропорта,

где никакой вышки может и не быть (а то и диспетчера). Более полезная модель ориентирована не столько на конкретные условия, сколько на суть задачи: «Самолеты прибывают, разгружаются, проходят техобслуживание, загружаются и улетают».

Фаза 1 — что делать

В предыдущем поколении программных архитектур (называемых *процедурными архитектурами*) данная фаза называлась *«анализом требований и разработкой системных спецификаций»*. В результате на свет появлялись документы с устрашающими названиями, каждый из которых вполне годился для отдельного крупного проекта. Конечно, в таких дебрях было легко заблудиться, хотя сами по себе эти подзадачи были весьма полезными. Под «анализом требований» понималось следующее: «Составить список условий, по которому мы узнаем, когда работа завершена и ее нужно передавать заказчику». Под разработкой системных спецификаций понимается: «Описание того, *что* (но не *как*) должна делать наша программа, чтобы удовлетворять поставленным требованиям». Фактически анализ требований определяет контракт между вами и заказчиком (даже если заказчик работает внутри вашей компании либо является каким-то другим объектом или системой). Системная спецификация представляет собой высокоуровневое исследование проблемы, в ходе которого вы определяете, можно ли решить поставленную задачу и сколько времени это займет. Поскольку обе подзадачи должны быть согласованы между разными людьми (а также из-за того, что они обычно изменяются со временем), автор считает, что в целях экономии времени их следует сократить до абсолютного минимума (в идеальном случае — до списка с несложными диаграммами). Возможно, в вашей ситуации действуют другие ограничения, из-за которых объем документов придется увеличить, но небольшие и лаконичные исходные документы могут быть созданы за несколько сеансов «мозгового штурма» с руководителем. Такая процедура не только требует вклада со стороны всех участников, но и способствует единству взглядов и взаимопониманию в группе на первоначальной стадии проекта. Существует еще одно, возможно, более важное обстоятельство: группа возьмется за работу с большим энтузиазмом.

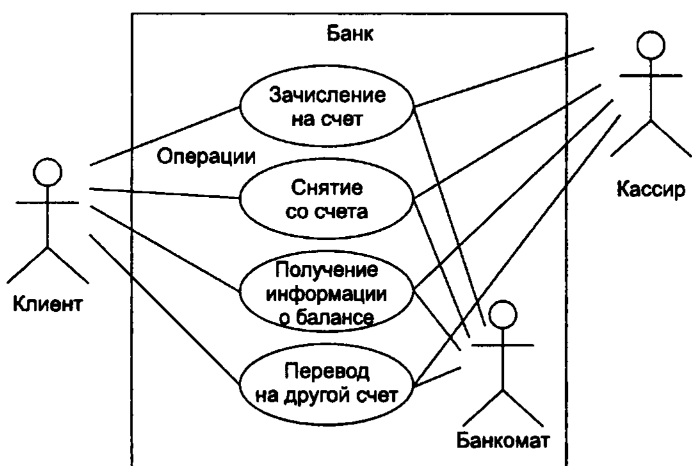
В этой фазе необходимо сосредоточиться на сути решаемой проблемы: определении того, что должна делать система. Самым ценным инструментом для этой цели является *набор функциональных требований*. Функциональные требования определяют ключевые особенности системы и дают представление о некоторых основных классах, которые будут использоваться в вашей программе. В сущности, набор функциональных требований состоит из содержательных ответов на некоторые вопросы[1].

- Кто будет использовать эту систему?
- Какие операции смогут выполнять операторы?
- Как оператор будет выполнять ту или иную операцию?
- Какие еще возможны варианты, если та же операция будет выполняться кем-то другим или у того же оператора появится другая цель (анализ вариантов)?
- Какие проблемы могут возникнуть при выполнении операции с системой (анализ исключений)?

[1] Спасибо за помощь Джеймсу Джаррету (James H. Jarrett).

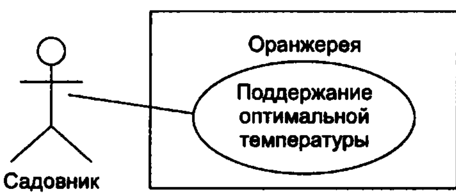
Например, если вы проектируете банкомат, то функциональные требования будут описывать действия банкомата в любой возможной ситуации. Каждая из таких ситуаций называется *сценарием*. Сценарий можно рассматривать как вопрос, начинающийся со слов: «Как поступит система, если...?» Например: «Как поступит банкомат, если клиент только что сделал вклад с 24-часовым оформлением, а без зачисления чека на счету не хватает средств для снятия желаемой суммы?»

Диаграммы функциональных требований должны быть как можно проще, чтобы вы раньше времени не увязли в подробностях реализации:



Фигурки на этом рисунке изображают «операторов» — обычно это люди или какие-либо независимые агенты (например, другая компьютерная система, как в случае с банкоматом). Прямоугольник изображает границы системы, а овалы — описания отдельных операций, выполняемых системой. Линии, соединяющие операторов с операциями, изображают взаимодействия.

В данном случае важна не конкретная реализация системы, а ее структура с точки зрения пользователя, поэтому даже для сложной системы функциональные требования не обязаны быть излишне сложными. Пример:



Функциональные требования определяются всеми взаимодействиями пользователя с системой. В результате выработки полного набора функциональных требований у вас появляется описание того, как должна работать система на самом общем уровне. Основным достоинством наборов функциональных требований является то, что они всегда ориентируются на суть происходящего и не дают отвлекаться на вопросы, не имеющие прямого отношения к решаемой задаче. Иначе говоря, если у вас появился полный набор функциональных требований, вы може-

те описать свою систему и перейти на следующую фазу. Возможно, вам не удастся правильно определить эти требования с первой попытки — ничего страшного. Все прояснится со временем, а любые попытки создания идеальной системной спецификации на этой стадии обычно заводят в тупик.

Если первые попытки завершатся неудачей, попробуйте воспользоваться методом грубого приближения: опишите систему в нескольких абзацах, а затем выделите в этом описании существительные и глаголы. Существительные могут определять операторов, контекст или место действия, а также объекты, с которыми выполняются операции. Глаголы обычно подразумевают взаимодействие между операторами и операциями и определяют этапы внутри функциональных требований. Помните, что глаголы и существительные часто соответствуют объектам и сообщениям на стадии проектирования (а также что функциональные требования описывают взаимодействие между подсистемами, а методика «существительных и глаголов» может использоваться только как вспомогательный инструмент «мозгового штурма» и ее результаты не преобразуются напрямую в функциональные требования).

Хотя весь этот процесс больше напоминает шаманство, нежели научный метод, на какой-то стадии важно провести начальное планирование. Вы уже имеете некоторое представление о системе и можете хотя бы приблизительно оценить, сколько времени займет ее разработка. При этом приходится учитывать множество факторов. Если срок окажется слишком длинным, ваша компания может отказаться от разработки системы и потратить свои ресурсы на что-нибудь более разумное. А может, ваш начальник уже решил, сколько времени должна занять работа над проектом, и попытается повлиять на вашу оценку. Но лучше с самого начала составить честное расписание и принять все ответственные решения. Известно много способов планирования (как и способов прогнозирования биржевых котировок), и все же лучше всего положиться в этом вопросе на собственный опыт и интуицию. Прикиньте, сколько времени потребует ваш проект, затем удвойте это число и прибавьте еще 10 %. Скорее всего, ваше внутреннее ощущение вас не подводит; вам действительно *удастся* получить нечто работоспособное за это время. Еще столько же времени уйдет на то, чтобы привести это «нечто» к более или менее приличному виду, а последние 10 % уйдут на шлифовку и всякие мелочи[1]. Однако все это еще нужно объяснить руководству, и несмотря на все стоны и попытки давления, которыми сопровождаются такие объяснения, дело обстоит именно так.

Фаза 2 — как делать

В этой фазе создается архитектура, описывающая классы и их взаимодействия. Прекрасная методика определения классов и их взаимодействий основана на использовании карточек CRC (Class-Responsibility-Collaboration — Класс-Ответственность-Взаимодействие»). Ценность данного инструмента отчасти обусловлена его предельной простотой. Вам понадобятся лишь пустые карточки 8 × 12 см. Каждая карточка представляет отдельный класс. На карточках записывается следующая информация.

---

[1] В последнее время мнение автора по этому вопросу изменилось. Удвоение и прибавление 10 % дает довольно точную оценку (при небольшом количестве непредсказуемых факторов), но чтобы уложиться в это время, необходимо прилежно работать. Если вы захотите сделать программу элегантной и получить удовольствие от своей работы, лучше умножать на 3 или на 4.

- Имя класса. Имя должно отражать сущность класса и быть понятным с первого взгляда.
- Ответственность класса — то, что он должен делать. Обычно задается простым перечислением имен функций классов (поскольку при хорошем проектировании эти имена должны быть содержательными), однако может содержать и другую информацию. Если понадобится с чего-то начать, взгляните на проблему с точки зрения ленивого программиста: какие объекты должны появиться, чтобы ваша проблема сама собой решилась?
- Взаимодействия класса — с какими другими классами должен взаимодействовать данный класс? Термин «взаимодействие» намеренно сделан предельно широким: под ним могут пониматься агрегирование или просто существование другого объекта, который выполняет работу по требованию объекта данного класса. Во взаимодействиях также должен учитываться круг пользователей класса. Например, если вы создаете класс Firecracker (фейерверк), какой класс будет им пользоваться: Chemist (химик) или Spectator (зритель)? Первый должен знать, какие реактивы требуются для составления смеси, а второй будет реагировать на яркие краски и узоры после взрыва.

Возможно, вам кажется, что карточки должны быть больше, чтобы на них поместилась вся нужная информация. Однако небольшие размеры выбраны намеренно — не только для того, чтобы классы были более компактными, но и чтобы предотвратить чрезмерную детализацию на слишком ранней стадии. Если все, что необходимо знать о классе, не помещается на маленькой карточке, значит, этот класс слишком сложен (вы либо увлеклись детализацией, либо класс нужно разбить на несколько классов). Идеальный класс должен быть понятен с первого взгляда. Карточки помогают выработать начальное представление об архитектуре системы, чтобы вы могли представить себе «общую картину», а уже потом занимались уточнением проекта.

Одно из самых выдающихся достоинств карточек проявляется в общении между людьми. Такое общение желательно проводить «в реальном времени» в группах и без компьютеров. Каждый участник отвечает за несколько классов, которые поначалу не имеют имен или иной информации. Группа имитирует работу системы, последовательно рассматривая разные сценарии, и решает, какие сообщения должны пересылаться различным объектам для выполнения текущего сценария. По ходу дела выявляются необходимые классы, их ответственности и взаимодействия, а карточки постепенно заполняются. После перебора всех сценариев у вас появится относительно полный набросок архитектуры системы.

Прежде чем автор начал пользоваться карточками, наибольшего успеха при разработке предварительной архитектуры удавалось добиваться рисованием объектов фломастером на белой доске перед группами, которые еще не создали ни одного ООП-проекта. Таким образом обсуждались взаимодействия между объектами, некоторые объекты стирались и заменялись другими. В сущности, автору просто приходилось рисовать все «карточки» на доске. Группа (которая хорошо знала, что должен делать проект) реально создавала архитектуру проекта, а не получала ее откуда-то сверху. Автору оставалось лишь направлять этот процесс: задавать нужные вопросы, выдвигать предположения и изменять их на основании мнения участников. Истинная красота процесса состояла в том, что группа училась объект-

но-ориентированному проектированию не на абстрактных примерах, а на примере той архитектуры, которая больше всего интересовала ее в тот момент.

После того как карточки будут заполнены, можно подумать о создании более формального описания архитектуры на языке UML. Использовать UML не обязательно, но это может быть полезно, особенно если вы хотите вывесить диаграмму на стену для всеобщего обозрения. Альтернативой UML может стать текстовое описание объектов и их интерфейсов или, в зависимости от вашего языка программирования, — программный код[1].

UML также обладает дополнительными средствами построения диаграмм для описания динамической модели системы. Диаграммы удобны при доминирующей роли переходов между состояниями системы или подсистемы (например, в системах управления). Возможно, вам также потребуется описать структуры данных в тех системах или подсистемах, в которых доминирующая роль отводится данным (например, при работе с базами данных).

Признаком завершения фазы 2 является законченное описание объектов и их интерфейсов... или по крайней мере их большей части — некоторые из них остаются неучтенными и выявляются только во время фазы 3. Это вполне нормальное явление. Важно лишь, чтобы все объекты были рано или поздно выявлены. Хорошо, когда они обнаруживаются на ранней стадии процесса, но ООП обладает достаточной структурной устойчивостью, так что более позднее обнаружение не принесет особого вреда. Процесс проектирования объектов можно разделить на пять стадий.

Пять стадий проектирования объектов

Проектирование объекта не ограничивается непосредственным написанием программы. В действительности процесс проектирования объектов состоит из нескольких последовательных стадий. Рассматривая его с этой точки зрения, вы не будете ждать, что вам удастся немедленно достичь совершенства. Достаточно осознавать, что понимание структуры объекта и его работы приходит со временем.

1. *Выявление объектов.* Эта стадия происходит во время первоначального анализа программы. Выявление объектов основано на анализе внешних факторов и ограничений, дублирования элементов в системе и наименьших концептуальных единиц. При использовании библиотек классов выбор некоторых объектов оказывается очевидным. Анализ сходства между классами, которое может быть положено в основу выбора базовых классов и иерархии наследования, может происходить как на этой стадии, так и позднее в процессе проектирования.

2. *Доработка объектов.* В процессе разработки объекта иногда выясняется, что объект должен содержать новые члены, отсутствующие в первоначальном проекте. Изменение внутреннего устройства объекта может потребовать поддержки со стороны других классов.

3. *Конструирование системы.* Как уже отмечалось, на этой стадии иногда выявляются дополнительные требования к объектам. По мере углубленного изучения задачи происходит совершенствование ваших объектов. Необходимость взаимодействия с другими объектами в системе может повлиять на

---

[1] В качестве языка для «исполняемого псевдокода» часто используется язык Python (www.python.org).

первоначальную структуру классов или потребовать введения новых классов. Например, может выясниться, что в системе нужны вспомогательные классы (вроде классов связанных списков), которые содержат минимум информации о состоянии (или вовсе не содержат ее) и просто облегчают работу других классов.

4. *Расширение системы.* По мере расширения возможностей системы иногда выясняется, что это расширение упирается в несовершенство предыдущей архитектуры. На основании полученной информации вы можете реструктурировать часть системы, возможно, добавить новые классы или иерархии.
5. *Повторное использование объектов.* На этой стадии класс проходит настоящее «испытание на прочность». Если кто-то попытается заново использовать его в принципиально новой ситуации, могут обнаружиться какие-то недостатки и недочеты. По мере адаптации класса к новым условиям все четче проявляются общие принципы работы класса, пока вы не получите тип, действительно подходящий для повторного использования. Но не стоит ожидать, что многие объекты в системной архитектуре должны иметь универсальный характер — вполне нормально, если большинство объектов ориентировано на конкретную систему. Типы, подходящие для повторного использования, встречаются реже и ориентируются на решение более общих задач.

Рекомендации по разработке объектов

Ниже приведены некоторые рекомендации, которыми следует руководствоваться при разработке классов.

- Создайте класс для решения конкретной задачи, обеспечьте его рост и развитие по мере решения других задач.
- Помните, выявление необходимых классов (и их интерфейсов) является важнейшей составляющей системного проектирования. Если у вас уже есть описания классов, работа значительно упрощается.
- Не пытайтесь узнать все с самого начала; изучайте проблему по мере надобности. От этого все равно никуда не деться.
- Переходите к программированию. Создайте рабочий прототип, который подтвердит или опровергнет первоначальную архитектуру. Не бойтесь, что у вас получится «спагетти-код» в стиле процедурного программирования, — классы обеспечивают логическое деление проблемы и помогают справиться с анархией и энтропией. Плохо спроектированные классы не нарушат работу хорошо спроектированных классов.
- Стремитесь к простоте. Небольшие понятные объекты с очевидными задачами лучше больших, запутанных интерфейсов. Принимая решения, руководствуйтесь принципом «бритвы Оккама»: рассмотрите все возможные варианты и выберите самый простой из них, потому что простые классы почти всегда оказываются лучшими. Начните с простых и компактных решений, и вы сможете расширить интерфейс класса, когда начнете его лучше понимать. С другой стороны, чем больше времени пройдет, тем сложнее будет удалять элементы из класса.

Фаза 3 — построение ядра

В этой фазе первоначальный набросок архитектуры преобразуется в компилируемый исполняемый код, который можно протестировать. Но самое главное, что прототип должен либо подтвердить, либо опровергнуть исходную архитектуру. Как будет показано при описании фазы 4, этот процесс выполняется не за один раз, а представляет собой последовательность этапов интерактивного построения системы.

Ваша задача — выявить ядро системной архитектуры, реализация которого абсолютно необходима для построения работоспособной системы, какой бы неполной ни была эта система в первом приближении. Фактически вы строите каркас, который будет наращиваться при последующих итерациях. Также при этом выполняется первая попытка системной интеграции и первое тестирование, а заказчик получает представление о том, как будет выглядеть его система и как идет работа. В идеальном случае при этом выявляются важнейшие риски, связанные с разработкой системы. Возможно, также обнаружатся некоторые изменения и усовершенствования в исходной архитектуре, о которых вы бы не узнали без реализации системы.

Одной из составляющих разработки системы является сравнение промежуточного результата с анализом требований и системными спецификациями (в той форме, в которой они существуют). Убедитесь в том, что результаты тестирования соответствуют всем требованиям и сценариям. Когда ядро системы заработает стабильно, можно двигаться дальше и добавлять новые возможности.

Фаза 4 — итеративный перебор сценариев

Получив работоспособное ядро, вы начинаете добавлять к нему новые возможности. Каждая группа возможностей может рассматриваться как мини-проект, а ее реализация занимает в цикле разработки относительно короткий период, который называется *итерацией*.

Какова продолжительность итераций? В идеале каждая итерация продолжается от одной до трех недель (срок зависит от языка реализации). В конце этого периода вы получаете интегрированную протестированную систему, которая обладает большими возможностями, чем раньше. Но особенно интересен тот факт, что итерации формируются на основе отдельных функциональных требований. Каждое функциональное требование представляет собой совокупность связанных функциональных возможностей, которые встраиваются в систему как единое целое во время одной итерации. Это не только дает лучшее представление о практических масштабах одной итерации, но и дополнительно подкрепляет саму идею функциональных требований, которые не теряют смысла после анализа и проектирования, а становятся фундаментальными единицами в процессе разработки программного обеспечения.

Итерация прекращается либо при достижении поставленной цели, либо в том случае, если на данном этапе заказчик удовлетворен текущей версией. Поскольку процесс разработки имеет итеративную природу, вместо единой конечной точки существует много возможностей поставки готового продукта; скажем, продукты с открытыми текстами создаются исключительно в итеративных условиях с хорошей обратной связью, чем и объясняется их успех.

Итеративный процесс разработки полезен по многим причинам. Итеративность позволяет обнаружить и решить критически важные проблемы на более ранней стадии; у заказчика появляется возможность откорректировать начальную постановку задачи; работа программиста становится более творческой, а сам проект лучше управляется. Не стоит забывать и о таком важном факторе, как обратная связь с заказчиком, который точно представляет себе текущее состояние продукта. В результате сокращается или вовсе исчезает необходимость в утомительных совещаниях, а разработчик получает большую поддержку и уверенность со стороны заказчика.

Фаза 5 — эволюция

Эта стадия цикла разработки традиционно называлась «сопровождением». Под этим универсальным термином может пониматься все что угодно: от «заставить систему работать так, как было задумано с самого начала», до «включить новые возможности, о которых забыл упомянуть заказчик» и более традиционных «исправить найденные ошибки» или «добавить новые возможности в случае необходимости». Термин «сопровождение» отягощен таким количеством ошибочных толкований, что теперь он уже воспринимается как лукавство. Он наводит на мысль, что вы создали безупречную программу, так что дальше остается время от времени менять запчасти, смазывать ее и вытирать насухо, чтобы не ржавела. Возможно, происходящее правильнее было бы назвать другим термином.

Автор предпочитает термин «эволюция». Он означает следующее: «С первого раза идеально все равно не получится, поэтому мы предусматриваем возможность для дополнительного анализа, возврата и внесения изменений». По мере изучения и более глубокого понимания проблемы иногда приходится вносить множество изменений. Элегантность, достигнутая в результате доработки программы, окупается как в краткосрочном, так и в долгосрочном плане. Эволюция обеспечивает тот промежуток времени, когда программа превращается из хорошей в замечательную, а проблемы, которые оставались непонятными при первой попытке, внезапно проясняются. Тогда же классы могут превратиться из разовых разработок в ресурсы многократного использования.

Под «доработкой» здесь понимается не только то, что программа должна начать функционировать в соответствии со всеми требованиями и сценариями. Также это означает, что внутренняя структура программы выглядит разумной с вашей точки зрения, программа не содержит неуклюжего синтаксиса, слишком крупных объектов или неловко пристегнутых «заплаток». Кроме того, вы должны быть уверены в том, что структура программы переживет все неизбежные изменения во время ее жизненного цикла и что эти изменения можно будет вносить быстро и просто. Необходимо не только понимать, что именно вы строите, но и в каком направлении будет развиваться программа (то, что автор называет *вектором изменений*). К счастью, объектно-ориентированные языки особенно хорошо подходят для повторных модификаций такого рода — границы объектов предохраняют структуру программы от распада. Благодаря им изменения, которые бы показались радикальными в процедурных программах, вносятся без распространения «ударной волны» по всей программе. В сущности, поддержка эволюции оказывается едва ли не самым важным преимуществом ООП.

В процессе эволюции вы создаете нечто, хотя бы отдаленно напоминающее итоговую цель, а затем с учетом ваших потребностей смотрите, где возникли расхождения. Далее можно вернуться и исправить ошибку переработкой архитектуры и повторной реализацией тех частей программы, которые работают неправильно[1]. Возможно, чтобы добраться до правильного решения, вам придется решать задачу многократно (или разбираться с ее отдельным аспектом). При этом особенно полезны типовые решения, или *эталоны*, описанные во втором томе книги.

Эволюция имеет место и в другом случае — когда вы строите систему, убеждаетесь, что она удовлетворяет начальным требованиям, а потом вдруг выясняется, что это совсем не то. Когда система уже работает, вы понимаете, что на самом деле она должна решать другую проблему. Если вы не исключаете подобную возможность, постарайтесь как можно быстрее построить первую версию и выяснить, соответствует ли она вашим ожиданиям.

Главное, о чем необходимо помнить, — что по умолчанию (а скорее, даже по определению) модификация класса не должна нарушать работу его базовых и производных классов. Не бойтесь модификации (особенно если у вас имеется встроенный набор модульных тестов для проверки их правильности). Модификации не всегда нарушают работу программы, а любые изменения результата будут ограничиваться производными классами и/или отдельными классами, взаимодействующими с измененным классом.

О пользе планирования

Никому не придет в голову строить жилой дом без множества тщательно проработанных планов. Если вы строите сарай или собачью будку, процесс планирования будет не столь тщательным, но, скорее всего, вы все же сделаете хотя бы простейшие наброски и будете руководствоваться ими. Программирование часто бросалось в крайности. В течение долгого времени разработки вообще не структурировались, но это стало приводить к провалам крупных проектов. В ответ появились новые методологии, невероятно детальные и структурированные, ориентированные прежде всего на крупные проекты. Эти методологии были слишком жуткими — казалось, программисту придется тратить все время на написание документации, а на программирование его уже не останется (кстати, на практике так часто и выходило). Все это наводит на мысль о «золотой середине» — подвижной шкале. Используйте тот способ, который лучше соответствует вашим потребностям (и вашим личным предпочтениям). Но каким бы расплывчатым ни был ваш план, помните, что проекты с *хоть каким-нибудь* планом идут гораздо лучше проектов, полностью лишенных всякого плана. Также не стоит забывать, что по статистике свыше 50 % проектов завершаются неудачей (а по некоторым данным — до 70 %)!

Если следовать некоторому плану (желательно простому и понятному) и заранее проработать общую структуру программы, работа пойдет легче, чем если вы просто возьметесь за программирование. Более того, ваша работа будет гораздо

[1] Эта методика напоминает «ускоренную разработку прототипа», когда вы наспех сооружаете простенькую модель для изучения системы, а затем выкидываете прототип и строите ее правильно. Главная проблема «ускоренной разработки прототипа» заключалась в том, что программисты его не выбрасывали, а использовали как базу для построения окончательной версии. В сочетании с недостаточной структурированностью процедурного программирования это часто приводило к созданию хаотичных систем, дорогих в сопровождении.

интереснее. Судя по опыту автора, разработка элегантного решения приносит необычайное удовольствие; такое программирование ближе к искусству, чем к технологии. Помните, что элегантность всегда окупается. Элегантное программирование упрощает не только разработку и отладку, но также чтение и сопровождение программы, а с этим связана прямая финансовая выгода.

Экстремальное программирование

Автор начал изучать методы анализа и проектирования еще в аспирантуре. Из всего, с чем удалось познакомиться, концепция экстремального программирования (eXtreme Programming, XP) была наиболее радикальной и восхитительной. Ее описание можно найти в книге Кента Бека (Kent Beck) «Extreme Programming Explained»[1] и на сайте www.xprogramming.com.

Экстремальное программирование — это одновременно и философия программирования, и набор рекомендаций по ее воплощению в жизнь. Некоторые из этих рекомендаций встречаются в других методологиях последнего времени, но, на взгляд автора, самыми важными и характерными положениями экстремального программирования являются тезисы «начинайте с написания тестов» и «программируйте парами». Хотя Бек настоятельно рекомендует взять на вооружение весь процесс, он указывает, что даже принятие этих двух правил заметно улучшит эффективность вашего труда и надежность программ.

Начинайте с написания тестов

Тестирование традиционно откладывалось до завершающей стадии проекта, когда «все уже работает, но проверить не помешает». Оно рассматривалась как низкоприоритетная задача, а к специалистам по тестированию часто относились пренебрежительно и даже содержали их в каком-нибудь чулане отдельно от «настоящих программистов». Тестеры отвечали окружающему миру взаимностью, носили черное и злорадно хихикали, когда им удавалось что-нибудь сломать (честно говоря, автор и сам ощущал нечто подобное, когда ломал компиляторы C++).

В экстремальном программировании концепция тестирования полностью пересматривается, и ей уделяется такое же (а то и более важное) значение по сравнению с программированием. Более того, тесты пишутся *раньше* тестируемого кода и остаются с ним навечно. Тесты должны успешно выполняться при каждой интеграции проекта (что нередко означает «чаще одного раза в день»).

Написание тестов в начале проекта имеет два исключительно важных последствия.

Во-первых, оно заставляет программиста четко определить интерфейс класса. Автор часто предлагает в процессе проектирования системы «представить себе идеальный класс для решения некоторой проблемы». Стратегия тестирования в XP идет еще дальше — вы должны точно указать, как выглядит класс и каково его поведение с точки зрения пользователя этого класса. Все предельно конкретно, ни малейшей неопределенности. Конечно, можно описать класс в прозе или нарисо-

[1] См. *Бек К., Фаулер М.* Экстремальное программирование: планирование: Библиотека программиста». СПб.: Питер, 2003.— *Примеч. ред.*

вать множество диаграмм, описывающих поведение класса и его структуру, но ничто так не приближено к реальности, как набор тестов. Описания и диаграммы — не более чем пожелания, а тест описывает обязательный контракт между компилятором и выполняемой программой. Трудно представить себе более конкретную спецификацию класса, чем тест.

При создании теста вам придется досконально продумать структуру класса, причем иногда при этом обнаруживаются возможности, упущенные во время мысленных экспериментов с UML-диаграммами, карточками, сценариями и т. д.

Второе важное последствие от опережающего написания тестов объясняется тем, что тесты все равно выполняются при каждом получении программы, причем половину тестирования фактически проводит компилятор. Если взглянуть на эволюцию языков программирования с этой точки зрения, вы увидите, что все подлинные усовершенствования в технологии так или иначе связаны с проверкой и тестированием. В ассемблере проверялся только синтаксис, а в C появились некоторые семантические ограничения, которые предотвращали ошибки определенных типов. В языках ООП установлены еще более жесткие семантические ограничения, которые, если хорошо подумать, тоже являются формами проверки. «Правильно ли используется этот тип? Правильно ли вызывается эта функция?» — все это разные типы проверок, осуществляемых компилятором или системой времени выполнения. Результаты включения этих проверок в язык уже известны: программисты создают более сложные системы и запускают их в работу с гораздо меньшими затратами времени и усилий. Раньше автор не понимал, из-за чего это происходит, но теперь стало ясно, что дело в тестах: если что-то пойдет не так, то встроенные тесты сообщат о наличии проблемы и ее источниках.

Однако на большее встроенные средства тестирования, поддерживаемые на языковом уровне, не способны. На каком-то этапе *вам* придется взяться за дело и добавить остальные тесты, которые завершали бы тестовый комплекс (в сочетании с тестами компилятора и системы времени выполнения) для проверки всей программы. Но если уж компилятор постоянно помогает вам в поиске ошибок, то почему бы не пользоваться тестами с самого начала? Вот почему лучше сразу написать тесты и автоматически включать их во все промежуточные версии системы. Ваши тесты станут логическим расширением проверочных средств, предоставляемых языком.

Используя все более и более мощные языки программирования, автор обнаружил, что они поощряют к смелым экспериментам, поскольку при поиске ошибок сам язык избавляет от лишних трат времени. Стратегия тестирования экстремального программирования делает то же самое с целым проектом. Вы уверены в том, что тесты всегда помогут выявить все проблемы (и регулярно добавляете в проект новые тесты, когда в голову приходит хорошая идея), поэтому в проект можно вносить серьезные изменения, не беспокоясь о том, что он придет в состояние полного хаоса.

Парное программирование

Парное программирование противоречит закоренелому индивидуализму, который нам прививают всю жизнь, начиная со школы (где все успехи и неудачи засчитываются лично нам, а взаимодействие с соседями считается жульничеством) и кончая средствами массовой информации, особенно голливудскими фильмами, в которых герой-одиночка обычно противостоит безликой массе

злодеев[1]. Программисты также считаются воплощением индивидуализма — «ковбоями от программирования», как любит выражаться Ларри Константайн (Larry Constantine). И все же экстремальное программирование, которое ломает сложившиеся стереотипы, утверждает, что программу должны писать двое работников за одним рабочим местом. Более того, рабочие места в группах не должны разделяться перегородками, которые так любимы дизайнерами наших рабочих помещений. Бек говорит, что при переходе на экстремальное программирование нужно прежде всего взять гаечные ключи и разобрать все, что мешает общению[2] (для этого также понадобится начальник, который усмирит гнев хозяйственного отдела).

Главная ценность парного программирования заключается в том, что один программист пишет программу, а второй думает. Этот второй представляет себе общую картину, помня не только о непосредственно решаемой задаче, но и о правилах XP. Когда работают двое, вряд ли кто-нибудь заявит: «Я не хочу писать тесты с самого начала». Если программист окажется в тупике, он может поменяться местами с напарником. А если в тупике окажутся оба, возможно, их разговоры услышит кто-нибудь другой, кто сможет помочь. Парная работа способствует нормальному продвижению проекта, но не менее важно и то, что программирование становится более социальным и интересным занятием.

Автор начал использовать парное программирование на практических занятиях своих учебных курсов. На взгляд автора, оно значительно улучшает восприятие материала учащимися.

Причины успеха C++

Одна из причин успеха C++ заключается в том, что проектировщики языка стремились не просто превратить C в объектно-ориентированный язык (хотя начиналось именно с этого), но также решить ряд других проблем, с которыми и в наше время сталкиваются разработчики, особенно имеющие большой опыт программирования на C.

Традиционно языки ООП требовали, чтобы программист забыл все, что он знает, и начал «с нуля» с новыми концепциями и новым синтаксисом. Это аргументировалось тем, что в долгосрочной перспективе лучше полностью отказаться от старого багажа, унаследованного из процедурных языков. Возможно, в долгосрочной перспективе это действительно так, но с практической точки зрения в этом багаже было немало ценного. Причем самым ценным элементом была даже не существующая база программного кода (которую при наличии хорошего инструментария можно было конвертировать), а скорее *база мышления*. Если действующему программисту C для перехода на новый язык приходилось отказываться от всего, что он знает о C, производительность его труда резко снижалась на многие месяцы, пока его ум не привыкал к новой парадигме. С другой стороны, если программист

---

[1] Наверное, этот пример больше рассчитан на американцев, хотя голливудские истории проникают повсюду.

[2] Включая (в первую очередь) систему местной громкой связи. Однажды автор работал в компании, которая требовала передавать по громкой связи все телефонные звонки всех руководителей, что основательно мешало нашей работе (однако начальство и представить себе не могло, как можно жить без громкой связи). Пришлось потихоньку резать провода, идущие к динамикам.

мог взять за основу существующие познания в C и расширять их, то при переходе в мир объектно-ориентированного программирования он продолжал эффективно работать. У каждого из нас есть собственная внутренняя модель программирования, и такие переходы достаточно хлопотны и без дополнительного бремени в виде новой языковой модели. Таким образом, успех C++ в основном объяснялся экономическими причинами: переход на ООП вообще требует затрат, но переход с C на C++ может обойдись дешевле[1].

Главной целью разработки C++ является повышение эффективности. Источников этого повышения много, но язык спроектирован так, чтобы по возможности помочь программисту, не отягощая его искусственными правилами или требованиями об использовании некоторого ограниченного набора средств. Язык C++ планировался как практичный язык, а главной целью его разработки была максимальная польза для программиста (по крайней мере, с точки зрения C).

Улучшенный язык C

Даже если вы продолжите писать C-образный код, переход на C++ сразу же принесет пользу, потому что C++ закрыл много «дыр» C, обеспечил более качественную проверку типов и анализ на стадии компиляции. Программист вынужден объявлять функции, чтобы компилятор мог проверить правильность их вызова. В C++ практически исчезла необходимость в использовании препроцессора для подстановки значений и макросов, что раньше приводило к трудноуловимым ошибкам. В C++ появились *ссылки*, обеспечивающие более удобную передачу адресов в аргументах и возвращаемых значениях функций. Механизм *перегрузки функций* существенно упростил работу с именами, поскольку теперь одно и то же имя может быть присвоено разным функциям. *Пространства имен* также улучшают контроль над использованием имен в программе. Существует еще множество мелких усовершенствований, которые делают C++ более надежным языком по сравнению с C.

Быстрое обучение

Главная проблема с изучением нового языка часто связана с производительностью труда программиста. Ни одна компания не может себе позволить вдруг лишиться эффективно работающего программиста только потому, что он изучает новый язык. C++ является расширением C, его синтаксис и модель программирования не будут абсолютно новыми для программиста. Благодаря этой особенности программист по-прежнему пишет полезные программы, используя новые возможности по мере их изучения. Возможно, это одна из главных составляющих успеха C++.

Кроме того, весь существующий код C может компилироваться в C++, но так как компилятор становится более придирчивым, при перекомпиляции в программах C часто обнаруживаются скрытые ошибки.

[1] Автор говорит «может», потому что из-за сложностей C++ переход на Java иногда обходится дешевле. Однако решение о выборе языка принимается на основе многих факторов, и в этой книге предполагается, что вы уже выбрали C++.

Эффективность

В некоторых ситуациях скоростью работы программы приходится жертвовать ради ускорения работы программиста. Например, финансовая модель обычно остается актуальной в течение ограниченного периода времени, поэтому ее быстрая реализация важнее быстрой работы программы. Однако большинство приложений требует определенной скорости работы, поэтому C++ всегда отдает предпочтение более эффективным решениям. А так как программисты С особенно внимательно следят за эффективностью, они не будут жаловаться, что новый язык неповоротлив и медлителен. В C++ предусмотрен целый ряд средств оптимизации на случай, если сгенерированный код окажется недостаточно эффективным.

В распоряжении программиста оказываются низкоуровневые средства С вместе с возможностью включения ассемблерного кода в программы C++. Статистика показывает, что быстродействие объектно-ориентированной программы C++ обычно находится в пределах ±10 % от быстродействия аналогичной программы С, а иногда гораздо ближе. С другой стороны, архитектура ООП-программы может быть гораздо эффективнее своего аналога в С.

Простота описания и понимания системы

Классы, спроектированные для решения проблемы, обычно выражают ее лучше обычного кода. Это означает, что при написании программы решение представляется в пространстве задачи («Замкнуть контакт»), а не в пространстве машины («Установить бит, который является признаком замыкания контакта»). Программист работает с концепциями более высокого уровня, а одна строка программы несет гораздо большую смысловую нагрузку.

Другое преимущество простоты выражения проявляется на стадии сопровождения, на которую (если верить статистике) уходит заметная часть расходов в жизненном цикле программы. Естественно, понятные программы проще сопровождать. Кроме того, при этом сокращаются затраты на создание и ведение документации.

Максимальная интеграция с библиотеками

Чтобы написать программу, быстрее всего воспользоваться уже готовым кодом, то есть библиотеками. Одной из главных целей, поставленных при проектировании C++, была простота работы с библиотеками. Для этого библиотеки трансформируются в новые типы данных (классы), поэтому подключение библиотеки означает добавление новых типов в язык. Компилятор C++ сам следит за использованием библиотеки, обеспечивает всю необходимую инициализацию и зачистку, а также проверяет правильность вызова функций, поэтому программист может сосредоточить внимание на том, что делает библиотека, а не на том, как с ней работать.

Поскольку пространства имен ограничивают область действия имен отдельными частями программы, свободное использование библиотек не создает конфликтов имен, характерных для С.

Шаблоны и повторное использование исходных текстов

Существует достаточно обширная категория типов, для повторного использования которых необходима модификация исходных текстов. *Шаблоны* C++ автома-

тически изменяют исходные тексты, что делает их исключительно важным инструментом для повторного использования библиотечного кода. Обобщенный класс, созданный на базе шаблона, легко адаптируется для другого типа-параметра. Шаблоны особенно хороши тем, что полностью скрывают от прикладного программиста все сложности, связанные с повторным использованием кода.

Обработка ошибок

Как известно, обработка ошибок в C порождала немало проблем. Нередко проблема просто игнорировалась и программист полагался «на авось». При построении большого и сложного проекта нет ничего хуже, чем обнаружить ошибку, которая находится неизвестно где. Механизм *обработки исключений* (кратко упомянутый в этом и подробно рассматриваемый во втором томе) гарантирует, что ошибка будет обнаружена и каким-то образом обработана программой.

Масштабное программирование

Во многих традиционных языках существуют встроенные ограничения на размеры и сложность программ. Например, BASIC прекрасно подходит для разработки быстрых решений некоторых классов задач, но если программа занимает больше нескольких страниц или выходит за рамки обычной области применения этого языка, программирование начинает напоминать плавание в непрерывно густеющей жидкости. У языка C тоже есть свои ограничения. Например, когда объем программы превышает 50 000 строк кода или около того, начинаются проблемы с конфликтами имен — фактически у вас просто кончаются удобные имена функций и переменных. Другая неприятная проблема связана с недоработками самого языка C — в большой программе иногда бывает очень трудно найти ошибку.

Не существует четких критериев, определяющих границы применимости языка, но даже если бы такие критерии и были, скорее всего, вы бы их игнорировали. Никто не скажет: «Моя программа на BASIC слишком велика — пожалуй, ее стоит переписать на C!» Вместо этого вы попытаетесь затолкнуть в нее еще несколько строк и добавить «всего одну» новую возможность. В конечном счете это обернется лишними затратами при сопровождении.

Язык C++ проектировался для *масштабного программирования*, при котором стираются грани между мелкими и крупными программами. Конечно, при написании утилиты сложности «Hello, World!» вам не придется использовать ООП, шаблоны, пространства имен и обработку исключений, но эти возможности присутствуют в языке и могут задействоваться при необходимости. С другой стороны, компилятор с одинаковым рвением выискивает ошибки как в больших, так и в мелких программах.

Стратегии перехода

Если вы твердо решили перейти на ООП, возникает следующий вопрос: «Как заставить начальство/коллег/другие отделы перейти на использование объектов?» Подумайте, как бы вы — независимый программист — перешли на изучение нового языка и новой парадигмы программирования. Вам ведь уже приходилось делать

это раньше, не правда ли? Все начинается с обучения и примеров; далее следует экспериментальный проект, который позволяет прочувствовать суть дела без лишних сложностей. Затем наступает черед «реального» проекта, который делает что-то полезное. На протяжении первых проектов вы продолжаете обучение — читаете книги, задаете вопросы знатокам и обмениваетесь полезной информацией с друзьями. Так выглядит рекомендуемая многими опытными программистами процедура перехода с C на C++. Конечно, при переходе целой организации добавится некоторая групповая динамика, но на каждом этапе стоит помнить, как бы эта задача решалась одним человеком.

Рекомендации

Ниже приводятся некоторые рекомендации по переходу на ООП и C++.

Обучение

Первым шагом должна стать некоторая форма обучения. Помните о капиталовложениях компании в существующий код C и постарайтесь избежать хаоса в то время, пока народ будет разбираться в тонкостях множественного наследования. Выберите небольшую группу для обучения; желательно, чтобы ее участники были любознательны, хорошо уживались друг с другом и могли оказывать взаимную помощь в ходе изучения C++.

Иногда предлагается другой подход — одновременное обучение всей компании сразу с обзорными курсами для руководства и занятиями по проектированию и программированию для разработчиков проектов. Подобные решения особенно хороши в небольших компаниях, желающих изменить свой профиль, или на уровне подразделений крупных компаний. Однако из-за возрастающей стоимости проекта в некоторых случаях сначала обучается небольшая группа персонала, которая создает опытный проект (возможно — с внешним наставником), а затем обучает остальных сотрудников.

Проекты с низкой степенью риска

Попробуйте для начала выполнить проект с низкой степенью риска, в котором ошибки не так критичны. Когда участники проекта получат определенный опыт, они смогут инициировать другие проекты или выполнять обязанности службы технической поддержки по ООП. Нельзя исключать, что первый проект провалится, поэтому он не должен быть жизненно важным для организации. Проекту достаточно быть простым, самодостаточным и поучительным; это означает, что разработанные в нем классы должны быть понятны для других программистов, которые будут следующими изучать C++.

Успешные решения

Прежде чем начинать работу «с нуля», стоит поискать примеры хорошего объектно-ориентированного проектирования. Вполне вероятно, что кто-то уже решил вашу задачу, но даже если полностью подходящего решения не найдется, возможно, вам удастся воспользоваться своими познаниями в области абстракции и адаптировать существующую архитектуру к своим потребностям. На этом принципе строится общая концепция *эталонов* (типовых решений), описанная во втором томе книги.

Готовые библиотеки классов

Основным экономическим стимулом для перехода на ООП является возможность простого использования готового кода в виде библиотек классов (и прежде всего стандартной библиотеки C++, подробно описанной во втором томе книги). Кратчайший цикл разработки приложения достигается тогда, когда программисту не приходится писать ничего, кроме функции main(), создающей и использующей объекты из готовых библиотек. Тем не менее многие новички не понимают этого, не знают о существовании готовых библиотек или от избытка энтузиазма готовы заново переписать уже существующие классы. Ваш переход на ООП и C++ будет еще эффективнее, если вы на ранней стадии поищете готовый код и задействуете его в своем проекте.

Существующие программы на C++

Компиляция кода C компилятором C++ обычно приносит пользу (причем порой огромную), так как она помогает обнаружить проблемы старых программ. Тем не менее обычно не стоит брать готовые, работоспособные программы и переписывать их на C++ (а если их необходимо привести к объектному виду, можно «завернуть» готовый код C в классы C++). Конечно, это может принести определенную выгоду, особенно если код предназначен для повторного использования, но вряд стоит ждать радикального повышения быстродействия в готовых проектах. Достоинства C++ и ООП наиболее очевидно проявляются при воплощении нового проекта из концепции в реальность.

Организационные трудности

Если вы работаете на руководящей должности, ваша задача — предоставить все необходимые ресурсы, преодолеть все препятствия на пути к успеху и вообще создать наиболее продуктивную и приятную рабочую среду, чтобы вашим подчиненным было легче совершать те чудеса, которых от вас вечно требует начальство. Переход на C++ влияет на все три категории задач, и все было бы замечательно, если бы за него не приходилось платить. Хотя для группы программистов C переход на C++ может обойтись дешевле перехода на другие объектно-ориентированные языки[1], он все же потребует определенных затрат. Существует целый ряд препятствий, о которых необходимо знать, прежде чем вы решитесь перейти на C++ и перевести на него вашу организацию.

Начальные издержки

Начальные издержки перехода на C++ не ограничиваются стоимостью компилятора C++ (кстати, один из лучших компиляторов — компилятор GNU C++ — распространяется бесплатно). Для сведения к минимуму средне- и долгосрочных затрат необходимо потратиться на подготовку персонала (а возможно, и на наем наставников для первого проекта), а также на поиск и приобретение библиотек классов, используемых в решении задачи (вместо самостоятельной разработки этих классов). Это непосредственные затраты, которые должны учитываться в любом реалистичном предложении. Кроме того, существуют скрытые издержки в виде снижения производительности во время изучения нового языка, а возможно,

[1] Также стоит упомянуть язык Java, который в последнее время стал весьма распространенным.

и новой среды программирования. Конечно, правильная организация обучения способна свести эти затраты к минимуму, но участникам группы все равно придется тратить силы на усвоение новой технологии. В это время они будут совершать больше ошибок (что только полезно, поскольку осознание своих ошибок — самый быстрый способ обучения) и работать менее эффективно. Но даже на стадии изучения C++ в определенном классе задач и при правильном выборе классов и среды разработки (даже с учетом большего количества ошибок и меньшего количества строк программы в день) может быть достигнута более высокая производительность, чем при работе на C.

Проблемы эффективности

Стандартный вопрос: «Разве от применения ООП мои программы не становятся больше и не начинают работать медленнее?». Ответ: «Когда как». Большинство объектно-ориентированных языков проектировалось с расчетом на эксперименты и ускоренную разработку прототипов, нежели на быстроту и эффективность. Соответственно, они практически всегда приводили к увеличению объема и снижению быстродействия программ. Однако язык C++ проектировался для реального программирования. Стремясь побыстрее создать прототип, вы как можно скорее связываете готовые компоненты, не обращая внимания на эффективность. Если при этом используются библиотеки сторонних производителей, они обычно уже оптимизированы своими разработчиками; впрочем, в режиме ускоренной разработки это обычно несущественно. Если в результате получается система, которая делает то, что нужно, и достаточно быстро, работа закончена. А если нет, приходится браться за профайлер и начинать оптимизацию. Поиск начинается с тех усовершенствований, которые могут быть реализованы встроенными средствами C++. Если это не приносит успеха, вы переходите к оптимизации внутренней реализации, чтобы сохранить в неприкосновенности код работы с классами. Только если это не помогает, приходится вносить изменения в архитектуру системы. Тот факт, что эффективность играет критически важную роль в архитектуре системы, свидетельствует о необходимости ее включения в исходные критерии проектирования. Средства быстрой разработки позволяют вовремя это понять.

Как уже упоминалось, различия в быстродействии и размерах программ C и C++ чаще всего характеризуются величиной ±10 %, хотя обычно они гораздо ближе друг к другу. В отдельных случаях применение C++ вместо C даже обеспечивает значительный выигрыш в быстродействии и размерах программы, потому что архитектура программ C++ может существенно отличаться от архитектуры программ C.

Результаты сравнения программ C и C++ по быстродействию и размерам не являются абсолютно достоверными и, скорее всего, таковыми останутся. Сколько бы народ ни предлагал реализовать один и тот же проект на C и C++, вряд ли какая-нибудь компания захочет тратить деньги на подобные эксперименты — разве что очень крупная и заинтересованная в таких исследованиях (но даже в этом случае деньги можно потратить лучше). Почти все программисты, переходящие на C++ или другой язык ООП с C (или другого процедурного языка), говорят, что их труд стал гораздо эффективнее, — и это самый убедительный аргумент, который только можно представить.

Распространенные ошибки проектирования

В первое время после перехода на ООП и C++ программисты обычно совершают одни и те же стандартные ошибки. Это часто происходит из-за нехватки общения с экспертами в ходе проектирования и реализации первых проектов, потому что свои эксперты в компании еще не появились, а намерение пригласить платных консультантов может столкнуться с возражениями. Многие новички воображают, что они уже разобрались в ООП, и идут в неверном направлении. То, что очевидно опытному программисту, может вызвать долгую внутреннюю борьбу у новичка. Чтобы обойтись без лишних переживаний, для обучения персонала лучше пригласить эксперта.

С другой стороны, простота, с которой совершаются эти ошибки проектирования, указывает на главный недостаток C++ (который одновременно является главным достоинством): его совместимость с языком С. Чтобы можно было компилировать код С, разработчикам C++ пришлось пойти на некоторые компромиссы, что привело к появлению неких «темных мест» — вполне реальных и требующих немалых усилий при изучении языка. В этой книге автор постарался раскрыть большинство ловушек, часто встречающихся при программировании на C++. Всегда помните, что система страховки от ошибок в C++ не идеальна.

Итоги

В этой главе автор постарался дать представление об общих перспективах объектно-ориентированного программирования и C++. В частности, были описаны отличительные особенности ООП (и C++ в частности), методологии ООП, а также некоторые проблемы, встречающиеся при переходе компании на ООП и C++.

Впрочем, ООП и C++ — не панацея. Очень важно правильно оценить ваши потребности и решить, удовлетворит ли их C++ оптимальным образом или лучше выбрать другую систему программирования (возможно, ту, которую вы используете в данный момент). Если вы знаете, что в обозримом будущем вам придется иметь дело с нестандартными задачами и специфическими ограничениями, на которые язык C++ не рассчитан, вам придется самостоятельно изучить возможные альтернативы[1]. Но даже если в итоге ваш выбор остановится на C++, вы, по крайней мере, будете хорошо знать возможные варианты и четко понимать, что привело вас к этому решению.

Вы знаете, как выглядит процедурная программа: она состоит из определений данных и вызовов функций. Чтобы понять смысл такой программы и построить ее мысленную модель, придется основательно поработать, проанализировать вызовы функций и низкоуровневые операции. Из-за этого при проектировании процедурных программ необходимы промежуточные представления — сами по себе программы получаются слишком запутанными, потому что выразительные средства процедурных языков больше ориентированы на компьютер, нежели на решаемую задачу.

---

[1] В частности, автор рекомендует познакомиться с языками Java (http://java.sun.com) и Python (http://www.Python.org).

Поскольку C++ дополняет язык C многими новыми концепциями, было бы естественно предположить, что функция main() в программе C++ гораздо сложнее своего аналога в эквивалентной программе C. Однако вас ждет приятный сюрприз: хорошо написанная программа C++ обычно гораздо проще и понятнее эквивалентной программы C. Она содержит определения объектов, которые соответствуют концепциям из пространства задачи (вместо аспектов компьютерного представления), и сообщений, пересылаемых объектам для выполнения операций в этом пространстве. Одна из прелестей объектно-ориентированного программирования заключается в том, что хорошо спроектированная программа понятна при простом чтении. Кроме того, программы C++ обычно содержат меньше кода, потому что многие проблемы решаются при помощи готового библиотечного кода.

Создание и использование объектов

Элементы синтаксиса и программные конструкции C++, описанные в этой главе, позволяют написать и запустить несложную объектно-ориентированную программу. В следующих главах базовый синтаксис С и C++ рассматривается более подробно.

С помощью этой главы читатель получит базовое представление о том, как использовать объекты при программировании на C++, а также узнает, почему у языка C++ так много поклонников. Этот материал поможет осилить главу 3, которая выглядит несколько утомительной из-за многочисленных подробностей о языке С.

Главное отличие C++ от классических процедурных языков воплощено в пользовательских типах данных, или *классах*. Класс представляет собой новый тип данных, определяемый вами или кем-то другим для решения конкретной разновидности задач. Созданный класс может использоваться кем угодно, причем для этого даже не нужно точно знать, как он работает и как вообще создаются классы. В этой главе классы рассматриваются как встроенные типы данных, доступные для использования в программах.

Классы, созданные другими программистами, обычно собираются в библиотеки. В настоящей главе применяются некоторые библиотеки классов, присутствующие во всех реализациях C++. Особенно заметная роль принадлежит стандартной библиотеке потоков ввода-вывода, которая (среди прочего) позволяет читать данные из файлов и с клавиатуры, а также выводить их в файлы и на экран. Заодно вы познакомитесь с очень удобным классом string и контейнером vector из стандартной библиотеки C++. К концу главы вы убедитесь в том, что работать с готовой библиотекой классов совсем несложно.

Но прежде чем браться за первую программу, необходимо разобраться в том, какие инструменты требуются для разработки приложений.

Процесс трансляции

Любой язык программирования транслируется (то есть переводится) из формы, удобной для человеческого восприятия (*исходного текста*), в форму, воспринимаемую компьютером (*машинный код*).

Традиционно принято делить трансляторы на две категории: *интерпретаторы* и *компиляторы*.

Интерпретаторы

Интерпретатор транслирует исходный текст в операции, которые могут состоять из нескольких групп машинных команд, и немедленно выполняет эти операции. Одним из популярных интерпретируемых языков был BASIC. Традиционные интерпретаторы BASIC транслируют и выполняют программу по строкам, причем сразу же после выполнения интерпретатор «забывает» о выполненной работе. Такие языки работают очень медленно, поскольку они должны заново транслировать уже обработанный код. Код BASIC также может компилироваться для повышения скорости. Более современные интерпретаторы, например интерпретатор языка Python, транслируют всю программу на промежуточный язык, который затем выполняется гораздо более быстрым интерпретатором[1].

Интерпретаторы обладают многими достоинствами. Преобразование исходного текста в исполняемый код происходит практически мгновенно, а связь с исходным текстом сохраняется, поэтому при возникновении ошибки интерпретатор может выдать гораздо более полную информацию. К достоинствам интерпретаторов также обычно относят интерактивность и ускорение разработки (но не выполнения) программ.

Серьезные недостатки интерпретируемых языков обычно проявляются в крупных проектах (похоже, к Python это не относится). Интерпретатор (или его усеченный вариант) всегда должен находиться в памяти для выполнения кода, и быстродействие даже самого скоростного интерпретатора может оказаться неприемлемым. Многие интерпретаторы требуют, чтобы весь исходный текст программы загружался в интерпретатор полностью. Это требование не только ограничивает максимальный объем программы, но и затрудняет ее диагностику.

Компиляторы

Компилятор транслирует исходный текст программы на язык ассемблера или в машинные команды. Конечным результатом компиляции является файл (или файлы), содержащий машинный код. Компиляция — весьма сложный процесс, который обычно состоит из нескольких этапов, а переход от исходного текста к исполняемому коду при компиляции обычно занимает гораздо больше времени, чем при интерпретации.

Чем выше мастерство автора компилятора, тем меньше места занимают сгенерированные программы и тем быстрее они работают. Хотя размер исполняемого

---

[1] Граница между компиляторами и интерпретаторами в последнее время утратила четкость, особенно в языке Python, в котором богатство возможностей и мощь компилируемых языков сочетаются с гибкостью интерпретируемых языков.

файла и быстродействие обычно считаются главными доводами в пользу компиляторов, во многих случаях эти факторы не столь важны. Некоторые языки (такие, как С) поддерживают независимую компиляцию компонентов программы. Компоненты в конечном счете объединяются в *исполняемую программу* с помощью специальной программы — *компоновщика*. Этот процесс называется *раздельной компиляцией*.

Раздельная компиляция обладает многими преимуществами. Даже если программа как единое целое выходит за рамки ограничений компилятора или среды компиляции, ее можно откомпилировать по частям. Построение и тестирование программ также проводится по частям. Когда отдельный компонент начинает успешно работать, его можно сохранить и интерпретировать как готовый «строительный блок». Наборы протестированных, заведомо работоспособных компонентов объединяются в *библиотеки*, которые могут использоваться другими программистами. В процессе создания каждого компонента программист избавлен от необходимости учитывать сложность других компонентов. Все эти возможности позволяют создавать большие программы[1].

Со временем отладочные возможности компиляторов значительно усовершенствовались. Ранние компиляторы только генерировали машинный код, а программист вставлял в программу команды отладочной печати, чтобы разобраться в происходящем. Такой подход обычно малоэффективен. Современные компиляторы способны вставлять в исполняемую программу информацию из исходного текста. Мощные отладчики используют эту информацию, позволяя программисту отслеживать работу программы на уровне исходных текстов.

В некоторых компиляторах проблема быстродействия решается за счет *компиляции в памяти*. Большинство компиляторов работает с файлами, содержимое которых читается и записывается на каждом этапе процесса компиляции. При компиляции в памяти программа-компилятор находится в ОЗУ компьютера. Для небольших программ такой подход по оперативности получения исполняемого кода не уступает использованию интерпретаторов.

Процесс компиляции

Чтобы программировать на С и С++, необходимо хорошо понимать основные этапы и инструменты, задействованные в процессе компиляции. В некоторых языках (в том числе С и С++) компиляция начинается с обработки исходного текста *препроцессором*. Препроцессор — простая программа, которая заменяет некоторые комбинации символов в исходном тексте другими комбинациями, определенными программистом (при помощи директив препроцессора). Директивы препроцессора уменьшают объем вводимого текста и делают программу более наглядной. (Как будет показано далее, язык С++ разработан так, чтобы свести применение препроцессора к минимуму, так как последний иногда является источником нетривиальных ошибок). Программа, обработанная препроцессором, часто записывается в промежуточный файл.

Компилятор обычно выполняет свою работу за два прохода. На первом проходе производится *лексический разбор* программы, обработанной препроцессором.

[1] Язык Python и на этот раз оказывается исключением из правила, поскольку в нем также поддерживается раздельная компиляция.

Компилятор разбивает исходный текст на составляющие и организует их в структуру, называемую *деревом*. Так, в выражении A + B элементы A, + и B представляются листовыми узлами на дереве разбора.

Между первым и вторым проходами иногда используется *глобальный оптимизатор* для получения более компактного и быстрого кода.

На втором проходе *генератор кода* анализирует дерево разбора и генерирует ассемблерный или машинный код для узлов дерева. Если генератор кода создает ассемблерный код, в системе должен быть запущен ассемблер. Конечным результатом в обоих случаях является объектный модуль (файл, которому обычно присваивается расширение .o или .obj). Иногда на втором проходе используется *локальный оптимизатор*, который находит фрагменты кода с лишними ассемблерными командами.

Использование термина «объектный» для описания фрагментов машинного кода является нежелательным «пережитком прошлого». Он появился еще до того, как объектно-ориентированное программирование получило широкое распространение. Термин «объект» в контексте компиляции был синонимом «цели», тогда как в объектно-ориентированном программировании он означает «логически обособленную сущность».

Компоновщик объединяет объектные модули в исполняемую программу, которая может загружаться и запускаться операционной системой. Если функции одного объектного модуля ссылаются на функции других объектных модулей, компоновщик разрешает эти ссылки; он также проверяет, что все внешние функции и данные, объявленные на стадии компиляции, действительно существуют. Кроме того, компоновщик включает в программу специальный объектный модуль для выполнения начальной инициализации.

Для разрешения ссылок компоновщик производит поиск по специальным файлам — так называемым *библиотекам*. Библиотека представляет собой набор объектных модулей, объединенных в один файл. Создание и сопровождение библиотек осуществляется при помощи специальных программ — *библиотекарей*.

Статическая проверка типов

На первом проходе компилятор выполняет *проверку типов*. В ходе проверки он убеждается в правильности типов аргументов функций, что позволяет предотвратить многие ошибки программирования. Поскольку проверка выполняется на стадии компиляции, а не в процессе выполнения, она называется *статической проверкой типов*.

В некоторых объектно-ориентированных языках (прежде всего в Java) типы проверяются на стадии выполнения — это называется *динамической проверкой типов*. В сочетании со статической проверкой типов динамическая проверка предоставляет новые возможности, однако она также увеличивает затраты ресурсов на выполнение программ.

В C++ используется статическая проверка типов, потому что на уровне среды времени выполнения язык не имеет каких-либо средств обеспечения безопасности типов. Статическая проверка типов извещает программиста об ошибочном использовании типов во время компиляции и тем самым увеличивает до максимума быстродействие программы. В процессе изучения C++ вы убедитесь в том, что в большинстве архитектурных решений предпочтение отдается эффективному,

рассчитанному на практический результат программированию, характерному для языка C.

Язык C++ позволяет отключить статическую проверку типов. Кроме того, вы можете реализовать собственную динамическую проверку типов — для этого достаточно написать соответствующий фрагмент программного кода.

Средства раздельной компиляции

Раздельная компиляция особенно важна при работе над крупными проектами. Языки C и C++ позволяют собирать программы из небольших, удобных, независимо тестируемых частей. Разбиение программ чаще всего осуществляется посредством определения именованных подпрограмм, которые в C и C++ называются *функциями*. Функция представляет собой фрагмент программного кода, который для раздельной компиляции может быть перемещен в другой файл. Иначе говоря, функция может рассматриваться как атомарный блок программного кода, поскольку разные части функции не могут находиться в разных файлах; вся функция обязательно размещается в одном файле (хотя один файл может содержать более одной функции).

При вызове функции обычно передаются *аргументы*, то есть исходные данные, с которыми должна работать функция во время своего выполнения. После завершения своей работы функция обычно выдает результат — *возвращаемое значение*. Впрочем, функции могут вызываться без аргументов и не возвращать никаких значений.

Если создаваемая программа состоит из нескольких файлов, то функции одного файла должны обращаться к функциям и данным, находящимся в других файлах. При компиляции файла компилятор C или C++ должен обладать информацией о функциях и данных из других файлов, в том числе знать их имена и способ их правильного использования. Компилятор гарантирует правильность использования функций и данных. Передача компилятору информации об именах внешних функций и данных называется *объявлением*. После объявления функции или переменной компилятор обеспечит проверку правильности ее использования.

Объявления и определения

Читатель должен хорошо понимать различия между *объявлениями* и *определениями*, поскольку в книге эти термины будут использоваться достаточно часто. Объявления встречаются практически во всех программах C и C++. Прежде чем браться за свою первую программу, необходимо понять, как правильно записать объявление.

Объявление лишь сообщает компилятору некоторое имя (идентификатор). Фактически объявление означает: «Эта функция или переменная где-то существует и выглядит так». С другой стороны, *определение* означает: «Создай здесь эту переменную» или «Создай здесь эту функцию», то есть выделяет память для указанного имени, причем это относится как к переменным, так и к функциям. В случае переменной компилятор выясняет ее размер и резервирует место в памяти для хранения данных, содержащихся в этой переменной. В случае функции компилятор генерирует код, который в конечном счете также займет место в памяти.

Переменные и функции могут объявляться в разных местах, но в С и С++ каждое определение существует только в одном экземпляре (иногда это называется «правилом единственного определения»). Если в процессе объединения объектных модулей компоновщик обнаруживает повторное определение функции или переменной, обычно он выдает сообщение об ошибке.

Определение может одновременно являться объявлением. Если компилятор еще не встречал имя x, то, найдя определение int x;, компилятор воспринимает имя как объявление и немедленно выделяет для него память.

Синтаксис объявления функции

Объявление функции в С и С++ содержит имя функции, типы передаваемых ей аргументов и возвращаемого значения. Ниже приведено объявление функции func1(), которая получает два целых аргумента (целые числа в С/С++ обозначаются ключевым словом int) и возвращает целое число:

```
int funcl(int,int);
```

Сначала в объявлении указывается тип возвращаемого значения — в данном случае int. Типы аргументов перечисляются в круглых скобках после имени функции в порядке передачи. Точка с запятой обозначает конец команды; в данном случае она сообщает компилятору: «Это все, определение функции отсутствует!»

Объявления функций в С и С++ имитируют синтаксис вызова. Например, если а — целочисленная переменная, то вызов функции, объявленной выше, может выглядеть так:

```
a = funcl(2,3);
```

Поскольку func1() возвращает целое число, компилятор С/С++ проверит вызов func1() и убедится в том, что переменная а способна принять целое значение, а аргументы относятся к правильным типам.

Аргументы в объявлениях функций могут быть снабжены именами. Компилятор игнорирует имена, но программа становится более наглядной для пользователя. Например, следующее объявление функции func1() эквивалентно приведенному выше:

```
int funcl(int length, int width);
```

Предупреждение

В С и С++ функции с пустым списком аргументов интерпретируются совершенно по-разному. В С следующее объявление означает функцию с произвольными количеством и типами аргументов:

```
int func2();
```

Такая интерпретация мешает проверке типов, поэтому в С++ это объявление означает функцию без аргументов.

Синтаксис определения функции

Определение функции похоже на ее объявление, к которому присоединено тело — набор команд в фигурных скобках. Фигурные скобки обозначают начало и конец программного блока. Определение func1() с пустым (то есть не содержащим кода) телом выглядит так:

```
int funcl(int length, int width) { }
```

Обратите внимание: в определении функции точка с запятой (;) заменяется фигурными скобками. Поскольку фигурные скобки задают границы команды или группы команд, точка с запятой в этом случае не нужна. Если аргументы используются в теле функции, в заголовке определения должны быть указаны их имена (в нашем примере аргументы в теле функции отсутствуют и поэтому являются необязательными).

Синтаксис объявления переменных

По историческим причинам выражение «объявление переменной» выглядит противоречиво и сбивает с толку программистов, поэтому для правильного чтения программы очень важно должным образом его понимать. Объявление переменной передает компилятору информацию о ней. Оно означает следующее: «Хотя это имя раньше не встречалось, я гарантирую, что оно где-то существует и соответствует переменной типа X».

Как было показано ранее, в объявлении функции указываются тип возвращаемого значения, имя функции, список аргументов и точка с запятой. По этой информации компилятор понимает, что перед ним объявление и то, как выглядит объявляемая функция. По аналогии можно предположить, что объявление переменной состоит из типа и имени. Например, следующая команда в соответствии с этой логикой должна объявлять целочисленную переменную a:

```
int a:
```

Но здесь возникает противоречие: приведенный фрагмент содержит достаточно информации, чтобы компилятор мог зарезервировать память для целой переменной с именем a, что он и делает. Для решения этой проблемы в C и C++ было добавлено специальное ключевое слово extern, которое указывает: «Это всего лишь объявление, а переменная определяется в другом месте». Оно может означать как то, что определение является внешним (external) по отношению к текущему файлу, так и то, что определение находится где-то в этом же файле.

При объявлении переменной без ее определения ключевое слово extern ставится перед описанием переменной:

```
extern int a;
```

Ключевое слово extern также может применяться к объявлениям функций. Объявление функции func1() в этом случае принимает вид:

```
extern int func1(int length, int width):
```

Оно эквивалентно предыдущему объявлению функции func1(). Поскольку тело функции отсутствует, компилятор интерпретирует его именно как объявление, а не как определение функции. Следовательно, при объявлении функций ключевое слово extern излишне, а его присутствие не обязательно. Жаль, что проектировщики языка C не потребовали обязательного использования ключевого слова extern при объявлении функций; такое требование сделало бы программы более логичными и вызывало бы меньше недоразумений у программистов (хотя и потребовало бы ввода нескольких лишних символов — вероятно, именно этим и объясняется принятое решение).

Еще несколько примеров объявлений:

```
//: C02:Declare.cpp
// Примеры объявлений и определений
```

```
extern int i; // Объявление без определения
extern float f(float); // Объявление функции

float b; // Объявление и определение
float f(float a) { // Определение
  return a + 1.0;
}

int i; // Definition
int h(int x) { // Объявление и определение
  return x + 1;
}

int main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
} ///:~
```

В объявлениях функций идентификаторы аргументов могут отсутствовать. В определениях их присутствие обязательно (идентификаторы обязательны только в C, но не в C++).

Включение заголовочных файлов

Многие библиотеки содержат большое количество функций и переменных. Чтобы избавить программиста от лишней работы и обеспечить логическую последовательность внешних объявлений, в C и C++ используется механизм *заголовочных файлов*. Заголовочным файлом называется файл, содержащий внешние объявления библиотеки; по общепринятой схеме ему присваивается имя файла библиотеки с расширением .h, например headerfile.h. В старых программах иногда встречаются другие расширения (например, .hxx или .hpp), но это бывает редко.

Создатель библиотеки представляет заголовочный файл. Чтобы объявить в программе функции и внешние переменные этой библиотеки, пользователь просто включает в программу заголовочный файл препроцессорной директивой #include. Директива указывает препроцессору, что он должен открыть файл с заданным именем и вставить его содержимое на место директивы #include. Имена файлов в директиве #include могут задаваться двумя способами: в угловых скобках (<>) или в кавычках.

При включении имени файла в угловых скобках препроцессор ищет файл способом, специфическим для конкретной реализации:

```
#include <header>
```

Обычно используется некая разновидность «пути поиска», задаваемого в переменной окружения или в командной строке компилятора. Способ определения пути поиска зависит от компьютера, операционной системы и реализации C++. Возможно, вам придется собрать дополнительную информацию по этому вопросу.

Директива с именем файла, заключенным в кавычки, сообщает препроцессору, что поиск заданного файла (согласно спецификации) должен осуществляться «способом, определяемым реализацией»:

```
#include "local.h"
```

Обычно это означает, что поиск файла начинается с текущего каталога. Если файл не обнаружен, то директива обрабатывается повторно так, как если бы вместо кавычек использовались угловые скобки.

Следующая директива включает в программу заголовочный файл библиотеки потоков ввода-вывода:

```
#include <iostream>
```

Препроцессор находит заголовочный файл библиотеки (обычно этот файл находится в подкаталоге с именем include) и включает его в программу.

Имена заголовочных файлов C++

По мере развития C++ производители компиляторов выбирали разные расширения для заголовочных файлов. Кроме того, в операционных системах устанавливаются различные ограничения для имен файлов, прежде всего — на длину имени. Все это создавало проблемы с переносимостью исходных текстов. Для сглаживания этих различий в стандарте была использована схема, при которой длина имени не ограничивается пресловутыми восемью символами, а расширение не указывается. Например, рассмотрим старую директиву для включения файла iostream.h:

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

Теперь эта директива выглядит так:

```
#include <iostream>
```

Транслятор обрабатывает директивы включения в соответствии со спецификой конкретного компилятора и операционной системы; при необходимости он усекает имя файла и добавляет к нему расширение. Конечно, вы также можете удалить расширения из заголовков, предоставленных разработчиком компилятора, если новый стиль еще не поддерживается вашим компилятором.

Библиотеки, унаследованные из языка С, сохранили традиционное расширение .h. Впрочем, их можно использовать и при включении в современном стиле C++, для чего имя файла снабжается префиксом c. Рассмотрим следующий фрагмент:

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

Теперь он выглядит так:

```
#include <cstdio>
#include <cstdlib>
```

Это правило распространяется на все стандартные заголовки С. Читатель программы сразу понимает, когда в программе используются библиотеки С, а когда — библиотеки C++.

Новая схема включения не тождественна старой: с расширением .h используется традиционный вариант включения, а без него — новая схема включения с применением шаблонов. Смешение двух форм в одной программе обычно ни к чему хорошему не приводит.

Компоновка

Компоновщик собирает объектные модули (обычно это файлы с расширением .o или .obj), сгенерированные компилятором, в исполняемую программу, которая

может загружаться и запускаться операционной системой. Компоновка является последней стадией процесса компиляции.

Поведение компоновщика зависит от операционной системы. Как правило, программист просто сообщает компоновщику имена объектных модулей и библиотек, а также имя исполняемого файла. В некоторых системах компоновщик запускается отдельно, но в большинстве пакетов C++ он запускается при помощи компилятора C++. Нередко запуск компоновщика происходит незаметно для программиста.

Некоторые старые компоновщики ограничиваются однократным поиском объектных файлов и библиотек, при этом переданный список просматривается слева направо. В таких случаях порядок перечисления объектных файлов и библиотек может оказаться существенным. Если вы столкнулись с загадочной проблемой, которая проявляется лишь на стадии компоновки, возможно, все дело в очередности передачи файлов компоновщику.

Использование библиотек

После знакомства с базовой терминологией вы сможете понять, как используются библиотеки. Для этого необходимо соблюсти определенные условия.

1. Включить в программу заголовочный файл библиотеки.
2. Использовать в программе функции и переменные библиотеки.
3. Скомпоновать библиотеку с исполняемой программой.

Все эти действия также выполняются и в том случае, если объектные модули не были скомпонованы в библиотеку. Включение заголовочного файла и компоновка объектного модуля являются важнейшими аспектами раздельной компиляции в C и C++.

Поиск библиотек компилятором

Обнаружив внешнюю ссылку на функцию или переменную в C или C++, компоновщик делает одно из двух. Если определение этой функции или переменной ранее не встречалось, идентификатор включается в список «неразрешенных ссылок». Если определение уже известно, то компоновщик разрешает ссылку.

Если определение отсутствует в списке объектных модулей, компоновщик просматривает библиотеки. В библиотеках поддерживаются средства индексирования, благодаря которым компоновщику не нужно просматривать все содержимое всех объектных модулей — достаточно просмотреть индекс. Если компоновщик находит определение в библиотеке, то к исполняемой программе подключается не отдельное определение, а объектный модуль. Обратите внимание: подключается не вся библиотека, а только объектный модуль с нужным определением (тем самым предотвращается излишнее увеличение объема программы). Если вы строите собственную библиотеку и хотите свести к минимуму размер исполняемых программ, подумайте о том, чтобы выделить каждую функцию в отдельный объектный файл. Такое решение увеличит объем работы по редактированию[1], но может помочь пользователям.

---

[1] Для автоматизации работы с библиотеками рекомендуется воспользоваться языком Perl (www.perl.org) или Python (www.python.org).

Поскольку компоновщик ищет файлы в порядке передачи, вызов библиотечной функции можно «перехватить» — для этого перед именем библиотеки вставляется файл, содержащий одноименную функцию. Так как при разрешении ссылок компоновщик перед поиском в библиотеке встретит вашу функцию, она и будет использоваться вместо библиотечной функции. Впрочем, подобные замены также становятся причиной ошибок, для предотвращения которых в С++ были определены пространства имен.

Секретные включения

При построении исполняемых программ С и С++ в них тайно включаются служебные модули. Например, модуль запуска содержит код инициализации, который должен выполняться в начале работы любой программы С или С++. Этот код готовит стек к использованию и инициализирует некоторые переменные программы.

Компоновщик всегда ищет в стандартной библиотеке откомпилированные версии всех «стандартных» функций, вызываемых в программе. Так как поиск в стандартной библиотеке производится всегда, для использования определений из стандартной библиотеки достаточно включить в программу соответствующий заголовочный файл; не нужно указывать, что поиск должен производиться в стандартной библиотеке. Например, функции потокового ввода-вывода определяются в стандартной библиотеке С++, и для их использования достаточно включить в программу заголовочный файл <iostream>.

Если вы используете внешнюю библиотеку, имя этой библиотеки необходимо включить в список файлов, передаваемый компоновщику.

Использовании библиотек С

При программировании на С++ не запрещается использовать библиотечные функции С. Более того, по умолчанию вся библиотека С входит в стандарт С++. Функции С выполняют огромный объем работы и экономят массу времени.

Там, где это удобно, в книге будут использоваться функции стандартной библиотеки С++ (а следовательно, и стандартной библиотеки С), поскольку только *стандартные* функции не нарушают переносимости программ. В редких случаях вынужденного использования библиотечных функций, не входящих в стандарт С++, автор постарается ограничиться POSIX-совместимыми функциями. Стандарт POSIX был создан на базе проектов стандартизации Unix; в него были включены функции, выходящие за рамки библиотеки С++. Функции POSIX обычно поддерживаются на платформе Unix (включая Linux), а также на платформах DOS и Windows. Например, при многопоточном программировании рекомендуется использовать средства POSIX, поскольку это упрощает чтение, адаптацию и сопровождение программ (многопоточные функции POSIX обычно ограничиваются использованием базовых средств операционной системы, если они поддерживаются).

Первая программа на С++

Теперь вы знаете почти все, что необходимо для создания и компиляции простейшей программы, использующей классы потоков ввода-вывода стандартной библиотеки С++. Эти классы предназначены для чтения и записи данных в «стандарт-

ные» каналы ввода и вывода (обычно эти каналы связываются с консолью, но ввод-вывод также может перенаправляться в файлы или на другие устройства). В этой простой программе потоковый объект используется для вывода сообщения на экран.

Использование классов библиотеки Iostream

Для объявления функций и внешних данных библиотеки потоков ввода-вывода в программу включается соответствующий заголовочный файл. Директива включения выглядит так:

```
#include <iostream>
```

В нашей первой программе используется концепция стандартного ввода — «единого приемника» для выводимых данных. В других примерах стандартный вывод будет использоваться иначе, а в этой программе данные просто направляются на консоль. Пакет потоков ввода-вывода автоматически определяет переменную (объект) с именем cout, которая принимает все данные, передаваемые в стандартный вывод.

Данные передаются в стандартный поток вывода оператором <<. Программисты С знают его как «оператор поразрядного сдвига влево» (эта тема рассматривается в следующей главе). Пока достаточно сказать, что поразрядный сдвиг не имеет никакого отношения к выводу, но в языке С++ поддерживается *перегрузка* операторов, то есть наделение их новым смыслом при использовании с объектами определенного типа. Для объектов потоков ввода-вывода оператор << означает «передать данные». Например, следующая команда отправляет строку «howdy!» в объект cout (имя объекта является сокращением от «console output», то есть «консольный вывод»):

```
cout << "howdy!";
```

Даже этих кратких сведений о перегрузке операторов для начала вполне достаточно. В главе 12 перегрузка операторов будет рассмотрена более подробно.

Пространства имен

Как упоминалось в главе 1, одной из проблем языка С был «дефицит имен» функций и идентификаторов в программах. Конечно, имен на самом деле хватало; просто в какой-то момент становилось трудно выдумывать новые «свободные» имена. Что еще важнее, при достижении определенного размера программы обычно делятся на части, которые создаются и сопровождаются разными программистами и группами. В языке С все идентификаторы и имена функций существуют в едином пространстве, а это означает, что разработчики должны внимательно следить за использованием имен и предотвращать возможные конфликты. Это занятие быстро утомляет, отнимает много времени и в конечном счете оборачивается дополнительными расходами.

В стандарте С++ определен механизм для предотвращения подобных конфликтов — ключевое слово namespace. Определения С++ в библиотеках или программах группируются по пространствам имен. Существование другого определения с тем же именем, но в другом пространстве имен не создает конфликтов.

Пространства имен — удобный и полезный механизм, и программист должен учитывать существование этих пространств при работе над программой. Если вы

просто включите заголовочный файл в программу и попробуете воспользоваться функциями или объектами из этого файла, вероятно, во время компиляции возникнут странные ошибки — компилятор не найдет объявления, ради которых в программу включался заголовочный файл! Встретив эту ошибку пару раз, вы поймете ее смысл (в сообщении говорится примерно следующее: «В программу был включен заголовочный файл, но все его объявления принадлежат пространству имен, а вы не сообщили компилятору об использовании этого пространства»).

В C++ существует ключевое слово using, которое означает: «В программе будут присутствовать объявления и/или определения из этого пространства имен». Все стандартные библиотеки C++ принадлежат к единому пространству имен std (сокращение от «standard»). В этой книге почти во всех примерах используются только стандартные библиотеки, поэтому следующая директива встречается буквально в каждой программе:

```
using namespace std;
```

Эта директива открывает программе доступ ко всем элементам из пространства имен std. После ее выполнения вам уже не придется беспокоиться о том, что компоненты указанной библиотеки выделены в пространство имен, — это пространство становится доступным во всем файле, содержащем директиву using.

Массовое получение доступа ко всем элементам пространства имен после того, как кто-то постарался скрыть их, выглядит не совсем логично. В самом деле, к использованию этой директивы следует подходить разумно и ответственно (как вскоре будет показано в книге). Однако директива using предоставляет доступ к именам только в границах текущего файла, поэтому ситуация не столь драматична, как кажется на первый взгляд. Тем не менее дважды подумайте перед тем, как включать директиву using в заголовочный файл, — это крайне опрометчивый поступок.

Между пространствами имен и способом включения заголовочных файлов существует связь. Перед стандартизацией современного стиля включения (без расширения — <iostream>) заголовочные файлы обычно включались с указанием расширения .h (<iostream.h>). В то время пространства имен также не поддерживались в языке C++. Рассмотрим следующую директиву:

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

Для сохранения совместимости с существующим кодом вместо этой директивы должны быть указаны две:

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

В этой книге будет использоваться стандартный формат включения (без расширения .h), поэтому директива using должна явно присутствовать в программе.

Пока это все, что вам необходимо знать о пространствах имен. В главе 10 эта тема рассматривается гораздо более подробно.

Базовые сведения о строении программы

Программа на языке C или C++ состоит из переменных, определений и вызовов функций. При запуске программы сначала выполняется код инициализации, а затем вызывается специальная функция main(), содержащая основной код.

Как упоминалось выше, определение функции состоит из типа возвращаемого значения (обязательное в C++), имени функции, списка аргументов в круглых скобках и тела функции в фигурных скобках. Пример определения функции:

```
int function() {
  // Код функции (данная строка - комментарий)
}
```

Приведенное определение содержит пустой список аргументов, а тело функции состоит из одного комментария.

Определение функции может содержать несколько пар фигурных скобок, но присутствие только одной пары (той, в которую заключено тело функции) является обязательным. Функция main() должна подчиняться тем же правилам, что и остальные функции. В C++ функция main() всегда возвращает тип int.

Программы на языках С и C++ пишутся в свободном формате. За немногочисленными исключениями компилятор игнорирует переводы строк и пропуски, поэтому конец команды приходится обозначать специальным синтаксическим элементом — символом точки с запятой (;).

Комментарии С начинаются с символов /* и завершаются символами */. Внутри таких комментариев допускаются разрывы строк. C++ поддерживает традиционные комментарии С, а также дополнительные однострочные комментарии, начинающиеся с символов //. Такой комментарий завершается символом новой строки. Для кратких однострочных пояснений этот тип комментариев удобнее конструкции /*...*/, поэтому он широко используется в книге.

Программа «Hello, world!»

Наконец, пора привести текст нашей первой программы:

```
//: C02:Hello.cpp
// Вывод приветствия на C++
#include <iostream> // Объявления потоков ввода-вывода
using namespace std;

int main() {
  cout << "Hello. World! I am "
       << 8 << " Today!" << endl;
} ///:~
```

Аргументы операции вывода передаются объекту cout при помощи оператора <<. Вывод осуществляется в порядке перечисления аргументов (то есть слева направо). Специальная функция ввода-вывода endl начинает новую строку в выходных данных. Потоки ввода-вывода для удобства работы с классом поддерживают «цепочечную» передачу аргументов, как в приведенном примере.

В языке С «строкой» традиционно называется текст, заключенный в кавычки. В стандартную библиотеку C++ был включен мощный класс string, предназначенный для работы с текстом, поэтому текст в кавычках автор называет более точным термином *символьный массив*.

Компилятор выделяет память для символьных массивов и сохраняет в выделенной области ASCII-коды всех указанных символов. Для обозначения конца символьного массива компилятор автоматически завершает его элементом, содержащим значение 0.

Символьные массивы могут содержать специальные символы, для включения которых используются так называемые *служебные последовательности*, состоящие из обратной косой черты (\) и специального кода. Например, последовательность \n обозначает символ новой строки. Полный список служебных последовательностей можно найти в документации по компилятору или в учебнике по C; среди других примеров стоит упомянуть \t (символ табуляции), \\ (обратная косая черта) и \b (забой).

Помните, что команда может записываться в нескольких строках, а ее конец обозначается символом точки с запятой (;).

В приведенной выше команде смешиваются аргументы разных типов — символьные массивы и числовые константы. Поскольку для объекта cout оператор << перегружен с множеством разных интерпретаций, вы можете передавать этому объекту разные аргументы, а он сам выберет нужную интерпретацию.

Примеры, приводимые в книге, обычно начинаются с признака комментария (как правило, //), за которым следует двоеточие, а последняя строка листинга завершается комментарием с символами /:~. Автор использует этот прием для того, чтобы упростить выборку информации из файлов с исходными текстами программ. В первой строке также указывается имя и местонахождение файла, что позволяет ссылаться на него в тексте книги и из других файлов, а также помогает найти пример в архиве исходных текстов, который можно найти по адресу http://www.piter.com.

Запуск компилятора

Загрузите и распакуйте архив примеров, найдите программу Hello.cpp в подкаталоге C02 и запустите компилятор с аргументом Hello.cpp. Для простых программ, состоящих из одного файла, большинство компиляторов не требует других аргументов. Например, для бесплатно распространяемого в Интернете компилятора GNU C++ команда выглядит так:

```
g++ Hello.cpp
```

Команды запуска других компиляторов имеют сходный синтаксис; за подробностями обращайтесь к документации компилятора.

О потоках ввода-вывода

Пока мы встретились только с простейшим применением классов библиотеки потоков ввода-вывода. Однако потоки ввода-вывода также обладают средствами форматирования чисел в десятичной, восьмеричной и шестнадцатеричной системах счисления. Рассмотрим пример использования потоков ввода-вывода:

```
//: C02:Stream2.cpp
// Дополнительные возможности потоков
#include <iostream>
using namespace std;

int main() {
  // Определение формата при помощи манипуляторов:
  cout << "a number in decimal: "
       << dec << 15 << endl;
  cout << "in octal: " << oct << 15 << endl;
```

```
    cout << "in hex: " << hex << 15 << endl;
    cout << "a floating-point number: "
         << 3.14159 << endl;
    cout << "non-printing char (escape): "
         << char(27) << endl;
} ///:~
```

В этом примере потоковый класс выводит числа в десятичной, восьмеричной и шестнадцатеричной системах счисления при помощи *манипуляторов* (которые сами ничего не выводят, но изменяют состояние потока вывода). Формат вещественных чисел автоматически определяется компилятором. Кроме того, символ с произвольным кодом можно передать в поток путем *приведения* к типу char (char — тип данных для хранения отдельных символов). Преобразование внешне напоминает вызов функции: char(), где в круглых скобках указывается ASCII-код символа. В приведенной выше программе конструкция char(27) передает в cout символ «Escape».

Конкатенация символьных массивов

Конкатенация символьных массивов является одной из важных возможностей препроцессора С и задействована в некоторых примерах, встречающихся в книге. Два смежных массива, заключенных в кавычки, объединяются компилятором в один символьный массив. Это особенно удобно при ограничении максимальной ширины листинга:

```
//: C02:Concat.cpp
// Конкатенация символьных массивов
#include <iostream>
using namespace std;

int main() {
  cout << "This is far too long to put on a "
    "single line but it can be broken up with "
    "no ill effects\nas long as there is no "
    "punctuation separating adjacent character "
    "arrays.\n";
} ///:~
```

На первый взгляд кажется, что эта программа содержит ошибки — ее строки не завершаются знакомым символом точки с запятой (;). Но не забывайте, что программы на языках С и С++ имеют свободный формат. Хотя на практике строки программы часто завершаются точкой с запятой, на самом деле этот символ должен завершать каждую *команду*, а команда может располагаться на нескольких строках.

Чтение входных данных

В библиотеку потоков ввода-вывода также входят классы для чтения входных данных. Для чтения данных из стандартного ввода используется объект cin (сокращение от «console input», то есть «консольный ввод»). Объект cin обычно получает данные с консоли, но ввод также можно перенаправить на другие источники. Пример перенаправления ввода приведен далее в этой главе.

Для ввода из объекта cin используется оператор >>. Предполагается, что оператор получает входные данные, тип которых соответствует типу аргумента. Например, при вызове с целочисленным аргументом оператор пытается прочитать с консоли целое число:

```
//: C02:Numconv.cpp
// Преобразование десятичных чисел в восьмеричную и шестнадцатеричную форму
#include <iostream>
using namespace std;

int main() {
  int number;
  cout << "Enter a decimal number: ";
  cin >> number;
  cout << "value in octal = 0"
       << oct << number << endl;
  cout << "value in hex = 0x"
       << hex << number << endl;
} ///:~
```

Программа преобразует целое число, введенное пользователем, в восьмеричное и шестнадцатеричное представление.

Запуск других программ

Хотя программы, читающие данные из стандартного ввода и выводящие их в стандартный вывод, обычно используются в сценариях командного интерпретатора Unix и в пакетных файлах DOS, программы C и C++ могут запускать любые программы при помощи стандартной функции C system(), объявленной в заголовочном файле <cstdlib>:

```
//: C02:CallHello.cpp
// Вызов другой программы
#include <cstdlib> // Объявление "system()"
using namespace std;

int main() {
  system("Hello");
} ///:~
```

При вызове функции system() передается символьный массив с данными, которые обычно вводятся в приглашении командной строки операционной системы (возможно, с включением аргументов). Массив может строиться динамически во время выполнения программы в отличие от статического массива в приведенном примере. После выполнения команды управление возвращается программе.

Этот пример показывает, как легко использовать простые библиотечные функции C в языке C++: достаточно включить заголовочный файл и вызвать нужную функцию. Совместимость с языком C — большое подспорье для тех, кто изучает C++, имея опыт программирования на C.

Знакомство со строками

Хотя символьные массивы приносят несомненную пользу, их возможности ограниченны. Символьный массив — не более чем простая группа символов в памяти, и при выполнении всех операций программисту приходится заботиться обо всех мелочах. Например, размер символьного массива фиксируется на стадии компиляции. Если вы создали символьный массив, а позднее захотели добавить в него новые символы, то для решения этой простой задачи вам придется разобраться во

многих аспектах, включая динамическое управление памятью, копирование символьных массивов и конкатенацию. Именно такую черновую работу обычно хочется поручить объектам.

Стандартный класс C++ string предназначен для всех низкоуровневых манипуляций с символьными массивами (а также маскировки этих манипуляций от пользователя), с которыми раньше неизбежно сталкивались программисты С. Подобные манипуляции с момента появления языка С постоянно приводили к потерям времени и ошибкам. Хотя во втором томе настоящей книги классу string посвящена целая глава, этот класс так важен и настолько упрощает работу программистов, что мы будем постоянно использовать его уже в этом томе.

Чтобы использовать в программе объекты string, необходимо включить в нее заголовочный файл C++ <string>. Класс string определен в пространстве имен std, поэтому в программу включается директива using. Благодаря перегрузке операторов синтаксис директивы string интуитивно понятен:

```
//: C02:HelloStrings.cpp
// Пример использования стандартного класса C++ string
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1, s2; // Пустые строки
  string s3 = "Hello, World."; // Инициализированная строка
  string s4("I-am"); // Другой пример инициализации
  s2 = "Today"; // Присваивание
  s1 = s3 + " " + s4; // Слияние строк
  s1 += " 8 "; // Присоединение новых символов к строке
  cout << s1 + s2 + "!" << endl;
} ///:~
```

Первые две строки, s1 и s2, создаются пустыми, а строки s3 и s4 иллюстрируют два эквивалентных способа инициализации объектов string по символьным массивам (объекты string также легко инициализируются по другим объектам string).

Любому объекту string можно присвоить новое значение оператором =. Прежнее содержимое строки замещается данными, находящимися в правой части оператора присваивания, и программисту не нужно беспокоиться об удалении старых данных — это происходит автоматически. Объединение (конкатенация) строк осуществляется оператором +, который также позволяет объединять символьные массивы с объектами string. Для присоединения объекта string или символьного массива к другому объекту string можно воспользоваться оператором +=. А так как потоки ввода-вывода уже умеют работать с объектами string, чтобы вывести такой объект, достаточно передать в cout его или выражение, результат которого относится к типу string (как выражение s1 + s2 + "!" в приведенном примере).

Чтение и запись файлов

В языке С операции чтения и модификации файлов выполнялись относительно сложно и требовали определенной подготовки со стороны программиста. Потоковая библиотека C++ предоставляет простые средства для работы с файлами.

Чтобы открывать файлы для чтения и записи, следует включить в программу заголовочный файл <fstream>. Включение этого файла автоматически приводит к включению файла <iostream>. Если вы собираетесь ограничиться потоками cin, cout и т. д., достаточно явного включения файла <iostream>.

Чтобы открыть файл для чтения, создайте объект ifstream. Операции с этим объектом выполняются почти так же, как с cin. Чтобы открыть файл для записи, создайте объект ofstream, аналогичный cout. После открытия файла чтение из него и запись в него осуществляются так же, как с любым другим потоком ввода-вывода. Как видите, все очень просто (для чего, собственно, и проектировались классы потокового ввода-вывода).

Функция getline() — одна из самых полезных функций библиотеки потоков ввода-вывода. Она читает одну строку (завершенную символом новой строки) в объект string. В первом аргументе функции передается объект ifstream, из которого читаются данные, а во втором аргументе передается объект string. После вызова функции объект string содержит прочитанную информацию.

В следующем простом примере функция getline() копирует содержимое одного файла в другой файл:

```
//: C02:Scopy.cpp
// Построчное копирование файлов
#include <string>
#include <fstream>
using namespace std;

int main() {
  ifstream in("Scopy.cpp"); // Открытие файла для чтения
  ofstream out("Scopy2.cpp"); // Открытие файла для записи
  string s;
  while(getline(in, s)) // Символ новой строки при чтении теряется...
    out << s << "\n"; // ... поэтому его необходимо передать отдельно.
} ///:~
```

Чтобы открыть файл, вы просто передаете его имя объекту ifstream или ofstream, как это сделано выше.

В этом примере представлена новая программная конструкция — цикл while. Тема циклов будет подробно рассмотрена в следующей главе, а пока достаточно сказать, что выражение в круглых скобках после ключевого слова while управляет выполнением следующей команды (или блока из нескольких команд, заключенных в фигурные скобки). Пока выражение в круглых скобках (в данном случае — getline(in,s)) остается истинным, тело цикла продолжает выполняться. Если очередная строка была прочитана успешно, функция getline() возвращает значение, которое интерпретируется как «истинное», а при достижении конца входных данных возвращается «ложное» значение. Таким образом, приведенный выше цикл while читает каждую строку входного файла и выводит ее в выходной файл.

Функция getline() читает символы до тех пор, пока не встретит символ новой строки (вообще говоря, символ-завершитель можно изменять, но эта возможность будет рассматриваться только во втором томе). Тем не менее функция игнорирует символ новой строки и не сохраняет его в итоговом объекте string. Следовательно, чтобы содержимое скопированного файла точно совпадало с исходным, необходимо отдельно вывести символ новой строки, как это сделано выше.

В другом интересном примере содержимое всего файла копируется в один объект string:

```
//: C02:FillString.cpp
// Чтение всего файла в отдельную строку
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in("FillString.cpp");
  string s, line;
  while(getline(in, line))
    s += line + "\n";
  cout << s;
} ///:~
```

Память для объектов string выделяется и освобождается динамически, поэтому вам не придется беспокоиться о том, сколько памяти занимает объект. Вы просто добавляете новую информацию, а объект string автоматически увеличивается по размерам хранящихся в нем данных.

Зачем нужно сохранять целый файл в объекте string? Класс string содержит множество функций поиска и модификации, которые позволяют изменять содержимое целого файла через один строковый объект. Тем не менее у такого подхода есть свои недостатки. В частности, с файлом часто удобнее работать как с совокупностью строк, а не как с одним большим текстовым фрагментом. Например, если потребуется перенумеровать строки, было бы гораздо проще, если бы каждая строка хранилась в отдельном объекте string. Для этого потребуется другое решение.

Знакомство с векторами

Работая с объектом string, можно заполнять его данными, не беспокоясь об объеме необходимой памяти. Однако при чтении строк файла в отдельные объекты string возникает другая проблема: нельзя заранее сказать, сколько объектов вам понадобится, — это выяснится лишь после чтения всего файла. Для решения этой проблемы нужно некое «хранилище», которое бы автоматически расширялось и вмещало столько объектов string, сколько мы в него занесем.

Собственно, стоит ли ограничиваться хранением объектов string? Оказывается, подобная проблема — работа с объектами, количество которых неизвестно на момент написания программы, — встречается довольно часто. И этот объект-«контейнер» принесет гораздо больше пользы, если его можно будет использовать для хранения *объектов произвольного типа*! К счастью, в стандартной библиотеке C++ имеется готовое решение — классы стандартных контейнеров. Контейнер является одним из «краеугольных камней» библиотеки C++.

Контейнеры и алгоритмы стандартной библиотеки C++ нередко путают с объектами библиотеки STL (Standard Template Library). Алексей Степанов, который тогда работал в Hewlett-Packard, использовал название STL при представлении своей библиотеки в комитет по стандартизации C++ на встрече в Сан-Диего, Калифорния, весной 1994 года. Название прижилось, особенно после того, как фирма Hewlett-Packard решила предоставить библиотеку для свободного распростра-

нения. Тем временем комитет интегрировал библиотеку STL в стандартную библиотеку C++ с внесением многочисленных изменений. Дальнейшая работа над STL продолжается в Silicon Graphics (SGI; см. http://www.sgi.com/Technology/STL). Между библиотекой SGI STL и стандартной библиотекой C++ существует множество тонких различий. Таким образом, несмотря на популярность этого заблуждения, STL *не является* частью стандартной библиотеки C++. Путаница возникает довольно часто, потому что контейнеры и алгоритмы стандартной библиотеки C++ происходят от одного корня (а нередко совпадают по именам). В этой книге автор будет использовать выражения «стандартная библиотека C++», «контейнеры стандартной библиотеки» или что-нибудь в этом роде, намеренно избегая упоминания библиотеки STL.

Хотя при реализации контейнеров и алгоритмов стандартной библиотеки C++ используются нетривиальные концепции, а их полное описание занимает две большие главы во втором томе, библиотека принесет пользу даже в том случае, если не вникать во все тонкости. Более того, она настолько полезна, что простейший из стандартных контейнеров — вектор — представляется в одной из начальных глав и используется во всей книге. Вскоре вы убедитесь, что можно очень много сделать, если знать основные принципы работы класса vector, не разбираясь в его реализации (кстати, это одна из важных целей ООП). Поскольку векторы и другие контейнеры будут подробно рассмотрены во втором томе при описании стандартной библиотеки, простительно, что класс vector в начале книги задействован не совсем так, как это бы сделал опытный программист C++. Как правило, и такого упрощенного применения оказывается достаточно.

Класс vector оформлен в виде *шаблона*, что позволяет эффективно использовать его с разными типами. Другими словами, можно создать вектор объектов shape, вектор объектов cat, вектор объектов string и т. д. При помощи шаблона можно создать «класс чего угодно». Чтобы сообщить компилятору, с каким типом данных будет работать класс (в данном случае — какие элементы будут храниться в векторе), укажите имя нужного типа в угловых скобках <...>. Так, вектор объектов string обозначается vector<string>. Такая запись определяет специализированный вектор, в котором могут храниться только объекты string. Если попытаться занести в него объект другого типа, компилятор выдаст сообщение об ошибке.

Поскольку класс vector представляет собой «контейнер», то есть предназначается для хранения однотипных элементов, в нем должны быть предусмотрены средства для сохранения и извлечения элементов. Новые элементы добавляются в конец вектора функцией push_back() (помните, что эта функция принадлежит классу, поэтому, чтобы вызвать ее для конкретного объекта, нужно отделить ее имя от имени объекта символом точки). На первый взгляд имя функции push_back() можно было бы заменить чем-нибудь попроще (например, put()), но на то есть свои причины: существуют другие контейнеры и другие функции для занесения новых элементов в контейнер. Например, функция insert() вставляет элементы в середину контейнера. Класс vector поддерживает эту функцию, но она более сложна, поэтому ее описание откладывается до второго тома книги. Также существует функция push_front() для занесения элементов в начало вектора, однако векторами она не поддерживается. Класс vector содержит много других функций, а в стандартную библиотеку входят другие контейнеры, но вы будете удивлены, как много можно сделать при помощи нескольких простых функций.

Итак, новые элементы заносятся в вектор функцией push_back(), но как прочитать их из вектора? Вместо отдельной функции используется более умное и элегантное решение — благодаря перегрузке операторов программист работает с вектором как с *массивом*. Массивы (более подробное описание приводится в следующей главе) присутствуют практически во всех языках программирования, поэтому читатель наверняка в той или иной степени знаком с ними. Массивы представляют собой *составные типы*, то есть состоят из элементов, собранных в единое целое. Отличительная особенность массивов заключается в том, что эти элементы имеют одинаковый размер и располагаются в определенном порядке. Что еще важнее, для получения элементов может использоваться механизм *индексирования* — программист запрашивает элемент с заданным номером и получает его (обычно очень быстро). В некоторых языках программирования существуют исключения, но обычно индексирование обозначается квадратными скобками. Так, для получения пятого элемента массива a используется запись a[4] (учтите, что индекс первого элемента всегда равен нулю).

Этот компактный и мощный синтаксис был внедрен в класс vector путем перегрузки операторов, по аналогии с тем, как операторы << и >> были внедрены в потоки ввода-вывода. Для нас пока несущественно, как перегружаются операторы, — эта тема будет рассмотрена в одной из следующих глав. Просто помните, что незаметно для вас происходит некое волшебство, позволяющее оператору [ ] работать с классом vector.

Учитывая все сказанное, рассмотрим пример использования векторов. Для этого следует включить в программу заголовочный файл <vector>:

```
//: C02:Fillvector.cpp
// Копирование всего содержимого файла в вектор строк
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream in("Fillvector.cpp");
  string line;
  while(getline(in, line))
    v.push_back(line); // Занесение строки в конец вектора
  // Нумерация строк:
  for(int i = 0; i < v.size(); i++)
    cout << i << ": " << v[i] << endl;
} ///:~
```

В основном эта программа аналогична предыдущей — мы открываем файл и последовательно читаем его строки в объекты string. Тем не менее на этот раз объекты string заносятся в конец вектора v. После завершения цикла while все содержимое файла будет находиться в памяти внутри объекта v.

В этом примере встречается новая программная конструкция, называемая *циклом* for. Эта разновидность циклов похожа на цикл while, но она предоставляет программисту дополнительные возможности. Как и в циклах while, после ключевого слова for следует «управляющее выражение» в круглых скобках. Но в данном случае это выражение состоит из трех частей: первая часть инициализирует, вторая

содержит условие завершения цикла, а третья вносит некоторые изменения (обычно переход к следующему элементу в некоторой последовательности). В программе продемонстрирован наиболее типичный вариант использования циклов for: инициализатор int i = 0 создает целочисленную переменную i для хранения счетчика цикла и присваивает ей нулевое значение. Условие проверки означает, что для продолжения работы цикла счетчик i должен быть меньше количества элементов в векторе v (количество элементов определяется функцией size() класса vector; автор как-то забыл упомянуть о ней выше, но смысл этой функции вполне очевиден). Наконец, в последней части переменная i увеличивается на 1, для чего используется сокращенная запись, принятая в С и С++, — *оператор инкремента*. Фактически конструкция i++ означает следующее: «взять значение i, прибавить к нему 1 и присвоить результат i». Итак, приведенный цикл for берет переменную i и последовательно присваивает ей числа от нуля до значения, на единицу меньшего размера вектора. Для каждого значения i выполняется команда вывода, которая строит строку из значения i (волшебным образом преобразованного в символьный массив объектом cout), двоеточия, пробела, текущей строки файла и символа новой строки endl. Откомпилируйте и запустите программу, и вы увидите результат — строки выходного файла окажутся пронумерованными.

Подход оператора >> к работе с потоками позволяет легко изменить программу так, чтобы файл разбивался не по строкам, а по словам, разделенным пропусками:

```
//: C02:GetWords.cpp
// Разбиение файла по словам, разделенным пропусками
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> words;
  ifstream in("GetWords.cpp");
  string word;
  while(in >> word)
    words.push_back(word);
  for(int i = 0; i < words.size(); i++)
    cout << words[i] << endl;
} ///:~
```

Следующее выражение обеспечивает ввод очередного «слова»:

```
while(in >> word)
```

Когда условие цикла становится ложным, это означает, что был достигнут конец файла. Конечно, разделение слов пропусками весьма примитивно, но для простого примера сойдет. Позднее в книге будут приведены более совершенные примеры, которые позволяют разбивать входные данные практически любым способом по вашему желанию.

Чтобы убедиться в том, как просто работать с классом vector, рассмотрим следующий пример, в котором создается вектор с элементами типа int:

```
//: C02:Intvector.cpp
// Создание вектора для хранения целых чисел
#include <iostream>
#include <vector>
```

```
using namespace std;

int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Присваивание
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
} ///:~
```

Если потребуется создать вектор для хранения данных другого типа, просто укажите этот тип в аргументе шаблона (то есть в угловых скобках). Шаблоны и хорошо спроектированные библиотеки шаблонов должны обеспечивать именно такую простоту использования.

В этом примере продемонстрировано другое важное свойство класса vector. Следующее выражение показывает, что возможности векторов не ограничиваются хранением и извлечением элементов:

```
v[i] = v[i] * 10;
```

Программист также может *присваивать* (а следовательно, изменять) значения элементов вектора, хотя для этого приходится использовать оператор индексирования [...]. Следовательно, класс vector является универсальным и гибким инструментом для работы с коллекциями объектов, и мы будем неоднократно задействовать его в будущем.

Итоги

Эта глава должна была показать, каким простым может быть объектно-ориентированное программирование *при условии*, что кто-то уже выполнил всю работу по определению объектов за вас. В этом случае остается лишь включить в программу заголовочный файл, создать объекты и по мере необходимости передавать им нужные сообщения. Мощные хорошо спроектированные типы избавляют программиста от «черновой работы», а его программы автоматически наделяются широкими возможностями.

В процессе демонстрации простоты ООП при использовании библиотечных классов в этой главе также были представлены некоторые из основных и самых полезных типов стандартной библиотеки C++, в том числе семейство классов потоков ввода-вывода, предназначенных для чтения с консоли и записи на консоль, а также класс string и шаблон vector. Теперь вы сами убедились, как просто работать с этими классами, и сможете найти для них возможные применения. Однако на самом деле возможности этих классов гораздо шире. В начале книги мы будем пользоваться крайне ограниченным подмножеством возможностей этих классов, но и они станут большим шагом вперед от примитивизма низкоуровневых языков типа C. Хотя изучение низкоуровневых аспектов C полезно с познавательной точки зрения, оно требует времени. В конечном счете ваша работа станет намного

эффективнее, если низкоуровневые задачи будут решаться объектами. Собственно, главная *цель* ООП заключается в скрытии технических подробностей, чтобы программист мог творить на более высоком уровне.

Но на какой бы высокий уровень вы ни пытались подняться при помощи ООП, без знания некоторых фундаментальных аспектов языка С вам все равно не обойтись. Эти аспекты будут рассматриваться в следующей главе.

Упражнения

1. Измените программу Hello.cpp так, чтобы она выводила ваше имя и возраст (размер обуви, возраст вашей собаки и т. д. — короче, любую информацию по вашему выбору). Откомпилируйте и запустите программу.
2. Взяв за образец программы Stream2.cpp и Numconv.cpp, напишите программу, которая запрашивает радиус круга и выводит его площадь. Для вычисления квадрата радиуса воспользуйтесь оператором *. Не пытайтесь выводить значение в восьмеричном или шестнадцатеричном представлении (они работают только с целыми типами).
3. Напишите программу, которая открывает файл и подсчитывает количество слов, разделенных пропусками.
4. Напишите программу для подсчета числа вхождений заданного слова в файл (для поиска слова воспользуйтесь оператором == класса string).
5. Измените программу Fillvector.cpp так, чтобы строки выводились в обратном порядке (от последней к первой).
6. Измените программу Fillvector.cpp так, чтобы перед выводом все элементы вектора объединялись в одну строку (нумерация строк при этом не нужна).
7. Выводите содержимое файла по строкам. После вывода каждой строки программа должна ждать, пока пользователь нажмет клавишу Enter.
8. Создайте вектор vector<float> и занесите в него 25 вещественных чисел в цикле for. Выведите содержимое вектора.
9. Создайте три объекта vector<float> и заполните первые два объекта так, как в предыдущем примере. Напишите цикл for, который суммирует соответствующие элементы первых двух векторов и заносит результат в соответствующий элемент третьего вектора. Выведите содержимое всех трех векторов.
10. Создайте вектор vector<float> и занесите в него 25 вещественных чисел, как в предыдущих примерах. Возведите каждое число в квадрат и сохраните результат в исходном элементе вектора. Выведите содержимое вектора до и после возведения в квадрат.

Элементы С в языке С++

Язык С++ создавался на базе С, поэтому для программирования на С++ необходимо знать синтаксис С — подобно тому, как дифференциальное и интегральное исчисление требует хороших познаний в области алгебры.

Если вы никогда не сталкивались с языком С, эта глава даст представление об элементах языка С, присутствующих в С++. Читатель, уже знакомый с языком С по первому изданию классической книги Кернигана (Kernighan) и Ричи (Ritchie), найдет в С++ как стандартные средства С, так и ряд новшеств и отличий. В этом случае можно бегло просмотреть главу и задержаться только на специфических аспектах С++. Кроме того, в этой главе представлены некоторые важные аспекты С++, имеющие прямые аналоги в языке С (причем в С++ их возможности часто расширены). О более сложных аспектах С++ речь пойдет лишь в следующих главах.

В данной главе приводится довольно краткий обзор конструкций С и некоторых базовых конструкций С++, при этом предполагается, что читатель обладает опытом программирования на другом языке.

Создание функций

В прежнем языке С (до появления стандарта) функции можно было вызывать с произвольными количеством или типами аргументов; компилятор не жаловался. Все выглядело хорошо до момента запуска программы. Без видимых причин программист получал необъяснимые результаты (или еще хуже, программа аварийно завершалась). Вероятно, полная свобода при передаче аргументов и загадочные ошибки, к которым она приводила, стали одной из причин, по которой С окрестили «ассемблером высокого уровня». Программисты С, работавшие до появления стандарта, просто привыкли к этой особенности языка.

В стандартных версиях С и С++ имеет место так называемая *прототипизация функций*. Это означает, что при объявлении и определении функций должны быть

указаны типы аргументов. Такое описание называется «прототипом». При вызове функции компилятор при помощи прототипа убеждается в том, что функция получила правильные аргументы, и возвращаемое значение интерпретируется правильно. Ошибки, допущенные программистом при вызове функции, успешно обнаруживаются компилятором.

В сущности, читатель уже знаком с прототипизацией по предыдущей главе (хотя там этот термин не использовался), поскольку объявление функции в C++ одновременно требует ее должной прототипизации. В прототипе функции на месте списка аргументов перечисляются типы аргументов, передаваемых функции, а также идентификаторы аргументов (не обязательные для объявлений). Порядок перечисления и типы аргументов должны совпадать при объявлении, определении и вызове функции. Пример прототипа функции при ее объявлении:

```
int translate(float x. float y. float z):
```

В прототипах функции не разрешается сокращение записи, которое характерно для определений обычных переменных. Другими словами, список аргументов не может иметь вид float x, y, z. Программист должен явно указать тип *каждого* аргумента. В объявлениях функций также допустима следующая форма записи:

```
int translate(float. float. float):
```

Поскольку объявление не выполняется, а всего лишь передает компилятору информацию для проверки типов в момент вызова, идентификаторы включаются в объявление только для того, чтобы программа стала более наглядной.

В определении функции имена аргументов обязательны, поскольку ссылки на них встречаются в теле функции:

```
int translate(float x. float y. float z) {
  x = y = z:
  // ...
}
```

Оказывается, это правило применимо только к языку C. В C++ список аргументов в определении функции может содержать аргументы без указания имен. Конечно, такие аргументы не могут использоваться в теле функции.

Анонимные аргументы разрешены для того, чтобы программист мог «зарезервировать место в списке аргументов». Кто бы ни использовал функцию, он все равно должен вызвать ее с правильным набором аргументов. Однако разработчик функции сможет задействовать аргумент в будущем, и при этом не придется изменять программы, в которых данная функция вызывается. Конечно, ничто не мешает присвоить аргументу имя и проигнорировать его в функции, но в этом случае при каждой компиляции функции вы будете получать раздражающее предупреждение о неиспользованном значении. При отсутствии имени это предупреждение не выводится.

В C и C++ существуют еще два способа объявления списка аргументов. Пустой список аргументов в C++ объявляется в виде func() — тем самым вы сообщаете компилятору, что количество аргументов равно нулю. Но следует помнить, что эта конструкция означает пустой список аргументов только в C++. В языке C она интерпретируется как «неопределенное количество аргументов» — своего рода «брешь» в системе проверки типов C, поскольку в этом случае проверка не выполняется). Как в C, так и в C++ объявление func(void); означает пустой список аргументов. Ключевое слово void в данном случае означает «ничто» (как будет по-

казано далее, при работе с указателями оно также может означать «неопределенный тип»).

Другой синтаксис списков аргументов используется в тех случаях, когда точное количество аргументов или их типы неизвестны — так называемые *переменные списки аргументов*. «Неопределенный список аргументов» обозначается многоточием (...). Функции с переменным списком аргументов определяются гораздо сложнее, чем обычные функции. Переменные списки аргументов также могут использоваться в функциях с фиксированным набором аргументов, если вы почему-либо хотите отключить механизм проверки типов по прототипу функции. По этой причине рекомендуется свести к минимуму использование переменных списков аргументов в С и полностью избегать их в C++ (где, как вы вскоре узнаете, существуют более надежные альтернативы). За информацией об обработке переменных списков аргументов обращайтесь к учебнику по языку С.

Возвращаемое значение функции

Прототип функции C++ должен явно определять тип возвращаемого значения функции (в С тип возвращаемого значения можно не указывать, по умолчанию им является тип int). Тип возвращаемого значения указывается перед именем функции. Если функция не возвращает никакого значения, используйте ключевое слово void. В этом случае любые ссылки на возвращаемое значение в программе приводят к ошибкам. Примеры прототипов функций:

```
int f1(void); // Возвращает int. вызывается без аргументов
int f2(); // Эквивалент f1() в C++. но не в стандартном языке C
float f3(float. int. char. double); // Возвращает тип float
void f4(void); // Вызывается без аргументов. не возвращает значения
```

Чтобы вернуть нужное значение из функции, воспользуйтесь ключевым словом return. Команда return передает управление из функции в точку, находящуюся непосредственно после ее вызова. Если команда return вызывается с аргументом, он становится возвращаемым значением функции. Если в прототипе функции указано, что функция возвращает некоторый тип, то этот тип должен возвращаться всеми командами return. Определение функции может содержать несколько команд return:

```
//: C03:Return.cpp
// Использование команды "return"
#include <iostream>
using namespace std;

char cfunc(int i) {
  if(i == 0)
    return 'a';
  if(i == 1)
    return 'g';
  if(i == 5)
    return 'z';
  return 'c';
}

int main() {
  cout << "type an integer: ";
  int val;
```

```
  cin >> val;
  cout << cfunc(val) << endl;
} ///:~
```

В функции cfunc() первая команда if с истинным условием завершает работу функции вызовом return. Обратите внимание: объявление функции в данном случае не обязательно, так как определение функции появляется перед ее использованием в main(), поэтому компилятор получает всю необходимую информацию из определения.

Использование библиотеки функций С

При программировании на С++ вам доступны все функции локальной библиотеки С. Прежде чем определять собственную функцию, внимательно поищите в библиотеке — вполне вероятно, что кто-то уже решил вашу задачу за вас, причем это решение может быть лучше спроектировано и отлажено.

Небольшое предупреждение: многие компиляторы содержат дополнительные функции, которые упрощают жизнь программиста, но не входят в стандартную библиотеку С. Если вы твердо уверены, что программа никогда не будет переноситься на другую платформу (а кто может быть в этом уверен?) — что же, используйте эти функции и упрощайте себе жизнь. А если вы предпочитаете сохранить переносимость приложения, ограничьтесь функциями стандартной библиотеки. При выполнении операций, специфических для конкретной платформы, постарайтесь изолировать соответствующий код в одном месте, это упростит его переработку при переносе программы на другую платформу. В С++ зависимые от платформы операции часто инкапсулируются в классах, такое решение следует признать идеальным.

Использование библиотечных функций происходит примерно так: сначала найдите функцию в справочнике (во многих справочниках по программированию функции упорядочиваются как по категориям, так и по алфавиту). Описание функции должно включать раздел с описанием программного синтаксиса. Обычно этот раздел содержит минимум одну строку #include, в которой указан заголовочный файл с прототипом функции. Включите эту директиву в свой файл, чтобы функция была должным образом объявлена. Если при вызове функции будет допущена ошибка, компилятор найдет ее, сравнив вызов с прототипом функции в заголовочном файле, и выдаст сообщение об ошибке. Компоновщик по умолчанию просматривает стандартную библиотеку, поэтому ничего больше не потребуется: включите заголовочный файл и вызовите функцию.

Разработка собственных библиотек

Ваши собственные функции тоже могут объединяться в библиотеки. Большинство программных пакетов содержит специальную утилиту-«библиотекаря» для управления группами объектных модулей. Каждый библиотекарь обладает собственным набором команд, но общая идея выглядит примерно так: сначала создается заголовочный файл с прототипами всех функций библиотеки. Поместите этот заголовочный файл в каталоги, просматриваемые препроцессором. Это может быть либо текущий каталог (чтобы заголовок был найден директивой #include "header"), либо каталог включаемых файлов (для директив #include <header>). Соберите все

объектные модули и передайте их библиотекарю вместе с именем библиотеки (большинство библиотекарей использует стандартные расширения, такие как .lib или .a). Разместите библиотеку вместе с другими библиотеками, чтобы компоновщик мог найти ее. При обращении к функциям библиотеки необходимо сообщить об этом в командной строке, чтобы компоновщик знал, что вызываемые функции следует искать в вашей библиотеке. За дополнительной информацией обращайтесь к документации, поскольку подробности зависят от конкретной системы.

Управляющие конструкции

В этом разделе описаны команды C++, управляющие выполнение программы. Знание этих команд абсолютно необходимо для чтения и написания программ на C или C++.

В C++ поддерживаются все управляющие команды языка C, в том числе if-else, while, do-while, for и команда выбора switch. Кроме того, поддерживается команда goto, имеющая дурную репутацию. В этой книге мы постараемся обойтись без этой команды.

Значения true и false

Все условные команды позволяют выбрать путь дальнейшего выполнения программы в зависимости от истинности логического выражения. Например, логическое выражение A == B при помощи условного оператора == проверяет, равны ли значения переменных A и B. Результатом выражения является логическая величина true или false (эти ключевые слова существуют только в C++; в языке C истинным считается выражение, результат которого отличен от нуля). Существуют и другие условные операторы: >, <, >= и т. д. Условные команды более подробно описаны ниже.

Цикл if-else

Команда if-else существует в двух вариантах: с секцией else и без нее. Первый вариант:

```
if(выражение)
  команда
```

Второй вариант:

```
if(выражение)
  команда
else
  команда
```

Результат *выражения* равен true или false. Под *командой* понимается либо одиночная команда, завершенная символом точки с запятой (;), либо составная команда, то есть группа простых команд в фигурных скобках. Учтите, что *командой* может быть другая команда if, то есть возможен вложенный вызов условных команд.

```
//: C03:Ifthen.cpp
// Условные команды if и is-else
#include <iostream>
using namespace std:
```

```
int main() {
  int i;
  cout << "type a number and 'Enter'" << endl;
  cin >> i;
  if(i > 5)
    cout << "It's greater than 5" << endl;
  else
    if(i < 5)
      cout << "It's less than 5 " << endl;
    else
      cout << "It's equal to 5 " << endl;

  cout << "type a number and 'Enter'" << endl;
  cin >> i;
  if(i < 10)
    if(i > 5)  // Вложенная команда "if"
      cout << "5 < i < 10" << endl;
    else
      cout << "i <= 5" << endl;
  else // Соответствует условию "if(i < 10)"
    cout << "i >= 10" << endl;
} ///:~
```

Обычно тело управляющих команд снабжается отступом, чтобы читателю было проще определить, где оно начинается и кончается[1].

Цикл while

Конструкции while, do-while и for управляют циклическим выполнением команд. Команда выполняется до тех пор, пока управляющее выражение равно false. Общая форма цикла while выглядит так:

```
while(выражение)
  команда
```

Выражение проверяется в начале цикла, а затем заново вычисляется после каждой итерации (то есть выполнения команды).

В следующем примере тело цикла while продолжает выполняться до тех пор, пока пользователь не введет правильное число или не прервет работу программы, нажав клавиши Ctrl+C.

```
//: C03:Guess.cpp
// Угадывание числа (демонстрация цикла while)
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess = 0;
  // "!=" - проверка условия "не равно":
  while(guess != secret) { // Составная команда
    cout << "guess the number: ";
    cin >> guess;
  }
```

[1] Разные авторы сходятся на том, что отступы в том или ином виде необходимы, но на этом все сходство заканчивается. Между программистами, исповедующими разными стили оформления кода, идут бесконечные войны. Стиль оформления кода, принятый в этой книге, описан в приложении А.

```
  cout << "You guessed it!" << endl;
} ///:~
```

Условные выражения циклов while не ограничиваются простым сравнением, как в рассмотренном примере. Они могут быть сколь угодно сложными, нужно лишь, чтобы выражение давало результат true или false. В некоторых программах даже встречаются циклы без тела, в которых после условия сразу же следует точка с запятой:

```
while (/* Выполнение нужных действий */)
  ;
```

Цикл do-while

Общая форма цикла do-while выглядит так:

```
do
  команда
while(выражение);
```

Цикл do-while отличается от while тем, что в нем команда всегда выполняется хотя бы один раз, даже если условие оказывается ложным при самой первой проверке. В обычных циклах while в этом случае тело цикла не выполняется.

Если бы в примере Guess.cpp использовался цикл do-while, то переменную guess не пришлось бы инициализировать фиктивным начальным значением, поскольку она инициализируется потоком cin перед проверкой:

```
//: C03:Guess2.cpp
// Угадывание числа с использованием цикла do-while
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess; // Инициализация не нужна
  do {
    cout << "guess the number: ";
    cin >> guess; // Здесь происходит инициализация
  }  while(guess != secret);
  cout << "You got it!" << endl;
} ///:~
```

По каким-то причинам многие программисты избегают циклов do-while и работают только с циклами while.

Цикл for

Цикл for проводит инициализацию перед первым выполнением. Затем проверяется условие, а в конце каждой итерации вносится некоторое изменение. Общая форма цикла for выглядит так:

```
for(инициализация;условие;изменение)
  команда
```

Любая из трех секций (*инициализация*, *условие* или *изменение*) может отсутствовать. Код *инициализации* выполняется только один раз в самом начале цикла. *Условие* проверяется перед каждой итерацией (если перед началом цикла оно окажется ложным, то *команда* не выполняется). В конце каждой итерации выполняется *изменение*.

Циклы for часто используются для последовательного перебора чисел:

```
//: C03:Charlist.cpp
// Вывод всех символов ASCII
// Демонстрация цикла "for"
#include <iostream>
using namespace std;

int main() {
  for(int i = 0; i < 128; i = i + 1)
    if (i != 26)  // Очистка экрана на терминалах ANSI
      cout << " value: " << i
           << " character: "
           << char(i) // Приведение типа
           << endl;
} ///:~
```

Обратите внимание: переменная i определяется в точке использования, а не в начале блока, обозначенном открывающей фигурной скобкой ({). В этом отношении язык C++ отличается от традиционных процедурных языков (в том числе C), в которых все переменные должны объявляться в начале блока. Эта особенность C++ обсуждается далее в этой главе.

Ключевые слова break и continue

Внутри тела любой циклической конструкции while, do-while и for можно управлять выполнением цикла при помощи ключевых слов break и continue. Команда break немедленно завершает цикл без выполнения оставшихся команд. Команда continue завершает текущую итерацию и возвращается к началу цикла, начиная следующую итерацию.

В качестве примера использования команд break и continue рассмотрим программу с очень простой системой меню:

```
//: C03:Menu.cpp
// Простая система меню
// для демонстрации команд "break" и "continue"
#include <iostream>
using namespace std;

int main() {
  char c; // Переменная для хранения ввода
  while(true) {
    cout << "MAIN MENU:" << endl;
    cout << "l: left, r: right, q: quit -> ";
    cin >> c;
    if(c == 'q')
      break; // Выход из "while(1)"
    if(c == 'l') {
      cout << "LEFT MENU:" << endl;
      cout << "select a or b: ";
      cin >> c;
      if(c == 'a') {
        cout << "you chose 'a'" << endl;
        continue; // Возврат к главному меню
      }
      if(c == 'b') {
        cout << "you chose 'b'" << endl;
```

```
        continue; // Возврат к главному меню
      }
      else {
        cout << "you didn't choose a or b!"
             << endl;
        continue; // Возврат к главному меню
      }
    }
    if(c == 'r') {
      cout << "RIGHT MENU:" << endl;
      cout << "select c or d: ";
      cin >> c;
      if(c == 'c') {
        cout << "you chose 'c'" << endl;
        continue; // Возврат к главному меню
      }
      if(c == 'd') {
        cout << "you chose 'd'" << endl;
        continue; // Возврат к главному меню
      }
      else {
        cout << "you didn't choose c or d!"
             << endl;
        continue; // Возврат к главному меню
      }
    }
    cout << "you must type l or r or q!" << endl;
  }
  cout << "quitting menu..." << endl;
} ///:~
```

Если пользователь вводит в главном меню символ «q», то цикл завершается командой break. В противном случае программа просто продолжает выполняться до бесконечности. После каждого ввода символа в подменю команда continue возвращает управление в начало цикла while.

Конструкция while(true) означает цикл, продолжающийся неограниченно долго. Команда break прерывает бесконечный цикл при вводе пользователем символа «q».

Команда switch

Команда switch выбирает один из нескольких фрагментов кода в зависимости от значения целочисленного выражения. Общая форма этой команды:

```
switch (выбор) {
  case целое_значение1 : команда; break;
  case целое_значение2 : команда; break;
  case целое_значение3 : команда; break;
  case целое_значение4 : команда; break;
  case целое_значение5 : команда; break;
  (...)
  default: команда;
}
```

Секция *выбора* представляет собой выражение с целочисленным результатом. Команда switch сравнивает этот результат с каждым *целым_значением*. При обнаружении совпадения выполняется соответствующая *команда* (простая или составная). При отсутствии совпадения выполняется команда в секции default.

Обратите внимание: в приведенном выше определении каждая секция case завершается командой break, которая передает управление на следующую после тела switch команду (то есть за фигурную скобку, завершающую switch). Такой подход к построению команд switch считается классическим, но на самом деле команды break не обязательны. При отсутствии этих команд происходит сквозное выполнение следующих секций case до тех пор, пока не будет обнаружена команда break. Обычно такое поведение нежелательно, но опытные программисты иногда им пользуются.

Команда switch четко и наглядно выражает множественный выбор (то есть выбор нескольких разных путей выполнения), но для нее необходим критерий выбора, который бы интерпретировался как целое число на стадии компиляции. Например, если выбор производится по значению объекта string, команда switch с ним работать не будет. В этом случае придется использовать серию команд if и сравнивать string внутри условий.

Предыдущий пример с меню особенно хорошо подходит для команды switch:

```
//: C03:Menu2.cpp
// Пример с меню с использованием команды switch
#include <iostream>
using namespace std;

int main() {
  bool quit = false;  // Флаг выхода
  while(quit == false) {
    cout << "Select a, b, c or q to quit: ";
    char response;
    cin >> response;
    switch(response) {
      case 'a' : cout << "you chose 'a'" << endl;
                 break;
      case 'b' : cout << "you chose 'b'" << endl;
                 break;
      case 'c' : cout << "you chose 'c'" << endl;
                 break;
      case 'q' : cout << "quitting menu" << endl;
                 quit = true;
                 break;
      default  : cout << "Please use a,b,c or q!"
                 << endl;
    }
  }
} ///:~
```

Флаг quit определяется с типом bool (сокращение от «boolean»), поддерживаемым только в C++. Он принимает всего два допустимых значения, которые определяются ключевыми словами true и false. Если введен символ «q», флагу quit присваивается значение true. При следующей проверке условия цикла выражение quit == false становится равным false, поэтому тело цикла while в этом случае не выполняется.

Ключевое слово goto

Ключевое слово goto существует в C, поэтому оно также поддерживается в C++. Команда goto часто считается проявлением плохого стиля программирования,

и в большинстве случаев это действительно так. Каждый раз, когда вы собираетесь использовать goto, проанализируйте программу и поищите другое решение. В редких случаях может оказаться, что команда goto успешно решает проблему, которая не решается иначе, и все же над этим стоит хорошо подумать. Вот один из таких примеров:

```
//: C03:gotoKeyword.cpp
// Пресловутая команда goto поддерживается в C++
#include <iostream>
using namespace std;

int main() {
  long val = 0;
  for(int i = 1; i < 1000; i++) {
    for(int j = 1; j < 100; j += 10) {
      val = i * j;
      if(val > 47000)
        goto bottom;
        // Команда break передаст управление только к внешнему циклу 'for'
    }
  }
  bottom: // Метка
  cout << val << endl;
} ///:~
```

Существует и другое решение — определить дополнительный логический флаг, который проверяется во внешнем цикле for, а затем прервать внутренний цикл for командой break. Тем не менее при нескольких уровнях вложенности циклов for или while такое решение становится слишком громоздким и неудобным.

Рекурсия

Рекурсия — интересный и нередко полезный прием программирования, при котором текущая функция вызывает сама себя. Конечно, если этим дело ограничится, функция будет вызываться снова и снова до полного исчерпания свободной памяти, поэтому должен существовать вариант выхода из цепочки рекурсивных вызовов. В следующем примере рекурсия продолжается до тех пор, пока значение cat не превысит 'Z'[1]:

```
//: C03:CatsInHats.cpp
// Простая демонстрация рекурсии
#include <iostream>
using namespace std;

void removeHat(char cat) {
  for(char c = 'A'; c < cat; c++)
    cout << "  ";
  if(cat <= 'Z') {
    cout << "cat " << cat << endl;
    removeHat(cat + 1); // Рекурсивный вызов
  } else
    cout << "VOOM!!!" << endl;
}

int main() {
```

[1] Спасибо Крису С. Мэтсону (Kris C.Matson) за предложенный пример.

```
  removeHat('A');
} ///:~
```

Как видно из этого листинга, до тех пор, пока cat остается меньше 'Z', функция removeHat() вызывается *из функции* removeHat(), то есть происходит рекурсивный вызов. При каждом вызове removeHat() аргумент увеличивается на 1 по сравнению с текущим значением cat, поэтому аргумент растет до заданного порога.

Рекурсия часто требуется при решении задач, сложность которых неизвестна заранее, поскольку это решение не ограничивается конкретным «размером» решения, — функция просто продолжает рекурсивно вызываться до тех пор, пока задача не будет решена.

Знакомство с операторами

Операторы можно рассматривать как особую разновидность функций (вскоре вы узнаете, что при перегрузке операторов они интерпретируются именно так). Оператор получает один или несколько аргументов и генерирует новое значение. Представление аргументов несколько отличается от обычного вызова функций, но суть остается той же.

Из предыдущего опыта программирования вы уже знакомы со многими операторами. Операторы сложения (+), вычитания и унарного минуса (-), умножения (*), деления (/) и присваивания (=) имеют одинаковый смысл практически во всех языках программирования. Полный перечень операторов приводится далее в этой главе.

Приоритет

Приоритет операторов определяет порядок обработки выражений, содержащих несколько операторов. В языках C и C++ этот порядок определяется четкими правилами. Проще всего запомнить, что умножение и деление выполняются перед сложением и вычитанием. Если после этого выражение не станет очевидным для вас, вероятно, оно останется непонятным и для читателей программы, поэтому порядок обработки лучше задать явно при помощи круглых скобок. Например, рассмотрим выражение:

```
A = X + Y - 2/2 + Z;
```

Это выражение существенно отличается по смыслу от похожего выражения, в котором используется группировка с круглыми скобками:

```
A = X + (Y - 2)/(2 + Z);
```

Попробуйте вычислить результат при X = 1, Y = 2 и Z = 3.

Инкремент и декремент

В языке C (а следовательно, и в C++) имеется множество сокращенных обозначений. Сокращения заметно упрощают ввод программы, но иногда еще сильнее затрудняют ее чтение. Вероятно, проектировщики языка C полагали, что в сложном фрагменте проще разобраться, если при этом не приходится пробегать глазами всю область вывода.

Одним из удобных сокращений являются операторы инкремента и декремента. Они часто применяются для изменения счетчиков, управляющих количеством итераций цикла.

Оператор декремента -- означает «уменьшить на единицу», а оператор инкремента ++ означает «увеличить на единицу». Например, если переменная А относится к типу int, то выражение ++А эквивалентно команде A = A + 1. Результатом выполнения операторов инкремента и декремента является значение переменной. Если оператор находится перед именем переменной (то есть ++А), то он сначала выполняет операцию, а затем возвращает новое значение. Если же оператор находится после имени переменной (то есть А++), то оператор сначала выдает текущее значение, а затем выполняет операцию. Пример:

```
//: C03:AutoIncrement.cpp
// Использование операторов инкремента и декремента.
#include <iostream>
using namespace std;

int main() {
  int i = 0;
  int j = 0;
  cout << ++i << endl; // Префиксное увеличение
  cout << j++ << endl; // Постфиксное увеличение
  cout << --i << endl; // Префиксное уменьшение
  cout << j-- << endl; // Постфиксное уменьшение
} ///:~
```

Если вы еще не догадались, как появилось название языка С++, то теперь все становится ясно — оно означает «следующий шаг после С».

Знакомство с типами данных

Типы данных определяют правила использования памяти в написанных вами программах. Указывая тип данных, вы тем самым сообщаете компилятору, какой размер имеет созданный блок памяти и как с ним работать.

Типы данных делятся на *встроенные* и *абстрактные*. Встроенные типы данных изначально поддерживаются компилятором, то есть жестко закодированы в нем. В языках С и С++ наборы встроенных типов практически идентичны. С другой стороны, абстрактные типы данных определяются программистами в виде классов (также встречается термин «пользовательские типы»). Компилятор уже умеет работать со встроенными типами данных в момент запуска. Что касается абстрактных типов, то компилятор «учится» работать с ними, читая заголовочные файлы с объявлениями классов (об этом будет рассказано в следующих главах).

Встроенные типы

В стандартной спецификации встроенных типов языка С, унаследованной языком С++, не сказано, сколько битов должен занимать тот или иной встроенный тип. Вместо этого задаются минимальное и максимальное значения, которые должны обеспечиваться этим типом. Если машина использует двоичное представление чисел, максимальное значение напрямую определяет минимальное количество битов, необходимое для хранения значения. Но если, например, машина исполь-

зует двоично-десятичное представление чисел, то объем памяти для хранения максимального числа каждого типа данных будет совершенно иным. Минимальное и максимальное значения для разных типов данных определяются в системных заголовочных файлах limits.h и float.h (в C++ вместо них обычно включаются заголовки <climits> и <cfloat>).

В C и C++ поддерживаются четыре основных встроенных типа данных, описанные далее для машин с двоичным представлением. Тип char предназначен для хранения символов и занимает минимум 8 бит (один байт) памяти, хотя может занимать больше. Тип int содержит целые числа и занимает минимум два байта. Типы float и double содержат вещественные числа, обычно в вещественном формате IEEE. Тип float предназначен для хранения вещественных чисел с одинарной точностью, а тип double предназначен для чисел с двойной точностью.

Как упоминалось ранее, переменные могут определяться в любой точке текущего блока, причем определение может быть совмещено с инициализацией. Примеры определения переменных для четырех основных типов данных:

```
//: C03:Basic.cpp
// Определения переменных четырех встроенных типов данных
// в C и C++

int main() {
  // Определение без инициализации:
  char protein;
  int carbohydrates;
  float fiber;
  double fat;
  // Определение, совмещенное с инициализацией:
  char pizza = 'A', pop = 'Z';
  int dongdings = 100, twinkles = 150,
    heehos = 200;
  float chocolate = 3.14159;
  // Экспоненциальная запись:
  double fudge_ripple = 6e-4;
} ///:~
```

В первой части программы переменные четырех встроенных типов определяются без инициализации. Согласно стандарту, содержимое неинициализированных переменных не определено (обычно в них находится «мусор», то есть ранее использованное содержимое памяти). Во второй части программы определение переменных совмещено с инициализацией. По возможности рекомендуется задавать начальное значение переменной в точке определения. Обратите внимание на экспоненциальную запись в константе 6e-4 — она означает «10 в минус четвертой степени».

Ключевые слова bool, true и false

До того как булев тип был включен в стандарт C++, программисты использовали разные приемы при работе с логическими значениями. Это создавало проблемы с переносимостью программ и приводило к появлению нетривиальных ошибок.

В стандарте C++ тип bool имеет два допустимых состояния, выраженных встроенными константами true (целое число 1) и false (целое число 0). Все три имени являются ключевыми словами. Кроме того, некоторые элементы языка были адаптированы для новых типов:

Элемент	Использование с bool
&& || !	Получают аргументы типа bool и выдают результат типа bool
< > <= >= == !=	Выдают результат типа bool
if, for, while, do	Условные выражения преобразуются в значения типа bool
?:	Первый операнд преобразуется к значению bool

Поскольку во многих существующих программах логические значения представляются типом int, компилятор производит автоматическое приведение от int к bool (ненулевые значения интерпретируются как true, а нулевые значения — как false). В идеальном случае компилятор выдает предупреждение с рекомендацией о внесении исправлений.

Перевод флага в состояние true оператором ++ считается проявлением «плохого стиля программирования». Такая возможность существует, но она считается *устаревшей*, а это означает, что когда-нибудь в будущем она будет запрещена. Дело в том, что операция сопровождается автоматическим приведением типа от bool к int, увеличением значения (возможно, с выходом за пределы обычных значений типа bool — нуля и единицы) и его возвратом к прежнему значению.

Указатели, которые будут рассматриваться позднее в этой главе, при необходимости также автоматически приводятся к типу bool.

Спецификаторы

Спецификаторы изменяют смысл основных встроенных типов и значительно расширяют их набор. Существуют четыре спецификатора: long, short, signed и unsigned.

Спецификаторы long и short изменяют минимальное и максимальное значения, поддерживаемые типом данных. Простой тип int должен быть по крайней мере не меньше short int. Целочисленные типы образуют следующий ряд: short int, int, long int. Теоретически размеры всех типов могут совпадать, если выполняются требования к минимальному и максимальному значениям. Например, на компьютерах с 64-разрядными словами размер всех трех типов может быть равен 64 бит.

Аналогичный ряд для вещественных типов выглядит так: float, double, long double. Тип long float не поддерживается. Спецификатор short для вещественных типов не определен.

Спецификаторы signed и unsigned указывают компилятору, как должен использоваться знаковый бит в целых числах и символах (вещественные числа всегда имеют знак). В беззнаковых (unsigned) числах знак не учитывается, освободившийся бит может пригодиться для представления данных, поэтому в беззнаковых переменных могут храниться вдвое большие положительные значения, нежели в знаковых (signed). По умолчанию задействуется спецификатор signed, который требуется только для типа char (по умолчанию тип char может быть как знаковым, так и беззнаковым). Объявление типа signed char обеспечивает принудительное использование знакового бита.

В следующем примере выводятся размеры разных типов данных в битах. При этом используется оператор sizeof, описанный далее в этой главе.

```
//: C03:Specify.cpp
// Использование спецификаторов
```

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

int main() {
  char c;
  unsigned char cu;
  int i;
  unsigned int iu;
  short int is;
  short iis; // Эквивалент short int
  unsigned short int isu;
  unsigned short iisu;
  long int il;
  long iil;  // Эквивалент long int
  unsigned long int ilu;
  unsigned long iilu;
  float f;
  double d;
  long double ld;
  cout
    << "\n char= " << sizeof(c)
    << "\n unsigned char = " << sizeof(cu)
    << "\n int = " << sizeof(i)
    << "\n unsigned int = " << sizeof(iu)
    << "\n short = " << sizeof(is)
    << "\n unsigned short = " << sizeof(isu)
    << "\n long = " << sizeof(il)
    << "\n unsigned long = " << sizeof(ilu)
    << "\n float = " << sizeof(f)
    << "\n double = " << sizeof(d)
    << "\n long double = " << sizeof(ld)
    << endl;
} ///:~
```

Результаты выполнения этой программы зависят от конкретных компьютера, операционной системы и компилятора, поскольку (как упоминалось ранее), гарантируются лишь минимальное и максимальное значения каждого типа, указанные в стандарте.

Как видно из приведенного примера, при модификации типа int спецификаторами short и long ключевое слово int не обязательно.

Знакомство с указателями

При запуске программа сначала загружается (как правило, с жесткого диска) в память компьютера. Следовательно, все элементы программы находятся в том или ином месте памяти. Традиционно память рассматривается как последовательный набор ячеек, которые обычно называются *байтами* и занимают 8 бит, но в действительности структура памяти зависит от архитектуры и *размера машинного слова* данного компьютера. Каждая ячейка памяти однозначно определяется ее *адресом*. В контексте настоящей темы будем считать, что последовательность адресов начинается с нуля и увеличивается до величины, соответствующей объему физической памяти на данном компьютере.

Во время работы программа находится в памяти, поэтому каждый элемент программы обладает собственным адресом. Начнем с рассмотрения простой программы:

```
//: C03:YourPets1.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j, k;
} ///:~
```

Каждый элемент программы в процессе выполнения занимает некоторый участок памяти. Даже функции занимают память. Как вы вскоре убедитесь, занимаемая область памяти определяется типами элементов и способом их определения.

В С и C++ имеется оператор &, предназначенный для получения адреса элемента. Все, что для этого нужно, — поставить префикс & перед идентификатором; оператор вернет адрес этого идентификатора. Следующий вариант программы YourPets.cpp выводит адреса всех своих элементов:

```
//: C03:YourPets2.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j, k;
  cout << "f(): " << (long)&f << endl;
  cout << "dog: " << (long)&dog << endl;
  cout << "cat: " << (long)&cat << endl;
  cout << "bird: " << (long)&bird << endl;
  cout << "fish: " << (long)&fish << endl;
  cout << "i: " << (long)&i << endl;
  cout << "j: " << (long)&j << endl;
  cout << "k: " << (long)&k << endl;
} ///:~
```

Конструкция (long) обозначает *приведение типа*. Другими словами, она говорит: «Данное значение должно интерпретироваться не со своим обычным типом, а как величина типа long». В принципе, программа будет работать и без приведения, но в этом случае адреса выводятся в шестнадцатеричном виде. Приведение к типу long делает выходные данные чуть более наглядными.

Конкретный результат выполнения этой программы зависит от компьютера, операционной системы и других факторов, но результаты в любом случае будут довольно интересными. При одном из запусков на компьютере автора программа вывела следующие данные:

```
f(): 4198736
dog: 4323632
cat: 4323636
```

```
bird: 4323640
fish: 4323644
i: 6684160
j: 6684156
k: 6684152
```

Видно, что переменные, определенные внутри функции main(), и переменные, определенные за ее пределами, находятся в разных блоках памяти; вы поймете, почему это происходит, когда узнаете больше о C++. Кроме того, похоже, что функция f() находится в отдельном блоке; в памяти код обычно хранится отдельно от данных.

Также следует заметить, что переменные, определяемые в программе по соседству, занимают смежные участки памяти — они отделяются друг от друга количеством байтов, необходимым для их хранения. В программе используется только тип int; переменная cat удалена на четыре байта от dog, переменная bird — на четыре байта от cat и т. д. Из этого можно сделать вывод, что на данном компьютере переменная int занимает четыре байта.

Но для чего еще могут пригодиться адреса, кроме экспериментов со структурой распределения памяти? Самое важное, что можно сделать с адресом, — сохранить его в другой переменной для последующего использования. В C++ существует специальный тип переменных для хранения адресов. Такие переменные называются *указателями*.

Оператор определения указателя выглядит так же, как оператор умножения (*). Компилятор различает эти два оператора по контексту, в котором они используются.

При определении указателя необходимо задать тип переменной, на которую он указывает. Сначала задается имя типа, но затем перед идентификатором переменной следует звездочка — признак указателя. Например, указатель на int определяется так:

```
int* ip; // ip указывает на переменную типа int
```

Ассоциация * с типом логично выглядит и легко читается, но иногда вызывает недоразумения. Возникает ощущение, что «указатель на int» — это самостоятельный тип. Однако для int или другого типа данных возможны объявления вида:

```
int a, b, c;
```

Тогда как для указателей вам лишь *хотелось бы* использовать запись:

```
int *ipa, ipb, ipc;
```

Синтаксис C (а соответственно, и синтаксис C++) не поддерживает эти вполне разумные выражения. В приведенных выше определениях указателем является только идентификатор ipa, а ipb и ipc — обычные переменные типа int (можно сказать, что * связывается с идентификатором, а не с типом). Соответственно, лучше ограничиться одним определением на строку; в этом случае синтаксис выглядит вполне разумно и предотвращает путаницу:

```
int* ipa;
int* ipb;
int* ipc;
```

Согласно общим рекомендациям для программистов C++ переменные должны инициализироваться в точке определения, для чего данная форма подходит лучше. Например, переменные из этого фрагмента не инициализируются никакой

конкретной величиной; в них хранится «мусор». Гораздо лучше использовать запись вида:

```
int a = 47;
int* ipa = &a;
```

Теперь обе переменные a и ipa инициализированы, а в ipa хранится адрес a.

Простейшее применение инициализированных указателей — модификация значения, на которое он ссылается. Чтобы обратиться к значению через указатель, следует *разыменовать* указатель тем же оператором *, который использовался для его определения:

```
*ipa = 100;
```

Теперь переменная a содержит значение 100 вместо 47.

Итак, основные принципы работы указателей понятны: указатели предназначены для хранения адресов, при помощи которых можно изменить исходную переменную. Остается ответить на вопрос — зачем использовать переменную как «посредника» для модификации другой переменной?

В этом вводном описании можно выделить две широкие области применения указателей:

- указатели могут использоваться для изменения «внешних объектов» внутри функций (вероятно, этот способ применения указателей является наиболее распространенным, поэтому он подробно рассматривается далее);
- многие нетривиальные приемы программирования основаны на работе с указателями (примеры такого рода неоднократно встречаются в книге).

Модификация внешних объектов

Функции, получающие аргументы при вызове, обычно создают внутренние копии этих аргументов. Этот механизм называется *передачей по значению*. Следующая программа показывает, что происходит при передаче по значению:

```
//: C03:PassByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
} ///:~
```

Переменная a является *локальной* для функции f(), то есть существует только на протяжении вызова f(). Так как переменная a передается в аргументе функции, она инициализируется значением, переданным при вызове. Внутри функции main() при вызове передается переменная x, равная 47. Это значение копируется в переменную a при вызове f().

При запуске программы выводится следующий результат:

```
x = 47
a = 47
a = 5
x = 47
```

В исходном состоянии переменная x равна 47. При вызове f() выделяется временная область памяти для хранения переменной a во время выполнения функции. Переменная a инициализируется копированием значения x, что подтверждается второй строкой вывода. Конечно, мы можем присвоить a новое значение и убедиться в том, что оно действительно изменилось (третья строка). Но после завершения функции f() временная память, выделенная для a, освобождается, и мы видим, что переменная x сохранила прежнее значение. Единственная связь между x и a существовала в тот момент, когда значение x было скопировано в a.

Вне функции f() переменная x является *внешним объектом* (авторский термин). Изменение локальной переменной не отражается на внешнем объекте, что вполне естественно, поскольку они занимают разные области памяти. Но что делать, если вы *хотите* изменить внешний объект? Именно здесь вам пригодятся указатели. В определенном смысле указатель является «псевдонимом» для другой переменной. Таким образом, если вместо обычного значения передать функции *указатель*, фактически передается псевдоним внешнего объекта, что позволяет функции модифицировать этот внешний объект:

```
//: C03:PassAddress.cpp
#include <iostream>
using namespace std;

void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
} ///:~
```

Функция f() получает в аргументе указатель и разыменовывает его в момент присваивания, что позволяет изменить внешний объект x. Результат выполнения программы:

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

Обратите внимание: значение переменной p в точности совпадает с адресом x, то есть указатель p действительно ссылается на x. А если и это выглядит недоста-

точно убедительно, мы видим, что после присваивания 5 по разыменованному указателю p переменная x тоже становится равной 5.

Итак, передача указателя функции позволяет ей модифицировать внешние объекты. Далее будет рассмотрено множество других применений указателей, но именно это применение можно считать основным, а возможно, и самым распространенным.

Знакомство со ссылками C++

Указатели в C и C++ работают примерно одинаково, но в C++ появился новый способ передачи адресов функциям. Механизм *передачи по ссылке* также поддерживается в ряде других языков программирования — он не был изобретен специально для C++.

При первом знакомстве со ссылками может показаться, что они являются чем-то лишним, а программы можно писать и без ссылок. Как правило, это действительно так, за исключением нескольких важных областей, о которых вы узнаете далее, когда ссылки будут рассматриваться более подробно. Пока достаточно сказать, что общий принцип их работы напоминает предыдущий пример с указателями: функции передается адрес аргумента при помощи ссылки. Различие между ссылками и указателями заключается в том, что *вызов* функции с передачей аргументов по ссылке с точки зрения синтаксиса выглядит более четко и наглядно, нежели вызов с передачей указателей (именно эти синтаксические различия делают ссылки незаменимыми в некоторых ситуациях). Если изменить пример PassAddress.cpp с помощью ссылок, различия нетрудно заметить при вызове функции из main():

```
//: C03:PassReference.cpp
#include <iostream>
using namespace std;

void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x); // Выглядит как передача по значению,
        // но в действительности аргумент передается по ссылке
  cout << "x = " << x << endl;
} ///:~
```

В списке аргументов f() вместо указателя (int*) передается ссылка (int&). Внутри f() достаточно сослаться на переменную r (которая в случае передачи указателя содержала бы адрес), *чтобы получить значение переменной, на которую ссылается* r. Новое значение, присваиваемое r, в действительности присваивается той переменной, на которую ссылается r. Более того, хранящийся в r адрес можно получить только при помощи оператора &.

В функции main() последствия от применения ссылок проявляются при вызове f(), который теперь принимает вид f(x). Хотя по внешнему виду он ничем не отли-

чается от обычной передачи по значению, вместо копии значения передается его адрес. Результат выполнения программы выглядит так:

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

Следовательно, передача по ссылке позволяет функциям изменять внешние объекты, как и передача указателей. Также следует заметить, что ссылка маскирует факт передачи адреса; этот вопрос будет рассмотрен позднее. Пока можно считать, что ссылки всего лишь являются другим, более удобным вариантом синтаксиса (так называемый «синтаксический сахар»), который делает то же, что и передача указателей, — позволяет функциям изменять внешние объекты.

Указатели и ссылки как модификаторы

До настоящего момента вы успели познакомиться с основными типами данных char, int, float и double и спецификаторами signed, unsigned, short и long, которые могут использоваться почти со всеми встроенными типами данных. Теперь к ним добавились указатели и ссылки, «ортогональные» по отношению к встроенным типам данных и спецификаторам, поэтому количество возможных комбинаций возрастает втрое:

```
//: C03:AllDefinitions.cpp
// Все возможные комбинации встроенных типов данных.
// спецификаторов, обозначений и ссылок
#include <iostream>
using namespace std;

void f1(char c, int i, float f, double d);
void f2(short int si, long int li, long double ld);
void f3(unsigned char uc, unsigned int ui,
  unsigned short int usi, unsigned long int uli);
void f4(char* cp, int* ip, float* fp, double* dp);
void f5(short int* sip, long int* lip,
  long double* ldp);
void f6(unsigned char* ucp, unsigned int* uip,
  unsigned short int* usip,
  unsigned long int* ulip);
void f7(char& cr, int& ir, float& fr, double& dr);
void f8(short int& sir, long int& lir,
  long double& ldr);
void f9(unsigned char& ucr, unsigned int& uir,
  unsigned short int& usir, unsigned long int& ulir);

int main() {} ///:~
```

Указатели и ссылки также могут применяться для передачи объектов функциям и их возврата; об этом будет рассказано в одной из следующих глав.

Существует еще один тип, который может использоваться с указателями, — тип void. Если в программе определяется указатель типа void*, это означает, что ему может быть присвоен адрес произвольного типа (тогда как указателю int* можно присвоить только адрес переменной типа int). Пример:

```
//: C03:VoidPointer.cpp
int main() {
void* vp;
char c;
int i;
float f;
double d;
// Указателю void* можно присвоить адрес ЛЮБОГО типа:
vp = &c;
vp = &i;
vp = &f;
vp = &d;
} ///:~
```

После присваивания void* вся информация о типе, связанном с данным адресом, теряется. Это означает, что перед использованием указателя его необходимо привести к правильному типу:

```
//: C03:CastFromVoidPointer.cpp
int main() {
int i = 99;
void* vp = &i;
// Разыменование указателя на void запрещено:
// *vp = 3; // Ошибка компиляции
// Перед разыменованием необходимо выполнить приведение к типу int:
*((int*)vp) = 3;
} ///:~
```

Приведение (int*)vp сообщает компилятору, что тип void* при разыменовании должен интерпретироваться как int*. Синтаксис выглядит уродливо, но хуже другое — указатели void* пробивают брешь в системе безопасности типов языка. Иначе говоря, они допускают и даже поощряют интерпретацию данных одного типа как данные другого типа. В приведенном выше примере vp приводится к int*, то есть int интерпретируется как int, но ничто не помешает привести указатель к char* или double*. В этом случае изменится объем памяти, занимаемой переменной, скорее всего, с аварийным завершением программы. В общем случае использовать указатели void не рекомендуется, за исключением редких случаев, смысл которых станет понятен лишь к концу книги.

Ссылки на void недопустимы по причинам, объясняемым в главе 11.

Видимость

Правила видимости определяют, в какой области программы переменная остается действительной, где она создается или уничтожается (то есть выходит из области видимости). Область видимости переменной продолжается от точки определения до первой закрывающей фигурной скобки, парной по отношению к ближайшей открывающей фигурной скобке перед определением переменной. Таким образом, область видимости определяется «ближайшей» парой фигурных скобок. Пример:

```
//: C03:Scope.cpp
// Видимость переменных
int main() {
int scp1;
// Здесь начинается область видимости переменной scp1
{
```

```
// Переменная scp1 остается видимой
// ...
int scp2;
// Здесь начинается область видимости переменной scp2
// ...
{
// Переменные scp1 и scp2 остаются видимыми
// ...
int scp3;
// Переменные scp1, scp2 и scp3 остаются видимыми
// ...
} // <-- здесь уничтожается переменная scp3
// Переменная scp3 недоступна
// Переменные scp1 и scp2 остаются видимыми
// ...
} // <-- здесь уничтожается переменная scp2
// Переменные scp3 и scp2 недоступны
// Переменная scp1 остается видимой
// ...
} // <-- здесь уничтожается переменная scp1
///:~
```

Приведенный пример показывает, где в программе переменные остаются видимыми, а где они недоступны (то есть *выходят из области видимости*). Переменные могут использоваться только внутри своей области видимости. Области видимости могут быть вложенными, что обозначается парами фигурных скобок внутри других пар фигурных скобок. Во внутренних областях видимости доступны переменные той области, в которую входит данная область. Так, в приведенном выше примере переменная scp1 доступна во всех вложенных областях видимости, тогда как переменная scp3 доступна только во внутренней области.

Определение переменных непосредственно перед использованием

Как уже упоминалось в этой главе, между C и C++ существуют значительные различия относительно определения переменных. Оба языка требуют, чтобы переменные определялись перед использованием, но C (а также многие другие процедурные языки) заставляет определять все переменные в начале области видимости, чтобы при создании блока компилятор мог выделить память для этих переменных.

При чтении программ C первое, что вы обычно видите при входе в новую область видимости, — блок определений переменных. Определение всех переменных в начале блока накладывает на программиста ограничения, обусловленные деталями реализации языка. Большинство программистов не знают заранее всех переменных, которые им понадобятся, поэтому им приходится часто возвращаться к началу блока и вставлять новые переменные. Во-первых, это неудобно; во-вторых, подобные переходы становятся причиной ошибок. Определения переменных обычно не сообщают сколько-нибудь полезной информации читателю программы, а на деле лишь сбивают его с толку, потому что они расположены далеко от места использования.

Язык C++ (но не C) позволяет определять переменные в произвольной точке области видимости, то есть переменная может определяться непосредственно перед

использованием. Кроме того, возможна инициализация переменных в точке определения, что предотвращает некоторые виды ошибок. Подобный способ определения переменных значительно упрощает чтение программы и уменьшает число ошибок из-за частых переходов от начала области к месту применения. Программа становится более наглядной, потому что переменные определяются в контексте их использования. Это особенно важно при совмещении определения с инициализацией — само использование переменной поясняет смысл начального значения.

Переменные также могут определяться в управляющих выражениях циклов for и while, в условиях команд if и в критериях выбора команды switch. Рассмотрим пример определения переменных непосредственно перед использованием:

```
//: C03:OnTheFly.cpp
// Определение переменных перед использованием
#include <iostream>
using namespace std;

int main() {
// ...
{ // Начало новой области видимости
int q = 0; // C требует, чтобы определения находились в этой точке
// ...
// Определение переменных перед использованием:
for(int i = 0; i < 100; i++) {
q++; // q принадлежит к внешней области видимости
// Определение в конце области видимости:
int p = 12;
}
int p = 1; // Другая переменная p
} // Завершение области видимости, содержащей q и внешнюю переменную p
cout << "Type characters:" << endl;
while(char c = cin.get() != 'q') {
cout << c << " wasn't it" << endl;
if(char x = c == 'a' || c == 'b')
cout << "You typed a or b" << endl;
else
cout << "You typed " << x << endl;
}
cout << "Type A, B, or C" << endl;
switch(int i = cin.get()) {
case 'A': cout << "Snap" << endl; break;
case 'B': cout << "Crackle" << endl; break;
case 'C': cout << "Pop" << endl; break;
default: cout << "Not A, B or C!" << endl;
}
} ///:~
```

Во внутренней области видимости переменная p определяется перед концом области видимости. На самом деле такое объявление абсолютно бесполезно, но оно показывает, что переменные действительно могут определяться в любой точке области видимости. То же самое можно сказать о переменной p из внешней области видимости.

Определение i в управляющем выражении цикла for — пример определения переменной *прямо* в момент использования (такое возможно только в C++). Область видимости i ограничивается данным циклом for, поэтому ничто не помешает заново определить переменную с именем i для следующего цикла for. Это одна из

удобных и распространенных идиом C++: имя i традиционно используется для счетчиков циклов, вам не придется изобретать новые имена.

Хотя в этом примере также продемонстрировано определение переменных в командах while, if и switch, подобные случаи встречаются значительно реже определения в циклах for. Возможно, это объясняется ограничениями синтаксиса. Например, выражение не может содержать круглых скобок, то есть следующая конструкция невозможна:

```
while((char c = cin.get()) != 'q')
```

Дополнительные круглые скобки выглядят вполне невинно и удобно, а без них результат выглядит не так, как можно было бы ожидать. Проблемы возникают из-за того, что оператор != обладает более высоким приоритетом, чем =, поэтому переменной char c в конечном счете присваивается значение bool, приведенное к типу char. На многих терминалах это значение выводится в виде улыбающейся рожицы.

Для полноты картины следует помнить о возможности определения переменных в командах while, if и switch, но на практике такие определения обычно встречаются только в циклах for (и притом довольно часто).

Распределение памяти

При создании переменных в распоряжении программиста имеются различные возможности для указания срока жизни переменной, способа выделения памяти и режима интерпретации переменной компилятором.

Глобальные переменные

Глобальные переменные определяются вне тел функций и доступны для всех частей программы (даже для кода других файлов). Глобальные переменные не подчиняются правилам области видимости и остаются доступными всегда (то есть глобальные переменные существуют до завершения программы). Если глобальная переменная из одного файла объявляется с ключевым словом extern в другом файле, то она становится доступной для второго файла. Пример использования глобальных переменных:

```
//: C03:Global.cpp
//{L} Global2
// Использование глобальных переменных
#include <iostream>
using namespace std;

int globe;
void func();
int main() {
globe = 12;
cout << globe << endl;
func(); // Модификация globe
cout << globe << endl;
} ///:~
```

А вот файл, который обращается к globe как к глобальной переменной:

```
//: C03:Global2.cpp {O}
// Обращение к внешней глобальной переменной
```

```
extern int globe:
// (Ссылка разрешается компоновщиком)
void func() {
globe = 47:
} ///:~
```

Память для переменной globe выделяется при ее определении в файле Global.cpp; далее к этой же переменной обращается программа Global2.cpp. Поскольку код Global2.cpp компилируется отдельно от кода Global.cpp, компилятору необходимо сообщить о существовании переменной объявлением

```
extern int globe:
```

Запуск программы наглядно показывает, что вызов func() действительно изменяет один глобальный экземпляр globe.

В файле Global.cpp встречается специальный комментарий, структуру которого придумал автор:

```
//{L} Global2
```

Он означает, что при создании программы требуется компоновка с объектным файлом Global2 (расширение отсутствует, потому что в разных системах используются разные расширения объектных файлов). В файле Global2.cpp первая строка содержит специальный комментарий {0}, который означает: «Данный файл не должен использоваться для построения исполняемой программы; он компилируется для компоновки с другим исполняемым файлом». Программа ExtractCode.cpp (см. второй том книги) читает эти комментарии и строит правильный make-файл для нормальной компиляции программы (make-файлы будут рассматриваться в конце этой главы).

Локальные переменные

Локальные переменные принадлежат некоторой области видимости; они «локальны» по отношению к соответствующей функции. Такие переменные часто называются *автоматическими*, потому что они автоматически создаются при входе в область видимости и автоматически уничтожаются при выходе из нее. На это обстоятельство можно указать явно при помощи ключевого слова auto, но локальные переменные являются автоматическими по умолчанию, поэтому на практике ключевое слово auto обязательным не является.

Регистровые переменные

Регистровые переменные являются разновидностью локальных переменных. Ключевое слово register указывает компилятору: «Обращения к этой переменной должны выполняться как можно быстрее». Способ увеличения скорости доступа зависит от реализации, но, как следует из названия, задача часто решается хранением переменной в регистре процессора. Ключевое слово register не дает гарантии, что переменная действительно будет размещена в регистре или что скорость доступа к ней повысится. Оно всего лишь дает рекомендации компилятору.

Использование регистровых переменных подчиняется некоторым ограничениям. Например, программа не может получить или вычислить адрес регистровой переменной. Регистровые переменные объявляются только в блоках (глобальные или статические регистровые переменные запрещены). Тем не менее регистровые переменные могут передаваться в формальных аргументах функций, то есть в списках аргументов.

Как правило, давать советы оптимизатору не рекомендуется, поскольку он обычно справляется со своей задачей лучше вас. Соответственно, ключевого слова register лучше избегать.

Статические переменные

Ключевое слово static имеет несколько разных значений. Обычно переменные, локальные для некоторой функции, исчезают в конце области видимости этой функции. При повторном вызове функции память для переменных выделяется повторно, а переменные инициализируются заново. Если значение переменной должно сохраняться на протяжении жизненного цикла программы, определите локальную переменную с ключевым словом static и присвойте ей начальное значение. Тогда инициализация выполняется только при первом вызове функции, а последующие значения переменной сохраняются между вызовами. В результате функция «запоминает» значения статических переменных до следующего вызова.

Спрашивается, почему бы не воспользоваться для этой цели глобальной переменной? Прелесть статических переменных заключается в том, что они остаются недоступными вне области видимости функции; тем самым предотвращается их случайная модификация в других местах программы. Таким образом локализуются ошибки.

Пример использования статической переменной:

```
//: C03:Static.cpp
// Использование статической переменной в функции
#include <iostream>
using namespace std;

void func() {
static int i = 0;
cout << "i = " << ++i << endl;
}

int main() {
for(int x = 0; x < 10; x++)
func();
} ///:~
```

При каждом вызове func() в цикле for выводится новое значение. Без ключевого слова static всегда выводилось бы одно и то же значение 1.

Второй смысл ключевого слова static связан с первым разве что в трактовке «недоступный за пределами заданной области». Применительно к имени функции или переменной, не принадлежащей ни одной функции, static означает: «Имя недоступно за пределами текущего файла». То есть имя функции или переменной локально по отношению к файлу; говорят, что оно имеет *файловую область видимости*. Для примера попробуйте откомпилировать и скомпоновать следующие два файла — это приведет к ошибке компоновки:

```
//: C03:FileStatic.cpp
// Демонстрация файловой области видимости.
// При попытке откомпилировать и скомпоновать этот файл с FileStatic2.cpp
// произойдет ошибка компоновки.

// Файловая видимость означает, что переменная доступна только
// в этом файле:
```

```
static int fs;

int main() {
fs = 1;
} ///:~
```

Хотя в следующем файле объявлено о существовании внешней (extern) переменной fs, компоновщик не находит переменную, поскольку она была объявлена статической в файле FileStatic.cpp.

```
//: C03:FileStatic2.cpp {O}
// Попытка использования ссылки на fs
extern int fs;
void func() {
fs = 100;
} ///:~
```

Спецификатор static также может использоваться внутри классов. Разъяснения откладываются до того момента, когда вы научитесь создавать классы.

Внешние переменные

Ключевое слово extern было кратко описано и продемонстрировано в предыдущем примере. Оно сообщает компилятору о существовании переменной или функции, даже если компилятор еще не обнаружил эту переменную или функцию в текущем компилируемом файле. Внешняя переменная или функция определяется либо в другом файле, либо далее в текущем файле. Пример второго случая:

```
//: C03:Forward.cpp
// Опережающие объявления функций и данных
#include <iostream>
using namespace std;

// Объявления не являются внешними в нормальном смысле.
// но компилятор должен знать о том, что они где-то существуют:
extern int i;
extern void func();
int main() {
i = 0;
func();
}
int i; // Определение данных
void func() {
i++;
cout << i;
} ///:~
```

Встречая объявление extern int i, компилятор знает, что определение i должно существовать где-то в виде глобальной переменной. Когда компилятор обнаруживает определение i, а другие видимые объявления для этого имени отсутствуют; компилятор делает вывод, что обнаружена та самая переменная i, которая была объявлена ранее в этом файле. Если включить в определение i ключевое слово static, компилятор сделает вывод, что переменная i определяется глобальной (посредством ключевого слова extern), но при этом обладает файловой видимостью (static). В результате произойдет ошибка компиляции.

Связывание

Чтобы понять, как работают программы C и C++, необходимо знать о принципах *связывания*. В исполняемой программе каждому идентификатору соответствует область памяти, содержащая значение переменной или откомпилированное тело функции. Тип связывания описывает эту область памяти с точки зрения компоновщика. Существуют два типа связывания: *внутреннее* и *внешнее*.

Внутреннее связывание означает, что память выделяется для представления идентификатора только в контексте компилируемого файла. Другие файлы могут использовать тот же идентификатор для внутреннего связывания или для глобальной переменной, и это не вызовет конфликтов компоновки — для каждого идентификатора создается отдельная область памяти. Признаком внутреннего связывания в C и C++ является ключевое слово static.

Внешнее связывание означает, что для представления идентификатора во всех компилируемых файлах выделяется единая область памяти. Эта память выделяется только один раз, и компоновщик должен разрешать остальные ссылки на нее. Глобальные переменные и имена функций определяются внешним связыванием. При обращении к ним из других файлов требуется объявление с ключевым словом extern. Внешнее связывание используется по умолчанию для переменных, определенных вне всех функций (кроме констант в C++) и определений функций. Ключевое слово static позволяет специально определить для них внутреннее связывание. Чтобы указать, что для идентификатора требуется внешнее связывание, определите его с ключевым словом extern. Определение переменных и функций с ключевым словом extern не нужно в языке C, но в C++ оно иногда бывает необходимым для констант.

Автоматические (локальные) переменные временно (на время вызова функции) существуют в стеке. Компоновщик ничего не знает об автоматических переменных, поэтому для них *связывание не определено*.

Константы

В прежнем языке C (до появления стандарта) для объявления констант приходилось использовать препроцессор:

```
#define PI 3.14159
```

Все ссылки на идентификатор PI заменялись препроцессором на значение 3.14159. Кстати, этот способ по-прежнему доступен в C и C++.

Однако константы, созданные при помощи препроцессора, выходят из-под контроля компилятора. Для имени PI не выполняется проверка типов, программа не может получить адрес PI, передать ссылку или указатель на PI. Кроме того, PI не может быть переменной пользовательского типа. Наконец, имя PI продолжает действовать от точки определения до конца файла; препроцессор не поддерживает концепцию видимости.

В C++ появилась концепция именованных констант, которые похожи на переменные во всем, кроме одного: их значения запрещено изменять. Модификатор const сообщает компилятору, что имя представляет константу. При объявлении константы могут задействоваться любые типы данных — как пользовательские, так и встроенные. Если определить в программе константу, а потом попытаться изменить ее, компилятор выдает сообщение об ошибке.

Тип константы должен быть указан при ее определении:

```
const int x = 10;
```

В стандартных версиях C и C++ разрешена передача именованных констант в списке аргументов, даже если соответствующий аргумент представляет собой указатель или ссылку (то есть получение адреса константы не запрещено). Константы, как и обычные переменные, обладают видимостью, поэтому константу можно «спрятать» внутри функции и быть уверенным в том, что ее имя останется невидимым в других частях программы.

Константы были позаимствованы из C++ и включены в стандарт C, хотя там они работают несколько иначе. В языке C компилятор рассматривает константу как обычную переменную с пометкой «Не изменять!» При определении константы в C компилятор выделяет для нее память, поэтому при определении нескольких одноименных констант в двух разных файлах (или при размещении определения в заголовочном файле) компоновщик обнаруживает конфликты и выдает сообщения об ошибках. Предполагаемые варианты использования констант в C и C++ заметно различаются (если не вдаваться в подробности, в C++ константы спроектированы лучше).

Константные значения

В C++ константы всегда инициализируются некоторым значением (в C это не обязательно). Константы встроенных типов представляются в десятичной, восьмеричной, шестнадцатеричной или вещественной форме (к сожалению, проектировщики сочли, что двоичное представление недостаточно важно), а также в символьном виде.

При отсутствии других признаков компилятор считает, что константа задается в десятичном виде. Так, числа 47, 0 и 1101 считаются десятичными.

Если константа начинается с 0, она считается восьмеричным числом (система счисления с основанием 8). Восьмеричные числа могут состоять только из цифр 0–7; наличие других цифр приводит к ошибке компиляции. Пример допустимого восьмеричного числа — 017 (15 в десятичной записи).

Константа с префиксом 0x интерпретируется как шестнадцатеричное число (система счисления с основанием 16). Шестнадцатеричные числа состоят из цифр 0–9 и букв A–F — например, 0x1fe (510 в десятичной записи).

Вещественные числа могут содержать десятичную точку и экспоненту (символ «e», за которым следует число — «10 в указанной степени»). Наличие как десятичной точки, так и символа «e» не обязательно. Если присвоить такую константу вещественной переменной, компилятор значение константы преобразует в вещественное число; при этом происходит так называемое *неявное приведение типа*. Тем не менее присутствие десятичной точки или символа «e» поможет читателю понять, что в программе присутствует вещественное число. Такая подсказка также необходима для некоторых старых компиляторов.

Примеры допустимых значений вещественных чисел: 1e4, 1.0001, 47.0, 0.0 и –1.159e–77. Число также может содержать суффикс, определяющий тип вещественного числа: «f» или «F» для типа float, «L» или «l» для типа long double; без суффикса считается, что число относится к типу double.

Символьные константы представляют собой символы, заключенные в апострофы: 'A', '0'. Обратите внимание: между символом '0' (ASCII-код 96) и значением 0

существует огромная разница. Для представления специальных символов используется префикс в виде обратной косой черты: \n (символ новой строки), \t (символ табуляции), \\ (забой), \r (возврат каретки), \" (кавычка), \'(апостроф) и т. д. Кроме того, символьные константы могут представляться в виде кодов, восьмеричных (\17) или шестнадцатеричных (\xff).

Квалификатор volatile

Если квалификатор const сообщает компилятору о неизменности некоторого значения (что позволяет компилятору выполнить дополнительные оптимизации), то квалификатор volatile означает, что значение может измениться в любой момент, поэтому компилятор не должен применять оптимизации, руководствуясь предположениями об устойчивости этой переменной. Это ключевое слово обычно используется при получении данных, не контролируемых программой (например, из регистров или от коммуникационных устройств). При каждом обращении переменная с квалификатором volatile всегда читается заново, даже если в последний раз она была прочитана в предыдущей строке программы.

Особым случаем «памяти, не контролируемой программой», являются многопоточные программы[1]. Если программа отслеживает состояние флага, изменяемого другим программным потоком или процессом, этот флаг должен быть объявлен с ключевым словом volatile, чтобы компилятор не пытался оптимизировать многократные обращения к этому флагу.

Ключевое слово volatile не влияет на работу программы при отсутствии оптимизаций, но предотвращает критические ошибки в случае оптимизации кода, когда компилятор пытается исключить лишние операции чтения.

Мы еще вернемся к ключевым словам const и volatile в этой главе.

Операторы и их использование

В настоящем разделе рассматриваются все операторы C и C++.

Каждый оператор генерирует некоторый результат на основании полученных операндов. Результат обычно получается без модификации операндов (исключение составляют операторы присваивания, инкремента и декремента). Модификация операнда называется *побочным эффектом*. Обычно операторы, изменяющие свои операнды, вызываются именно для получения побочного эффекта, но следует помнить, что такие операторы тоже генерируют результат наравне с операторами, не имеющими побочных эффектов.

Присваивание

Присваивание выполняется оператором =. Выражение, находящееся в правой части (*r-значение*), копируется в область памяти, определяемую выражением в левой части (*l-значением*). R-значением может быть любая константа, переменная или выражение, дающая нужный результат, а l-значение должно определять физиче-

[1] В данном случае имеются в виду не рассматривавшиеся ранее потоки ввода-вывода (streams), а потоки выполнения (threads). — *Примеч. ред.*

скую область памяти для хранения данных. Скажем, переменной можно присвоить константу (A = 4), а присвоить что-либо константе невозможно — она не может быть l-значением (конструкция 4 = A недопустима).

Математические операторы

Основные математические операторы присутствуют в большинстве языков программирования: сложение (+), вычитание (-), деление (/), умножение (*), получение остатка от целочисленного деления (%). Целочисленное деление производится с усечением результата (без округления). Оператор получения остатка не может использоваться с вещественными числами.

В С и C++ также определена сокращенная запись для совмещения математических операций с присваиванием — знак операции, за которым следует знак =. Такая запись поддерживается всеми операторами языка, для которых она имеет смысл. Например, чтобы увеличить переменную x на 4 и присвоить результат x применяется команда x += 4.

Пример использования математических операторов:

```
//: C03:Mathops.cpp
// Математические операторы
#include <iostream>
using namespace std;

// Вспомогательный макрос для вывода строки и значения
#define PRINT(STR, VAR) \
cout << STR " = " << VAR << endl

int main() {
int i, j, k;
float u, v, w; // Также относится к double
cout << "enter an integer: ";
cin >> j;
cout << "enter another integer: ";
cin >> k;
PRINT("j",j); PRINT("k",k);
i = j + k; PRINT("j + k",i);
i = j - k; PRINT("j - k",i);
i = k / j; PRINT("k / j",i);
i = k * j; PRINT("k * j",i);
i = k % j; PRINT("k % j",i);
// Следующая команда работает только с целыми числами:
j %= k; PRINT("j %= k", j);
cout << "Enter a floating-point number: ";
cin >> v;
cout << "Enter another floating-point number:";
cin >> w;
PRINT("v",v); PRINT("w",w);
u = v + w; PRINT("v + w", u);
u = v - w; PRINT("v - w", u);
u = v * w; PRINT("v * w", u);
u = v / w; PRINT("v / w", u);
// Следующий фрагмент также работает с int, char и double:
PRINT("u", u); PRINT("v", v);
u += v; PRINT("u += v", u);
u -= v; PRINT("u -= v", u);
u *= v; PRINT("u *= v", u);
```

```
u /= v; PRINT("u /= v", u);
} ///:~
```

Конечно, r-значения при присваивании могут быть гораздо более сложными.

Знакомство с препроцессорными макросами

Обратите внимание на применение макроса PRINT() для сокращения объема программы (а также для предотвращения ошибок ввода!). Традиционно имена препроцессорных макросов задаются символами верхнего регистра, чтобы они сразу бросались в глаза, — как будет показано далее, макросы часто бывают опасными (хотя иногда они также бывают очень полезными).

Аргументы, указанные в круглых скобках при вызове, подставляются во всем коде макроса. При каждом вызове макроса препроцессор удаляет из программы имя PRINT и заменяет его кодом макроса, поэтому компилятор не может генерировать сообщения об ошибках с именем макроса и не проверяет типы аргументов (причем последнее иногда бывает полезным, как будет показано на примере отладочных макросов в конце главы).

Операторы отношения

Операторы отношения проверяют выполнение некоторого условия между значениями операндов. Если условие истинно, оператор возвращает логическую (ключевое слово bool в C++) величину true, а если условие ложно, возвращается false. Определены следующие операторы отношения: меньше (<), больше (>), меньше либо равно (<=), больше либо равно (>=), равно (==) и не равно (!=). Они работают со всеми встроенными типами C и C++. В C++ программист может определить собственную интерпретацию этих операторов для пользовательских типов данных (эта возможность будет рассматриваться в главе 12, посвященной перегрузке операторов).

Логические операторы

Логические операторы && (конъюнкция) и || (дизъюнкция) возвращают true или false в зависимости от логической связи их аргументов. Как отмечалось ранее, в C и C++ выражение, результат которого отличен от нуля, интерпретируется как true, а выражение с нулевым результатом интерпретируется как false. При выводе данных типа bool обычно выводится 1 для true и 0 для false.

Пример использования операторов отношения и логических операторов:

```
//: C03:Boolean.cpp
#include <iostream>
using namespace std;

int main() {
int i,j;
cout << "Enter an integer: ";
cin >> i;
cout << "Enter another integer: ";
cin >> j;
cout << "i > j is " << (i > j) << endl;
cout << "i < j is " << (i < j) << endl;
cout << "i >= j is " << (i >= j) << endl;
```

```
cout << "i <= j is " << (i <= j) << endl;
cout << "i == j is " << (i == j) << endl;
cout << "i != j is " << (i != j) << endl;
cout << "i && j is " << (i && j) << endl;
cout << "i || j is " << (i || j) << endl;
cout << " (i < 10) && (j < 10) is "
<< ((i < 10) && (j < 10)) << endl;
} ///:~
```

Тип int в этой программе можно заменить типом float или double. Однако следует помнить, что сравнение вещественного числа с нулем выполняется с максимальной точностью: даже если два числа различаются где-то в младших разрядах дробной части, они не равны. Вещественное число, на совсем ничтожную величину большее нуля, все равно интерпретируется как true.

Поразрядные операторы

Поразрядные операторы позволяют работать с числами на уровне отдельных битов. Поскольку в представлении вещественных чисел используется специальный внутренний формат, поразрядные операторы работают только с целочисленными типами (char, int и long). Для получения результата поразрядные операторы выполняют операции булевой алгебры с соответствующими битами аргументов.

Оператор *поразрядной конъюнкции* (&) включает в результат единичный бит только в том случае, если оба входных бита равны 1; в противном случае соответствующий бит результата равен 0. Оператор *поразрядной дизъюнкции* (|) включает в результат единичный бит, если хотя бы один из входных битов равен 1; нулевой бит включается только в том случае, если оба входных бита равны 0. Оператор поразрядной *исключающей конъюнкции* (^) включает в результат единичный бит, если один из входных битов (но не оба сразу) равен 1. Оператор *поразрядного отрицания* (~), также называемый оператором *дополнения до 1*, является унарным — он получает только один аргумент (остальные поразрядные операторы являются бинарными). Оператор поразрядного отрицания меняет значения входных битов на противоположные. Если входной бит был равен 0, соответствующий бит результата равен 1, и наоборот.

Поразрядные операторы могут объединяться со знаком = для совмещения операции с присваиванием. Операции &=, |=, ^= являются допустимыми. Унарный оператор ~ не может объединяться со знаком =.

Операторы сдвига

Операторы сдвига тоже работают на уровне отдельных битов. Оператор сдвига влево (<<) сдвигает левый операнд влево на количество разрядов, определяемое правым операндом. Оператор сдвига вправо (>>) сдвигает левый операнд вправо на количество разрядов, определяемое правым операндом. Если значение правого операнда превышает количество битов в левом операнде, результат не определен. Если левый операнд не имеет знака, то сдвиг вправо выполняется на логическом уровне, то есть старшие биты заполняются нулями. Если левый операнд является знаковым, то способ сдвига не определен.

Сдвиг может объединяться со знаком = (<<= и >>=). В этом случае левый операнд заменяется результатом сдвига его на количество разрядов, определяемое правым операндом.

Следующий пример демонстрирует использование всех поразрядных операторов. В начале листинга определяется функция общего назначения, которая выводит байт в двоичном представлении. Выделение кода в отдельную функцию упрощает его многократное использование. Объявление функции в заголовочном файле выглядит так:

```
//: C03:printBinary.h
// Вывод байта в двоичном представлении
void printBinary(const unsigned char val);
///:~
```

Реализация этой функции:

```
//: C03:printBinary.cpp {O}
#include <iostream>
void printBinary(const unsigned char val) {
for(int i = 7; i >= 0; i--)
if(val & (1 << i))
std::cout << "1";
else
std::cout << "0";
} ///:~
```

Функция printBinary() получает один байт и выводит его в поразрядном виде. Следующее выражение генерирует единичный бит со смещением позиции (в двоичном представлении — 00000001, 00000010 и т. д.):

```
(1 << i)
```

Если результат поразрядной конъюнкции этого бита с val отличен от нуля, значит, соответствующий разряд val содержит единичный бит.

Наконец, следующая функция демонстрирует использование поразрядных операторов:

```
//: C03:Bitwise.cpp
//{L} printBinary
// Демонстрация поразрядных операций
#include "printBinary.h"
#include <iostream>
using namespace std;

// Вспомогательный макрос:
#define PR(STR, EXPR) \
cout << STR; printBinary(EXPR); cout << endl;

int main() {
unsigned int getval;
unsigned char a, b;
cout << "Enter a number between 0 and 255: ";
cin >> getval; a = getval;
PR("a in binary: ", a);
cout << "Enter a number between 0 and 255: ";
cin >> getval; b = getval;
PR("b in binary: ", b);
PR("a | b = ", a | b);
PR("a & b = ", a & b);
PR("a ^ b = ", a ^ b);
PR("~a = ", ~a);
PR("~b = ", ~b);
// Интересная битовая маска:
unsigned char c = 0x5A;
```

```
PR("c in binary: ", c);
a |= c;
PR("a |= c; a = ", a);
b &= c;
PR("b &= c; b = ", b);
b ^= a;
PR("b ^= a; b = ", b);
} ///:~
```

В этом примере также используется препроцессорный макрос, уменьшающий объем программы. Макрос последовательно выводит заданную строку, двоичное представление некоторого выражения и символ новой строки.

В функции main() переменные объявлены беззнаковыми (unsigned). Это объясняется тем, что в общем случае при поразрядных операциях знак не нужен. Промежуточная переменная getval должна быть объявлена с типом int вместо char, поскольку в противном случае команда cin>> интерпретирует первую цифру как символ. Присваивание переменной getval переменным a и b преобразует прочитанное значение к одному байту (посредством усечения).

Операторы << и >> выполняют поразрядный сдвиг. Биты, выходящие за границу числа, теряются. Согласно программистскому фольклору, они попадают в мифическую *битовую корзину* — место, в котором хранятся все ненужные биты, ожидая своего использования когда-нибудь в будущем... При поразрядных операциях также нередко требуется *циклический сдвиг*, когда биты, «выталкиваемые» с одного конца числа, вставляются с другого конца. Хотя почти все современные процессоры поддерживают машинную команду циклического сдвига (и, соответственно, такая команда присутствует в ассемблере этих процессоров), прямая поддержка циклического сдвига в C и C++ отсутствует. Вероятно, проектировщики C (которые, по их словам, старались создать минимальный язык) решили проигнорировать эту команду, поскольку ее легко реализовать самостоятельно. Например, функции для выполнения циклического сдвига влево и вправо могут выглядеть так:

```
//: C03:Rotation.cpp {O}
// Циклический сдвиг влево и вправо

unsigned char rol(unsigned char val) {
int highbit;
if(val & 0x80) // 0x80 - маска, в которой установлен только старший бит
highbit = 1;
else
highbit = 0;
// Сдвиг влево (младший бит становится равным 0):
val <<= 1;
// Ввод старшего бита в младший бит:
val |= highbit;
return val;
}

unsigned char ror(unsigned char val) {
int lowbit;
if(val & 1) // Проверка младшего бита
lowbit = 1;
else
lowbit = 0;
val >>= 1; // Сдвиг вправо на одну позицию
// Ввод младшего бита в старший бит:
```

```
val |= (lowbit << 7);
return val;
} ///:~
```

Попробуйте использовать эти функции в программе Bitwise.cpp. Помните, что перед вызовом функций rol() и ror() их определения (или хотя бы объявления) должны быть обработаны компилятором.

Обычно поразрядные операции работают чрезвычайно эффективно, поскольку они напрямую транслируются в команды ассемблера. Иногда для одной команды С или С++ генерируется всего одна команда ассемблерного кода.

Унарные операторы

Кроме оператора поразрядного отрицания существуют и другие операторы, вызываемые с одним аргументом. Так, оператор *логического отрицания* (!) для полученного значения true генерирует false, и наоборот. Унарный минус (-) и плюс (+) внешне не отличаются от своих бинарных аналогов; компилятор выбирает нужный оператор в зависимости от контекста. Например, смысл следующей команды абсолютно очевиден:

```
x = -a;
```

С другой стороны, компилятор может разобраться и в такой команде:

```
x = a * -b;
```

Однако такой синтаксис плохо воспринимается при чтении программы, поэтому лучше использовать уточненную запись:

```
x = a * (-b);
```

Унарный минус изменяет знак числа на противоположный. Унарный плюс был определен для симметрии с унарным минусом, хотя на самом деле этот оператор ничего не делает.

Операторы инкремента и декремента (++ и --) уже упоминались в этой главе. Кроме оператора присваивания только эти два оператора обладают побочными эффектами. Операторы инкремента и декремента увеличивают и уменьшают переменную на единицу, причем «единица» в данном случае может иметь разные значения в зависимости от типа данных (это относится прежде всего к указателям).

В последнюю группу унарных операторов входят операторы получения адреса (&), разыменования (* и ->) и приведения типа. Эти операторы поддерживаются в С и С++, а операторы new и delete существуют только в С++. Операторы получения адреса и разыменования используются с указателями, о которых рассказывалось ранее. Приведение типов описано далее в этой главе, а операторы new и delete представлены в главе 4.

Тернарный оператор

Тернарный условный оператор выглядит несколько необычно, потому что он вызывается с тремя операндами. В отличие от традиционной условной конструкции if-else, тернарный оператор возвращает результат. Он состоит из трех выражений: если первое выражение, за которым следует знак ?, истинно, то оператор вычисляет выражение после знака ?, а полученный результат возвращается тернарным операто-

ром. Если же первое выражение ложно, то вычисляется выражение после знака двоеточия (:), а полученный результат определяет значение, возвращаемое оператором.

Тернарный оператор может использоваться как для создания побочных эффектов, так и для получения своего значения. В следующем фрагменте демонстрируются обе возможности:

```
a = --b ? b : (b = -99);
```

В данном случае переменной a присваивается значение b, если результат уменьшения b отличен от нуля. Если же этот результат равен нулю, то и a, и b присваивается –99. Переменная b уменьшается всегда, но число –99 присваивается ей только в том случае, если в результате уменьшения значение b становится равным 0. Аналогичная команда может использоваться и без присваивания a=, просто ради ее побочных эффектов:

```
--b ? b : (b = -99);
```

Здесь второе вхождение b является лишним, поскольку сгенерированное значение не используется. Однако правила требуют, чтобы между ? и : находилось выражение. В этом случае вместо выражения можно подставить обычную константу, что немного ускорит работу программы.

Оператор запятой

Роль запятой не ограничивается разделением имен переменных в совмещенных определениях вида

```
int i, j, k;
```

Конечно, запятые также требуются в списках аргументов функций. Еще запятая может применяться в качестве оператора для разделения выражений — в этом случае результатом оператора является значение последнего из перечисленных выражений. Все остальные выражения в списке, разделенном запятыми, вычисляются только ради побочных эффектов. В следующем примере эта конструкция увеличивает значения нескольких переменных, последнее из которых используется в качестве r-значения при присваивании:

```
//: C03:CommaOperator.cpp
#include <iostream>
using namespace std;
int main() {
int a = 0, b = 1, c = 2, d = 3, e = 4;
a = (b++, c++, d++, e++);
cout << "a = " << a << endl;
// Круглые скобки важны. Без них выражение интерпретируется как:
(a = b++), c++, d++, e++;
cout << "a = " << a << endl;
} ///:~
```

Обычно не стоит задействовать запятые для чего-либо, кроме разделения, поскольку многие программисты не привыкли к тому, что запятые могут быть операторами.

Характерные ошибки при использовании операторов

Как было показано ранее, одна из распространенных ошибок при использовании операторов заключается в том, что программист забывает о круглых скобках, когда

существуют хотя бы малейшие сомнения в порядке обработки выражения (информацию о порядке обработки операторов в выражениях можно найти в руководстве по языку C).

Другая чрезвычайно распространенная ошибка выглядит так:

```
//: C03:Pitfall.cpp
// Ошибки при использовании операторов

int main() {
int a = 1, b = 1;
while(a = b) {
// ...
}
} ///:~
```

Выражение a = b всегда будет истинным, пока значение переменной b отлично от нуля. Переменной a присваивается значение b, и это же значение возвращается как результат оператора =. Как правило, в условных конструкциях следует использовать оператор проверки равенства == вместо оператора присваивания =. С этой ошибкой сталкиваются очень многие программисты (впрочем, компилятор обычно предупреждает о подозрительном присваивании, что весьма полезно).

Аналогичная ошибка возникает при использовании поразрядных операторов вместо их логических аналогов. Поразрядные операторы конъюнкции и дизъюнкции состоят из одного символа (& или |), а логические операторы состоят из двух символов (&& и ||). Как и в случае с операторами = и ==, программисты часто машинально вводят один символ вместо двух. Для таких ситуаций существует мнемоническое правило: «Биты маленькие, поэтому для их операторов нужно меньше символов».

Операторы приведения типов

Компилятор автоматически приводит типы данных там, где это оправдано контекстом применения. Например, если целое значение присваивается вещественной переменной, компилятор незаметно вызовет функцию (или более вероятно, вставит фрагмент кода) для приведения int к типу float. Приведение типов позволяет выполнять подобные переходы явно или обеспечить их принудительное выполнение там, где обычно их быть не должно.

Чтобы выполнить приведение типа, следует указать нужный тип данных (вместе со всеми модификаторами) в круглых скобках слева от значения — переменной, константы, результата выражения или возвращаемого значения функции. Пример:

```
//: C03:SimpleCast.cpp
int main() {
int b = 200;
unsigned long a = (unsigned long int)b;
} ///:~
```

Конечно, в данном примере можно обойтись без приведения типа; достаточно использовать константу 200f (кстати, обычно компилятор именно так оптимизирует это выражение). Приведение чаще используются с переменными, а не с константами.

Явное приведение типов в С++

При приведении типов необходима осторожность, потому что вы фактически говорите компилятору: «Забудь о проверке типов — это значение нужно интерпретировать как относящееся к другому типу». Другими словами, вы намеренно открываете брешь в системе безопасности типов С++ и запрещаете компилятору сообщать о неправильных операциях с типом. Что еще хуже, компилятор безоговорочно верит вам и не пытается обнаруживать возможные ошибки. Приведение типов часто становится источником всевозможных проблем. Более того, к любой программе с многочисленными операциями приведения типов следует относиться с подозрением, сколько бы вам ни твердили, что задача «неизбежно» решается именно таким способом. В общем случае количество таких операций должно быть минимальным, причем они должны ограничиваться решением узкоспециализированных задач.

Допустим, вам принесли неработоспособную программу, и вы решаете начать поиск ошибок с операций приведения типов. Но как обнаружить приведения типов в стиле С? Они представляют собой обычные имена типов в круглых скобках, а такие конструкции на первый взгляд трудно отличить от обычных элементов программы.

Стандарт С++ описывает синтаксис операций явного приведения типов, способных полностью заменить операции приведения в стиле С (конечно, запрет на прежний синтаксис нарушил бы работу существующих программ, но компиляторы могут выводить предупреждения об операциях приведения в стиле С). Новый синтаксис приведения типа сразу бросается в глаза и отличается от других элементов программы.

Оператор	Описание
static_cast	Для «безопасного» и «более или менее безопасного» приведения, включая то, которое может быть выполнено неявно (например, автоматическое приведение типа)
const_cast	Изменение статуса объявлений const и/или volatile
reinterpret_cast	Приведение к типу с совершенно иной интерпретацией. Принципиальная особенность этого приведения заключается в том, что для надежного использования значение должно быть приведено обратно к исходному типу. Тип, к которому выполняется приведение, обычно задействуется для манипуляций с битами или других неочевидных целей. Это самый опасный из всех видов приведения
dynamic_cast	Понижающее приведение, безопасное по отношению к типам (глава 15)

Первые три разновидности оператора явного приведения типа более подробно описываются в следующих разделах, а знакомство с последним оператором требует более глубокого знания материала, поэтому откладывается до главы 15.

Оператор static_cast

Приведение static_cast используется для всех четко определенных операций приведения типа. К этой категории относятся операции «безопасного» приведения типа, которые могут автоматически выполняться компилятором, и менее безопасные операции, все же считающиеся четко определенными. К числу операций, обычно выполняемых при помощи оператора static_cast, относятся типичное автомати-

ческое приведение, приведение с потерей информации, принудительное приведение к типу **void***, неявное приведение и статические перемещения по иерархиям классов (поскольку классы и наследование еще толком не рассматривались, последняя тема откладывается до главы 15).

```
//: C03:static_cast.cpp
void func(int) {}

int main() {
int i = 0x7fff; // Максимальное положительное значение = 32767
long l;
float f;
// (1) Типичные автоматические преобразования:
l = i;
f = i;
// Также могут выполняться командами:
l = static_cast<long>(i);
f = static_cast<float>(i);

// (2) Преобразования с потерей информации:
i = l; // Возможна потеря данных
i = f; // Возможна потеря данных
// Означает "Я знаю, что делаю", подавляет предупреждения:
i = static_cast<int>(l);
i = static_cast<int>(f);
char c = static_cast<char>(i);

// (3) Принудительное преобразование void* :
void* vp = &i;
// Опасный старый способ:
float* fp = (float*)vp;
// Новый способ не менее опасен:
fp = static_cast<float*>(vp);

// (4) Неявное приведение типа, обычно выполняемое компилятором:
double d = 0.0;
int x = d; // Автоматическое приведение типа
x = static_cast<int>(d); // Явное приведение типа
func(d); // Автоматическое приведение типа
func(static_cast<int>(d)); // Явное приведение типа
} ///:~
```

В первой секции продемонстрированы разновидности операций приведения типа, знакомые по языку C, — как явные, так и неявные. Повышение от int к long или float не вызывает проблем, поскольку новый тип всегда позволяет представить любые величины, представляемые старым типом. Хотя это не обязательно, повышающее приведение типа можно выполнить при помощи конструкции static_cast.

Во второй секции продемонстрированы обратные преобразования. Они могут привести к потере данных, поскольку тип int более «узок» по сравнению с long или float; он не позволяет хранить все числа исходного размера. По этой причине такие преобразования называются операциями *сужающего приведения типа*. Компилятор выполняет их, но часто выдает предупреждения. Программист может подавить предупреждение и указать, что приведение типа выполняется вполне осознанно.

Как показывает третья секция, присваивание через **void*** в C++ возможно лишь при приведении типа (в C дело обстоит иначе). Такие преобразования особенно

опасны, программист должен хорошо понимать, что он делает. По крайней мере, в процессе отладки вам будет проще найти оператор static_cast, чем команду приведения в старом варианте синтаксиса.

В четвертой секции представлены операции неявного приведения типов, которые обычно автоматически выполняются компилятором. Явное приведение в этих случаях не обязательно, но, как и прежде, оператор static_cast выделяет операцию на случай, если вы захотите привлечь к ней внимание читателя или упростить ее поиск.

Оператор const_cast

Оператор const_cast позволяет лишить переменную статуса const или volatile. Это *единственная* разновидность преобразований, допустимых для const_cast. Любые другие операции приведения типов должны выполняться в отдельном выражении, в противном случае произойдет ошибка компиляции.

```
//: C03:const_cast.cpp
int main() {
const int i = 0;
int* j = (int*)&i; // Устаревшая форма
j = const_cast<int*>(&i); // Рекомендуемая форма
// Одновременное выполнение других преобразований не допускается:
//! long* l = const_cast<long*>(&i); // Ошибка
volatile int k = 0;
int* u = const_cast<int*>(&k);
} ///:~
```

При получении адреса константного объекта результат представляет собой указатель на const, который без приведения типа нельзя присвоить указателю на неконстантный объект. Задача также решается приведением типа в прежнем стиле, но лучше использовать конструкцию const_cast. То же самое относится и к volatile.

Оператор reinterpret_cast

Оператор reinterpret_cast обеспечивает самый ненадежный из всех механизмов приведения типов, чаще всего приводящий к ошибкам. При преобразовании reinterpret_cast предполагается, что объект представляет собой простую последовательность битов, которая может интерпретироваться (для каких-то темных целей) как объект совершенно другого типа. Это типичный пример низкоуровневых манипуляций с битами, которыми так печально известен язык C. Прежде чем что-либо делать с результатом оператора reinterpret_cast, его практически всегда приходится приводить обратно к исходному типу (или иным образом интерпретировать переменную как относящуюся к исходному типу).

```
//: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; };

void print(X* x) {
for(int i = 0; i < sz; i++)
cout << x->a[i] << ' ';
cout << endl << "-------------------" << endl;
}
```

```
int main() {
X x;
print(&x);
int* xp = reinterpret_cast<int*>(&x);
for(int* i = xp; i < xp + sz; i++)
*i = 0;
// Указатель xp можно будет использовать как X*
// только после обратного преобразования.
print(reinterpret_cast<X*>(xp));
// В данном примере также можно просто использовать
// исходный идентификатор:
print(&x);
} ///:~
```

В этом простом примере struct X содержит массив int, но после создания структуры в стеке X x значения элементов массива содержат «мусор» (что доказывается выводом содержимого struct функцией print()). Чтобы инициализировать элементы, мы получаем адрес X и преобразовываем его в указатель на int, а затем последовательно перебираем элементы массива и обнуляем каждый элемент типа int. Обратите внимание на вычисление верхней границы i «прибавлением» sz к xp; компилятор знает, что в действительности речь идет о sz элементах начиная с адреса xp и выполняет правильные математические операции с указателями.

Главным в работе оператора reinterpret_cast является то, что результат приведения типов оказывается настолько чуждым, что он не может использоваться в контексте исходного типа до выполнения обратного преобразования. В нашем примере в команде вывода производится обратное приведение к типу X*, хотя, конечно, при наличии исходного идентификатора можно просто задействовать его. Но xp может использоваться только как int*, то есть речь действительно идет о новой интерпретации исходной структуры X.

Наличие оператора reinterpret_cast часто является признаком плохого стиля программирования и/или ухудшает переносимость программ, но если вы решите, что без него не обойтись, — он в вашем распоряжении.

Оператор sizeof

Оператор sizeof стоит отдельно от других, поскольку решает весьма необычную задачу — получение информации об объеме памяти, выделенном для хранения данных. Как упоминалось ранее в этой главе, sizeof возвращает количество байтов, используемых для хранения конкретной переменной. Кроме того, он также может возвращать размер типа данных (если имя переменной не указано):

```
//: C03:sizeof.cpp
#include <iostream>
using namespace std;
int main() {
cout << "sizeof(double) = " << sizeof(double);
cout << ", sizeof(char) = " << sizeof(char);
} ///:~
```

По определению для любой разновидности типа char (signed, unsigned или без уточнения) sizeof всегда возвращает 1 независимо от того, действительно ли char занимает один байт памяти. Для остальных типов результат равен размеру в байтах.

Обратите внимание: sizeof является оператором, а не функцией. Применительно к типам должна использоваться форма с круглыми скобками, приведенная выше, а для переменных допускается вызов без круглых скобок:

```
//: C03:sizeofOperator.cpp
int main() {
int x;
int i = sizeof x;
} ///:~
```

Оператор sizeof также позволяет получать информацию о размерах пользовательских типов данных. Эта возможность будет рассмотрена далее.

Ключевое слово asm

Предусмотрен механизм включения в программы С++ ассемблерного кода для работы с устройствами. Часто в ассемблерном коде можно ссылаться на переменные С++, что позволяет легко взаимодействовать с кодом С++ и ограничиться ассемблером для целей оптимизации или использования специальных команд процессора. Синтаксис ассемблерных вставок зависит от компилятора, за дополнительной информацией обращайтесь к документации по компилятору.

Синонимы операторов

Для поразрядных и логических операторов в С++ были определены специальные ключевые слова. Некоторые программисты за пределами США, на клавиатурах которых не было символов &, |, ^ и т. д., были вынуждены использовать кошмарные *триграфы* языка С, которые не только затрудняли ввод, но и плохо воспринимались при чтении. Ситуация была исправлена в С++ введением новых ключевых слов.

Ключевое слово	Описание
and	&& (логическая конъюнкция)
or	|| (логическая дизъюнкция)
not	! (логическое отрицание)
not_eq	!= (логическое неравенство)
bitand	& (поразрядная конъюнкция)
and_eq	&= (поразрядная конъюнкция с присваиванием)
bitor	| (поразрядная дизъюнкция)
or_eq	|= (поразрядная дизъюнкция с присваиванием)
xor	^ (поразрядная исключающая дизъюнкция)
xor_eq	^= (поразрядная исключающая дизъюнкция с присваиванием)
compl	~ (дополнение до 1)

Эти ключевые слова поддерживаются всеми компиляторами, соответствующими стандарту С++.

Создание составных типов

Встроенные типы данных и их разновидности чрезвычайно важны, но весьма примитивны. В С и С++ предусмотрены средства для создания более сложных типов

данных на основе встроенных типов. Важнейшим из этих средств является ключевое слово struct, аналог ключевого слова class в C++. Тем не менее простейший способ создания более сложных типов заключается в определении псевдонимов при помощи ключевого слова typedef.

Определение псевдонимов

Ключевое слово typedef выглядит обманчиво: оно наводит на мысли об «определении типов» (TYPe DEFinition), хотя, вероятно, правильнее было бы использовать термин «псевдоним», поскольку переименование типа — это именно то, что оно делает в действительности. Синтаксис:

```
typedef описание_существующего_типа псевдоним
```

Программисты часто используют слово typedef для усложненных типов данных просто для того, чтобы избавиться от ввода лишних символов. Пример распространенного псевдонима:

```
typedef unsigned long ulong
```

Встретив в программе слово ulong, компилятор знает, что имелось в виду unsigned long. Возникает мысль, что того же результата можно легко добиться при помощи препроцессорных подстановок, но в некоторых ключевых ситуациях компилятор должен знать, что имя соответствует именно типу, поэтому ключевое typedef действительно необходимо.

В частности, typedef может пригодиться для определения типов указателей. Как упоминалось ранее, следующая запись определяет переменную x типа int* и переменную y типа int (а не int*!):

```
int* x, y;
```

Таким образом, * ассоциируется с именем, а не с типом. Проблема решается определением псевдонима:

```
typedef int* IntPtr;
IntPtr x, y;
```

На этот раз обе переменные x и y относятся к типу int*.

Считается, что определение псевдонимов для примитивных типов нежелательно, поскольку программа становится менее наглядной. В самом деле, большое количество слов typedef затрудняет чтение программы. Однако в языке C это слово играет особенно важную роль в сочетании с ключевым словом struct.

Объединение переменных в структуры

Ключевое слово struct предназначено для группировки переменных в структуры. После создания структуры можно получить несколько экземпляров «нового» типа данных. Пример:

```
//: C03:SimpleStruct.cpp
struct Structure1 {
char c;
int i;
float f;
double d;
};
```

```
int main() {
struct Structure1 s1, s2;
s1.c = 'a'; // Выбор элемента при помощи символа '.'
s1.i = 1;
s1.f = 3.14;
s1.d = 0.00093;
s2.c = 'a';
s2.i = 1;
s2.f = 3.14;
s2.d = 0.00093;
} ///:~
```

Объявление struct должно заканчиваться символом точки с запятой. В функции main() создаются два экземпляра структуры Structure1: s1 и s2. Каждый экземпляр содержит собственные версии переменных c, i, f и d. Таким образом, s1 и s2 представляют группы полностью независимых переменных. Обращение к элементам s1 и s2 производится с использованием уточнения в виде точки (.), встречавшегося в предыдущих главах при описании классов C++. Поскольку классы произошли от структур, становится понятно, откуда взялся этот синтаксис.

Обратите внимание на неудобство использования структуры Structure1 (впрочем, это неудобство характерно только для C, но не для C++). В языке C при определении переменной нельзя ограничиться указанием типа Structure1, обязательно приходится указывать тип struct Structure1. В таких ситуациях ключевое слово typedef особенно удобно:

```
//: C03:SimpleStruct2.cpp
// Использование ключевого слова typedef со структурами
typedef struct {
char c;
int i;
float f;
double d;
} Structure2;

int main() {
Structure2 s1, s2;
s1.c = 'a';
s1.i = 1;
s1.f = 3.14;
s1.d = 0.00093;
s2.c = 'a';
s2.i = 1;
s2.f = 3.14;
s2.d = 0.00093;
} ///:~
```

В случае такого использования ключевого слова typedef при определении переменных s1 и s2 имя Structure2 может быть именем встроенного типа, такого как int или float (это относится к языку C, — попробуйте убрать typedef в C++). Структуры C обладают только данными, то есть характеристиками; они не обладают собственным поведением, присущим настоящим объектам C++. Обратите внимание на отсутствие идентификатора структуры в начале — при определении typedef начальный идентификатор не обязателен. Однако иногда ссылки на структуру должны использоваться в ее определении, и в таких случаях имя структуры указывается как для struct, так и для typedef:

```
//: C03:SelfReferential.cpp
// Структура может ссылаться сама на себя

typedef struct SelfReferential {
int i;
SelfReferential* sr; // Голова еще не кружится?
} SelfReferential;

int main() {
SelfReferential sr1, sr2;
sr1.sr = &sr2;
sr2.sr = &sr1;
sr1.i = 47;
sr2.i = 1024;
} ///:~
```

Если присмотреться к этому листингу, становится ясно, что sr1 и sr2 ссылаются друг на друга, при этом каждая из структур также содержит данные.

Вообще говоря, имя struct не обязано совпадать с именем typedef, но обычно используются одинаковые имена, так как это упрощает программу.

Указатели и структуры

В приведенных примерах работа со структурами осуществлялась напрямую через переменную. Однако вы можете получить адрес структуры, как и адрес любого объекта в памяти (как показано в программе SelfReferential.cpp). Если для обращения к элементам структуры через обычную переменную требуется оператор точка (.), то при обращении к элементу через указатель на структуру используется оператор ->. Пример:

```
//: C03:SimpleStruct3.cpp
// Использование указателей на структуры
typedef struct Structure3 {
char c;
int i;
float f;
double d;
} Structure3;

int main() {
Structure3 s1, s2;
Structure3* sp = &s1;
sp->c = 'a';
sp->i = 1;
sp->f = 3.14;
sp->d = 0.00093;
sp = &s2; // Указатель на другой экземпляр структуры
sp->c = 'a';
sp->i = 1;
sp->f = 3.14;
sp->d = 0.00093;
} ///:~
```

В функции main() указатель sp сначала указывает на s1. Он требуется для инициализации полей структуры s1, выбираемых оператором -> (этот же оператор используется для чтения полей). Затем указатель sp переводится на структуру s2, после чего переменные этой структуры инициализируются аналогичным образом.

В этом проявляется еще одно преимущество указателей: они могут динамически перенаправляться на разные объекты, благодаря чему программирование становится более гибким.

На данный момент это все, что вам необходимо знать о структурах, но по мере изложения материала вы познакомитесь с ними (а также с их наследниками — классами) более подробно.

Перечисляемые типы

Перечисляемый тип данных связывает числа с именами, чтобы сделать их более понятными для человека, читающего программу. Ключевое слово enum языка C автоматически нумерует список переданных идентификаторов и присваивает им значения 0, 1, 2 и т. д. Допускается объявление переменных типа enum (такие переменные всегда представляются в виде целых значений). Объявление перечисляемого типа внешне напоминает объявление структуры.

Перечисляемый тип данных удобен при работе с фиксированными наборами из нескольких вариантов:

```
//: C03:Enum.cpp
// Отслеживание типа фигуры

enum ShapeType {
circle.
square.
rectangle
}; // Определение перечисляемого типа. как и определение структуры.
// должно завершаться точкой с запятой.

int main() {
ShapeType shape = circle;
// Некоторые действия...
// Дальнейшее зависит от разновидности фигуры:
switch(shape) {
case circle: /* для кругов */ break;
case square: /* для квадратов */ break;
case rectangle: /* для прямоугольников */ break;
}
} ///:~
```

Здесь shape — переменная перечисляемого типа ShapeType, значение которой сравнивается со значением перечисляемого типа. Но поскольку переменная shape в действительности содержит обычное число типа int, это может быть любое значение, допустимое для типа int (в том числе и отрицательное). Переменные int также можно сравнивать с переменными перечисляемого типа.

Учтите, что конструкция switch с выбором по перечисляемому типу, как в этом примере, являет собой сомнительный стиль программирования. В C++ предусмотрены гораздо более совершенные средства программирования таких решений, к которым мы вернемся гораздо позже.

Если вам не нравится распределение значений компилятором, вы можете выполнить его самостоятельно:

```
enum ShapeType {
circle = 10. square = 20. rectangle = 50
};
```

Если для одних идентификаторов значение указывается, а для других — нет, компилятор автоматически назначает следующее целое число. Пример:

```
enum snap { crackle = 25, pop };
```

Компилятор присваивает переменной pop значение 26.

Как нетрудно убедиться, перечисляемые типы делают программу более наглядной. Тем не менее они в определенной степени остаются С-аналогом того, что делается в C++ при помощи ключевого слова class, поэтому ключевое слово enum используется в C++ гораздо реже.

Проверка типов для перечислений

Перечисления С весьма примитивны. Они просто связывают целые значения с именами, но не поддерживают проверки типов. Как вы, вероятно, уже поняли, в C++ концепция типов играет более важную роль, и это в полной мере относится к перечислениям. Создавая именованное перечисление, вы фактически определяете новый тип так же, как это делается с классом: имя перечисления становится зарезервированным словом в контексте данной единицы трансляции.

Кроме того, в C++ для перечислений выполняется гораздо более жесткая проверка типов, чем в С. Если в программе определена переменная a перечисляемого типа color, то в программе С можно использовать выражение a++, но в программе C++ оно запрещено. Дело в том, что увеличение переменной перечисляемого типа сопряжено с двумя операциями приведения типа; одна из них разрешена в C++, а другая запрещена. Сначала значение перечисляемого типа неявно преобразуется из color в int, происходит увеличение, после чего результат снова преобразуется в color. В C++ последняя фаза не разрешена — color является отдельным типом, не эквивалентным int. И это вполне логично; откуда нам знать, что результат увеличения blue вообще будет присутствовать в списке цветов? Если вы хотите увеличивать значения перечисляемого типа color, оформите его в виде класса со своим оператором ++, потому что для классов можно обеспечить гораздо более надежную защиту. Встречая в программе код, подразумевающий неявное приведение к типу enum, компилятор предупреждает о потенциально опасной операции.

Для объединений, описанных далее, в C++ также реализована дополнительная проверка типов.

Экономия памяти при использовании объединений

Иногда программе приходится работать с разными типами данных через одну переменную. В подобных ситуациях существуют два варианта: создать структуру с полями для всех возможных типов или воспользоваться *объединением*. Ключевое слово union собирает разные типы данных в одну область памяти; компилятор вычисляет максимальный объем памяти для хранения наибольшего объекта, входящего в объединение, и использует полученный результат как размер объединения. Таким образом, объединения экономят память.

Любое значение, сохраняемое в объединении, всегда начинается с начального адреса объединения, но для его хранения реально используется ровно столько памяти, сколько необходимо. В сущности, получается некая «сверхпеременная», способная вместить любую из переменных объединения. Все адреса переменных, входящих в объединение, совпадают (в классах и структурах адреса переменных различаются).

Рассмотрим простой пример использования объединения. Попробуйте исключить те или иные элементы и посмотрите, как это отразится на размере объединения. Помните, что включать в объединение несколько однотипных переменных бессмысленно (разве что если вам требуются разные имена).

```
//: C03:Union.cpp
// Размер и простейшее использование объединения
#include <iostream>
using namespace std;

union Packed { // Объявление выглядит почти так же, как для структуры
char i;
short j;
int k;
long l;
float f;
double d;
// Размер объединения равен размеру double,
// поскольку элемент этого типа является наибольшим.
}; // Объединение, как и структура, завершается символом ";"

int main() {
cout << "sizeof(Packed) = "
<< sizeof(Packed) << endl;
Packed x;
x.i = 'c';
cout << x.i << endl;
x.d = 3.14159;
cout << x.d << endl;
} ///:~
```

Компилятор выполняет правильное присваивание в зависимости от того, какой из элементов объединения указан при присваивании.

Компилятор не следит за тем, что делается с объединением после присваивания. Скажем, в предыдущем примере можно присвоить x вещественное значение:

```
x.f = 2.222;
```

Теперь можно вывести это значение в выходной поток как значение типа int:

```
cout << x.i;
```

В результате будет выведен «мусор».

Массивы

Массивы тоже относятся к категории составных типов, поскольку они позволяют сгруппировать несколько переменных, расположенных последовательно друг за другом, под одним идентификатором. Например, следующая запись выделяет память для 10 последовательных переменных int, но без присваивания уникальных идентификаторов:

```
int a[10];
```

Вместо этого все переменные группируются под общим именем a.

При обращении к *элементу массива* используется такая же форма записи с квадратными скобками, как и при определении массива:

```
a[5] = 47;
```

Хотя *размер* массива а равен 10, индексирование (нумерация элементов) начинается с нуля, поэтому допустимые индексы элементов находятся в интервале 0–9:

```
//: C03:Arrays.cpp
#include <iostream>
using namespace std;

int main() {
int a[10];
for(int i = 0; i < 10; i++) {
a[i] = i * 10;
cout << "a[" << i << "] = " << a[i] << endl;
}
} ///:~
```

Обращения к массивам выполняются чрезвычайно быстро. Тем не менее страховка на случай нарушения границ массива не предусмотрена — программа начнет портить содержимое других переменных. Другой недостаток заключается в том, что размер массива должен определяться на стадии компиляции; если вдруг потребуется изменить размер массива во время выполнения программы, то сделать это в приведенном выше синтаксисе не удастся (вообще говоря, в С предусмотрен способ создания динамических массивов, но он громоздок и неудобен). Класс C++ vector, представленный в предыдущей главе, реализует объектный аналог массива с автоматическим изменением размеров. Если размер массива не известен на стадии компиляции, это решение обычно гораздо удобнее.

Элементы массивов могут относиться к произвольному типу, даже к структурному:

```
//: C03:StructArray.cpp
// Массив структур

typedef struct {
int i, j, k;
} ThreeDpoint;

int main() {
ThreeDpoint p[10];
for(int i = 0; i < 10; i++) {
p[i].i = i + 1;
p[i].j = i + 2;
p[i].k = i + 3;
}
} ///:~
```

Обратите внимание: идентификатор поля структуры i никак не связан с одноименным счетчиком цикла for.

Чтобы убедиться в том, что элементы массива действительно хранятся в смежных областях памяти, можно вывести их адреса:

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std;

int main() {
int a[10];
cout << "sizeof(int) = "<< sizeof(int) << endl;
for(int i = 0; i < 10; i++)
cout << "&a[" << i << "] = "
```

```
<< (long)&a[i] << endl;
} ///:~
```

Запустив эту программу, вы увидите, что каждый элемент действительно находится на расстоянии размера соответствующего типа от предыдущего элемента. Следовательно, элементы располагаются в памяти вплотную друг к другу.

Указатели и массивы

Идентификаторы массивов отличаются от идентификаторов обычных переменных. Во-первых, идентификатор массива не является l-значением; ему нельзя присвоить значение. Он всего лишь обеспечивает доступ к синтаксису индексирования, а само имя массива без квадратных скобок определяет начальный адрес массива:

```
//: C03:ArrayIdentifier.cpp
#include <iostream>
using namespace std;

int main() {
int a[10];
cout << "a = " << a << endl;
cout << "&a[0] =" << &a[0] << endl;
} ///:~
```

При запуске этой программы два адреса совпадут (адреса выводятся в шестнадцатеричном формате, поскольку приведение к типу long отсутствует).

Следовательно, идентификатор массива можно трактовать как указатель на начало массива, доступный только для чтения. Хотя идентификатор массива нельзя перевести на другой адрес памяти, вы *можете* создать другой указатель и использовать его для перебора элементов массива. Более того, синтаксис индексирования работает с обычными указателями:

```
//: C03:PointersAndBrackets.cpp
int main() {
int a[10];
int* ip = a;
for(int i = 0; i < 10; i++)
ip[i] = i * 10;
} ///:~
```

Тот факт, что имя массива определяет его начальный адрес, оказывается весьма важным при передаче массива в функцию. При объявлении массива как аргумента функции в действительности объявляется указатель. Это означает, что в следующем примере функции func1() и func2() имеют одинаковые списки аргументов:

```
//: C03:ArrayArguments.cpp
#include <iostream>
#include <string>
using namespace std;

void func1(int a[], int size) {
for(int i = 0; i < size; i++)
a[i] = i * i - i;
}

void func2(int* a, int size) {
for(int i = 0; i < size; i++)
a[i] = i * i + i;
```

```
}

void print(int a[], string name, int size) {
for(int i = 0; i < size; i++)
cout << name << "[" << i << "] = "
<< a[i] << endl;
}

int main() {
int a[5], b[5];
// Вероятно, будет выведен "мусор":
print(a, "a", 5);
print(b, "b", 5);
// Инициализация массивов:
func1(a, 5);
func1(b, 5);
print(a, "a", 5);
print(b, "b", 5);
// Обратите внимание: массивы всегда модифицируются:
func2(a, 5);
func2(b, 5);
print(a, "a", 5);
print(b, "b", 5);
} ///:~
```

Хотя аргументы функций func1() и func2() объявляются по-разному, внутри функций они используются одинаково. Этот пример демонстрирует еще одно важное обстоятельство: массивы не могут передаваться по значению[1], то есть передача массива функции не приводит к автоматическому созданию его локальной копии. Следовательно, при модификации массива всегда изменяется внешний объект. Иногда это вызывает недоразумения, особенно если программист ожидает стандартной передачи по значению, характерной для обычных аргументов.

Обратите внимание: в функции print() для аргументов-массивов используется синтаксис с квадратными скобками. Хотя синтаксис указателей и синтаксис индексирования фактически эквивалентны при передаче аргументов-массивов, синтаксис индексирования более четко сообщает читателю программы, что аргумент рассматривается как массив.

Также обратите внимание на то, что в обоих случаях передается аргумент с размером массива. Одного адреса массива недостаточно; функция всегда должна знать, сколько памяти занимает массив, чтобы случайно не выйти за его пределы.

Массивы могут содержать элементы произвольного типа, в том числе и указатели. Оказывается, для передачи программе аргументов командной строки в С и С++ предусмотрен специальный список аргументов функции main(), который выглядит так:

```
int main(int argc, char* argv[]) { // ...
```

Первый аргумент определяет количество элементов в массиве, передаваемом во втором аргументе. Вторым аргументом всегда является массив char*, потому

[1] Разве что если рассматривать вопрос с сугубо формальных позиций: «Все аргументы в C/C++ передаются по значению, а „значением“ массива является то, что соответствует его идентификатору, то есть адрес». С точки зрения ассемблера это похоже на правду, но вряд ли подобные рассуждения годятся для высокоуровневых концепций. Введение ссылок в С++ еще больше запутывает утверждения о «передаче только по значению» вплоть до того, что происходящее становится удобнее рассматривать в терминах «передачи по значению» и «передачи адресов».

что аргументы командной строки передаются в виде символьных массивов (а как отмечалось ранее, массив может передаваться только в виде указателя). Каждая группа символов командной строки, окруженная пробелами, преобразуется в отдельный аргумент-массив. Следующая программа выводит все значения аргументов командной строки посредством перебора в массиве:

```
//: C03:CommandLineArgs.cpp
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
cout << "argc = " << argc << endl;
for(int i = 0; i < argc; i++)
cout << "argv[" << i << "] = "
<< argv[i] << endl;
} ///:~
```

Элемент argv[0] содержит путь и имя запущенной программы, что позволяет программе получить информацию о себе. Однако при этом в массиве аргументов появляется дополнительный элемент, поэтому при получении аргументов командной строки часто встречается стандартная ошибка, когда программист запрашивает argv[0] вместо argv[1].

Вообще говоря, вы не обязаны использовать идентификаторы argc и argv в функции main(); это всего лишь условное обозначение (но выбор других идентификаторов может вызвать недоразумения). Также существует альтернативный способ объявления argv:

```
int main(int argc, char** argv) { // ...
```

Эти две формы эквивалентны, но, как считает автор, версия, использованная в книге, более наглядна — она прямо говорит: «перед вами массив указателей на символы».

Из командной строки можно получить только символьные массивы. Если вдруг потребуется интерпретировать аргумент в контексте другого типа, вам придется самостоятельно выполнить в программе его приведение. Для упрощения приведения к числовой форме в стандартную библиотеку С были включены вспомогательные функции, объявленные в <cstdlib>. Простейшие из этих функций — atoi(), atol() и atof() — преобразуют символьный массив ASCII соответственно в значения типа int, long и double. Ниже приведен пример использования функции atoi() (две другие функции вызываются аналогичным образом):

```
//: C03:ArgsToInts.cpp
// Приведение аргументов командной строки к типу int
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
for(int i = 1; i < argc; i++)
cout << atoi(argv[i]) << endl;
} ///:~
```

В этой программе количество аргументов командной строки может быть любым. Обратите внимание: счетчик цикла for начинается с 1, чтобы в преобразование не включалось имя программы в элементе argv[0]. Если включить в командную строку вещественное число с десятичной точкой, функция atoi() обработает

только цифры до десятичной точки. Если командная строка содержит нечисловые аргументы, функция atoi() вернет для них нулевые значения.

Формат вещественных чисел

Функция printBinary(), представленная ранее в этой главе, хорошо подходит для исследования внутренней структуры различных типов данных. Наибольший интерес представляет формат вещественных чисел, позволяющий хранить представления как очень больших, так и очень малых величин в относительно небольшой области памяти. Если описывать подробно все детали, двоичное представление float и double состоит из трех частей: экспоненты, мантиссы и бита знака. Таким образом, данные хранятся в экспоненциальной (научной) записи. При помощи следующей программы немного поэкспериментируйте с разными вещественными числами, проанализируйте их двоичные представления и попробуйте самостоятельно вычислить схему представления вещественных чисел (обычно для вещественных чисел характерен формат IEEE, но ваш компилятор может использовать другой формат):

```
//: C03:FloatingAsBinary.cpp
//{L} printBinary
//{T} 3.14159
#include "printBinary.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
if(argc != 2) {
cout << "Must provide a number" << endl;
exit(1);
}
double d = atof(argv[1]);
unsigned char* cp =
reinterpret_cast<unsigned char*>(&d);
for(int i = sizeof(double)-1; i >= 0 ; i -= 2){
printBinary(cp[i-1]);
printBinary(cp[i]);
}
} ///:~
```

Сначала программа убеждается в том, что при вызове ей был передан аргумент. Для этого проверяется значение argc, которое для одного аргумента равно 2 (а без аргументов оно равно 1, поскольку в первом элементе массива argv всегда хранится имя программы). Если проверка завершается неудачей, программа выводит сообщение и завершается вызовом функции exit() стандартной библиотеки C.

Программа берет аргумент из командной строки и с помощью функции atof() приводит последовательность символов к типу double. Тип double интерпретируется как массив байтов, для чего его адрес приводится к типу unsigned char*. Затем каждый байт передается функции printBinary() для вывода.

В данном примере байты выводятся в порядке, при котором вывод начинается со знакового бита (на компьютере автора). В вашем случае порядок может быть другим; возможно, вам придется изменить порядок вывода. Также учтите, что разобраться в форматах вещественного представления может быть нелегко. Например, экспонента и мантисса в общем случае не выравниваются по границам байтов,

а для каждой части резервируется некоторое количество битов, которые по возможности плотно упаковываются в памяти. Чтобы понять, что же происходит в действительности, необходимо выяснить размер каждой части числа (знак всегда занимает один бит, но экспонента и мантисса имеют переменные размеры) и выводить биты каждой части по отдельности.

Математические операции с указателями

Если бы возможности указателей на массивы ограничивались альтернативными средствами обращения к массиву, то они не представляли бы особого интереса. Однако в действительности указатели обладают большей гибкостью, потому что их можно переводить на разные элементы массива (но помните, что идентификатор массива всегда ссылается только на его начало).

Математические операции с указателями рассматриваются отдельно от обычных математических операций, поскольку с указателями эти операции ведут себя нестандартно. Например, с указателями часто используется оператор ++, который «увеличивает указатель на единицу». Однако в действительности указатель должен переместиться к *следующему элементу* массива, независимо от его размера. Пример:

```
//: C03:PointerIncrement.cpp
#include <iostream>
using namespace std;

int main() {
int i[10];
double d[10];
int* ip = i;
double* dp = d;
cout << "ip = " << (long)ip << endl;
ip++;
cout << "ip = " << (long)ip << endl;
cout << "dp = " << (long)dp << endl;
dp++;
cout << "dp = " << (long)dp << endl;
} ///:~
```

При одном из запусков на компьютере автора был получен следующий результат:

```
ip = 6684124
ip = 6684128
dp = 6684044
dp = 6684052
```

Хотя с int* и double* вроде бы выполняется одна и та же операция ++, из результатов видно, что для int* указатель сместился всего на 4 байта, а для double* — на целых 8 байтов. Смещения соответствуют размерам int и double на компьютере автора, и это не простое совпадение. В этом и кроется вся хитрость вычислений с указателями: компилятор определяет смещение, при котором указатель будет указывать на следующий элемент массива (математические операции с указателями имеют смысл только при работе с массивами). Это правило выполняется даже для массивов структур:

```
//: C03:PointerIncrement2.cpp
#include <iostream>
using namespace std;

typedef struct {
```

```
char c;
short s;
int i;
long l;
float f;
double d;
long double ld;
} Primitives;

int main() {
Primitives p[10];
Primitives* pp = p;
cout << "sizeof(Primitives) = "
<< sizeof(Primitives) << endl;
cout << "pp = " << (long)pp << endl;
pp++;
cout << "pp = " << (long)pp << endl;
} ///:~
```

Результат одного из запусков программы на компьютере автора:

```
sizeof(Primitives) = 40
pp = 6683764
pp = 6683804
```

Как видите, компилятор правильно работает с указателями на массивы структур (а также на массивы классов и объединений).

Операторы --, + и - тоже правильно работают с указателями, но возможности последних двух операторов ограниченны: суммирование двух указателей запрещено, а при вычитании указателей результат равен количеству элементов между двумя указателями. Тем не менее допускается сложение или вычитание указателя и целого числа. Следующий пример показывает, как выполняются математические вычисления с указателями:

```
//: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;

#define P(EX) cout << #EX << ": " << EX << endl;

int main() {
int a[10];
for(int i = 0; i < 10; i++)
a[i] = i; // Присвоить значения индексов
int* ip = a;
P(*ip);
P(*++ip);
P(*(ip + 5));
int* ip2 = ip + 5;
P(*ip2);
P(*(ip2 - 4));
P(*--ip2);
P(ip2 - ip); // Возвращает количество элементов
} ///:~
```

Программа начинается с очередного макроса, в котором используется новая возможность препроцессора, которая называется *преобразованием в строку* и обозначается символом #: выражение, следующее за этим знаком, преобразуется в символьный массив. Это довольно удобно, поскольку макрос может сначала вывести

выражение в текстовом виде, а затем через двоеточие вывести значение этого выражения.

Указатели могут использоваться как с префиксной, так и с постфиксной формой операторов ++ и --, но в примере встречаются только префиксные версии, потому что они применяются до разыменования указателей и позволяют увидеть эффект от выполненных операций. В операции сложения участвуют только указатель и целое число; компилятор не разрешит сложить два указателя.

Результат выполнения программы:

```
*ip: 0
*++ip: 1
*(ip + 5): 6
*ip2: 6
*(ip2 - 4): 2
*--ip2: 5
```

Во всех случаях указатель автоматически смещается на нужную величину в зависимости от размера элементов, на которые он указывает.

Если вычисления с указателями покажутся вам слишком сложными, не огорчайтесь. На практике обычно достаточно создать массив и обращаться к его элементам при помощи конструкции [ ], а самым сложным при операциях с указателями является применение операторов ++ и --. Как правило, математические операции с указателями широко представлены только в сложных и нетривиальных программах. Многие контейнеры стандартной библиотеки С++ скрывают от программиста все технические детали и избавляют его от лишних забот.

Рекомендации по отладке

В идеальной рабочей среде в вашем распоряжении окажется превосходный отладчик, который сделает поведение вашей программы абсолютно прозрачным и поможет быстро обнаружить ошибки. Однако у большинства отладчиков имеются свои слабые места, а это означает, что вам придется вставлять в программу специальные фрагменты кода, чтобы разобраться в происходящем. Кроме того, в некоторых средах — скажем, во встроенных системах — отладчик вообще недоступен, а обратная связь сведена к минимуму (например, ограничивается однострочным светодиодным индикатором). В таких случаях требуется творческий подход к диагностике и выводу информации о выполнении программы. Настоящий раздел содержит описание ряда полезных приемов, которые помогут вам в этом.

Флаги отладки

Жесткое включение отладочного кода в программу может создать проблемы. Увеличение объема выводимой информации затрудняет поиск и локализацию ошибок. Когда кажется, что ошибка успешно найдена, вы начинаете убирать отладочный код, а потом выясняется, что его нужно вставлять снова. Проблемы такого рода решаются при помощи двух разновидностей флагов: отладочных флагов препроцессора и отладочных флагов времени выполнения.

Отладочные флаги препроцессора

Отладочные флаги препроцессора задаются директивой #define (желательно это делать в заголовочном файле). Далее вы можете проверить состояние флага командой #ifdef и произвести условное включение отладочного кода. Завершив процедуру отладки, достаточно просто отменить определения флагов директивой #undef, и отладочный код будет автоматически удален. Тем самым сокращаются размеры и время выполнения исполняемого файла.

Имена отладочных флагов лучше выбрать в начале работы над проектом. Флаги препроцессора традиционно записываются символами верхнего регистра, чтобы они визуально отличались от переменных. На практике часто встречается имя DEBUG (будьте внимательны и не используйте имя NDEBUG, зарезервированное в C). Примерная последовательность команд выглядит так:

```
#define DEBUG // Вероятно, в заголовочном файле
// ...
#ifdef DEBUG // Проверить, определен ли данный флаг
/* отладочный код */
#endif // DEBUG
```

Многие реализации C и C++ также позволяют определять и отменять определения флагов из командной строки компилятора, поэтому перекомпиляция программы и вставка отладочной информации может производиться одной командой (желательно с применением make-файлов, о которых речь пойдет далее). За подробностями обращайтесь к документации.

Отладочные флаги времени выполнения

В некоторых ситуациях удобнее устанавливать и сбрасывать флаги во время выполнения программы, особенно если информация о состоянии флагов передается в командной строке. Никому не хочется заново компилировать большую программу только для того, чтобы вставить в нее отладочный код.

Чтобы использовать отладочные флаги времени выполнения, определите в программе переменные типа bool:

```
//: C03:DynamicDebugFlags.cpp
#include <iostream>
#include <string>
using namespace std;
// Отладочные флаги не обязательно объявлять глобальными!
bool debug = false;

int main(int argc, char* argv[]) {
for(int i = 0; i < argc; i++)
if(string(argv[i]) == "--debug=on")
debug = true;
bool go = true;
while(go) {
if(debug) {
// Отладочный код
cout << "Debugger is now on!" << endl;
} else {
cout << "Debugger is now off." << endl;
}
cout << "Turn debugger [on/off/quit]: ";
string reply;
```

```
cin >> reply;
if(reply == "on") debug = true; // Установка флага
if(reply == "off") debug = false; // Сброс флага
if(reply == "quit") break; // Выход из 'while'
}
} ///:~
```

Программа устанавливает и сбрасывает отладочный флаг, пока пользователь не введет команду quit, приказывая программе завершиться. Обратите внимание: вводить нужно целые слова, а не первые буквы (впрочем, при желании вы можете изменить программу и сократить команды до букв). При помощи специального аргумента командной строки можно включить отладку в начале сеанса — этот аргумент может находиться в любой позиции командной строки, поскольку функция main() просматривает все аргументы. Все сводится к простой проверке выражения:

```
string(argv[i])
```

Мы берем символьный массив argv[i] и создаем на его основе объект string, который затем легко сравнивается с выражением в правой части оператора ==. Приведенная выше программа ищет строку --debug = on. Чтобы расширить возможности программы, также можно найти выражение --debug = и установить режим отладки в зависимости от следующей за ним подстроки. Во втором томе целая глава посвящена классу string из стандартной библиотеки С++.

Хотя флаги отладки принадлежат к тем относительно немногочисленным областям, в которых оправданно применение глобальных переменных, ничто не заставляет вас действовать именно таким способом. Обратите внимание: имя переменной записывается символами нижнего регистра, напоминая читателю, что переменная не является флагом препроцессора.

Преобразование переменных и выражений в строки

При написании отладочного кода часто приходится решать одну и ту же скучную задачу вывода выражений, состоящих из символьного массива с именем переменной и значения этой переменной. К счастью, в стандарте С++ предусмотрен препроцессорный оператор #, который уже использовался в этой главе. Если поставить знак # перед аргументом препроцессорного макроса, то препроцессор преобразует этот аргумент в символьный массив. В сочетании с тем фактом, что смежные символьные массивы объединяются в один символьный массив[1], это позволяет создать очень удобный отладочный макрос для вывода значений переменных:

```
#define PR(x) cout << #x " = " << x << "\n";
```

Вызов макроса PR(a) эквивалентен выполнению следующей команды:

```
cout << "a = " << a << "\n";
```

Макрос работает не только с именами отдельных переменных, но и с целыми выражениями. В следующей программе он используется для вывода выражения в строковом виде с последующим вычислением и выводом результата:

```
//: C03:StringizingExpressions.cpp
#include <iostream>
```

[1] См. с. 82. — *Примеч. перев.*

```
using namespace std;

#define P(A) cout << #A << ": " << (A) << endl;

int main() {
int a = 1, b = 2, c = 3;
P(a); P(b); P(c);
P(a + b);
P((c - a)/b);
} ///:~
```

Подобные приемы оказывают неоценимую помощь, особенно при недоступности отладчика (или если разработка ведется в нескольких средах). Также можно вставить дополнительную директиву #ifdef, которая при отключенном режиме отладки определяет P(A) как пустой макрос.

Макрос assert() языка С

В стандартном заголовочном файле <cassert> определен удобный отладочный макрос assert(). При вызове макросу assert() передается аргумент с выражением, которое, как предполагается, должно быть истинным. Препроцессор генерирует код проверки этого выражения. Если условие окажется ложным, программа выводит сообщение о проверяемом условии и о неудачной проверке, после чего завершает работу. Тривиальный пример:

```
//: C03:Assert.cpp
// Использование отладочного макроса assert()
#include <cassert> // Заголовочный файл, содержащий макрос
using namespace std;

int main() {
int i = 100;
assert(i != 100); // Неудача
} ///:~
```

Этот макрос описан в стандарте С, поэтому он также доступен в заголовочном файле assert.h.

Чтобы после завершения отладки удалить код, сгенерированный макросом, вставьте в программу перед включением <cassert> следующую директиву:

```
#define NDEBUG
```

Можно также определить флаг NDEBUG в командной строке компилятора. От состояния флага NDEBUG зависит то, какой код будет сгенерирован макросами заголовочного файла <cassert>.

Позднее в книге будут представлены более совершенные альтернативы макросу assert().

Адреса функций

Функция, откомпилированная и загруженная в память компьютера для выполнения, занимает некоторую область памяти. Эта область памяти (а следовательно, и сама функция) обладает определенным адресом.

Язык С никогда не запрещал программисту делать то, что считалось слишком рискованным в других языках. Адреса функций могут применяться в указателях

наряду с адресами переменных. На первый взгляд синтаксис объявления и использования указателей на функции выглядит излишне усложненным, но он соответствует синтаксису других конструкций языка.

Определение указателя на функцию

Указатель на функцию, которая вызывается без аргументов и не имеет возвращаемого значения, определяется так:

```
void (*funcPtr)();
```

Столкнувшись с подобным сложным определением, лучше всего взяться за него с середины и постепенно двигаться наружу. «Серединой» в данном случае является имя переменной, то есть funcPtr. В процессе «движения наружу» вы сначала смотрите направо (где ничего нет — закрывающая круглая скобка завершает выражение), затем налево (звездочка — признак указателя), затем снова направо (пустой список аргументов обозначает функцию, которая вызывается без аргументов) и снова налево (ключевое слово void указывает на то, что функция не имеет возвращаемого значения). Такие перемещения справа налево и слева направо подходят для большинства объявлений.

Итак, еще раз. Мы начинаем с середины («funcPtr — это...»), двигаемся направо (тупик — закрывающая круглая скобка), налево (звездочка — «...указатель на ...»), направо (пустой список аргументов — «...функцию, которая вызывается без аргументов...») и на последнем перемещении влево находим ключевое слово void. Получается, «funcPtr — это указатель на функцию, которая вызывается без аргументов и возвращает void».

Возможно, вас интересует, зачем *funcPtr заключается в круглые скобки. Если убрать их, то компилятор увидит выражение

```
void *funcPtr();
```

Но эта строка объявляет не переменную, а функцию, которая возвращает void*! Представьте, что в процессе анализа определения или объявления компилятор действует так, как описано выше. Круглые скобки нужны, чтобы компилятор «столкнулся» с ними и отправился налево к символу *, вместо того чтобы продолжать двигаться направо к пустому списку аргументов.

Сложные определения и объявления

Разобравшись в том, как работает синтаксис объявлений C и C++, вы сможете создавать гораздо более сложные конструкции. Пример:

```
//: C03:ComplicatedDefinitions.cpp

/* 1. */ void * (*(*fp1)(int))[10];

/* 2. */ float (*(*fp2)(int,int,float))(int);

/* 3. */ typedef double (*(*(*fp3)())[10])();
fp3 a;

/* 4. */ int (*(*f4())[10])();

int main() {} ///:~
```

Проанализируйте каждое объявление и разберитесь в нем, используя уже знакомый прием «движения справа налево». Например, объявление 1 гласит: «fp1 — указатель на функцию, которая получает аргумент типа int и возвращает указатель на массив с 10 указателями на void».

Объявление 2 означает: «fp2 — указатель на функцию, которая получает три аргумента (int, int и float) и возвращает указатель на функцию, получающую аргумент int и возвращающую float».

Если вы создаете много сложных объявлений, стоит подумать о ключевом слове typedef. Объявление 3 показывает, как ключевое слово typedef избавляет от необходимости многократно вводить сложные описания. Оно означает следующее: «fp3 — указатель на функцию, которая вызывается без аргументов и возвращает указатель на массив из 10 указателей на функции, вызываемые без аргументов и возвращающие double». Далее следует определение: «Переменная a относится к типу fp3». Ключевое слово typedef особенно часто используется для построения сложных определений из более простых.

Объявление 4 объявляет функцию вместо определения переменной. Оно гласит: «f4 — функция, которая возвращает указатель на массив из 10 указателей на функции, возвращающие тип int».

На практике такие сложные объявления и определения встречаются очень редко. Но стоит лишь один раз потратить усилия и досконально разобраться в них, и у вас уже никогда не возникнет затруднений с умеренно сложными конструкциями, встречающимися в реальной жизни.

Использование указателя на функцию

После определения указателя на функцию перед дальнейшим использованием ему необходимо присвоить адрес функции. Подобно тому как адрес массива arr[10] задается именем массива без квадратных скобок (arr), адрес функции func() задается именем функции без списка аргументов (func). Допустим и более очевидный синтаксис &func(). Чтобы вызвать функцию, разыменуйте указатель так же, как это делалось при его объявлении (и помните: C и C++ всегда стремятся к тому, чтобы синтаксис объявлений как можно более походил на синтаксис использования). Следующий пример показывает, как определить и применить указатель на функцию:

```
//: C03:PointerToFunction.cpp
// Определение и использование указателя на функцию
#include <iostream>
using namespace std;

void func() {
cout << "func() called..." << endl;
}

int main() {
void (*fp)(); // Определение указателя на функцию
fp = func; // Инициализация
(*fp)(); // Разыменование означает вызов функции
void (*fp2)() = func; // Определение и инициализация
(*fp2)();
} ///:~
```

После определения указателя на функцию fp ему присваивается адрес функции func() командой fp = func (обратите внимание: аргументы после имени функции

не указываются). Во второй части продемонстрировано определение указателя, совмещенное с инициализацией.

Массивы указателей на функции

Стоит особо упомянуть такую интересную конструкцию, как *массив указателей на функции*. Чтобы вызвать нужную функцию, следует проиндексировать массив и разыменовать полученный указатель. Этот механизм воплощает концепцию *кода, управляемого таблицами*: вместо условных команд или конструкций выбора выполняемая функция выбирается в зависимости от переменной состояния (или комбинации переменных состояния). Такая архитектура особенно удобна при частом добавлении или удалении функций из таблицы, а также при динамическом создании или изменении таблицы.

В следующем примере создается несколько фиктивных функций при помощи препроцессорного макроса, после чего создается массив указателей на эти функции с автоматической инициализацией по списку. Как нетрудно убедиться, добавление или удаление функций из таблицы (а следовательно, изменение функциональных возможностей программы) достигается минимальным объемом программного кода:

```
//: C03:FunctionTable.cpp
// Использование массива указателей на функции
#include <iostream>
using namespace std;

// Макрос для определения фиктивных функций:
#define DF(N) void N() { \
cout << "function " #N " called..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

void (*func_table[])() = { a, b, c, d, e, f, g };

int main() {
while(1) {
cout << "press a key from 'a' to 'g' "
"or q to quit" << endl;
char c, cr;
cin.get(c); cin.get(cr); // Вторая операция - для возврата курсора
if ( c == 'q' )
break; // ... Выход из while(1)
if ( c < 'a' || c > 'g' )
continue;
(*func_table[c - 'a'])();
}
} ///:~
```

Нетрудно представить себе возможные применения этой методики для создания интерпретаторов или программ обработки списков.

Утилита make и управление раздельной компиляцией

При *раздельной компиляции* (то есть разбиении программы на несколько единиц трансляции) необходимо каким-то образом автоматизировать компиляцию всех

файлов и сборку компоновщиком всех частей (вместе с необходимыми библиотеками и кодом запуска) в исполняемый файл. В большинстве компиляторов эта задача решается всего одной командой. Например, в компиляторе GNU C++ эта команда может выглядеть так:

```
g++ SourceFile1.cpp SourceFile2.cpp
```

Недостаток такого подхода заключается в том, что компилятор компилирует все файлы независимо от того, *нужно* их перестраивать заново или нет. Если проект содержит большое количество файлов, полная перекомпиляция при изменении одного файла может оказаться неприемлемой.

Эта проблема была изначально решена для системы Unix и затем в той или иной форме — для всех платформ. Речь идет о служебной программе make. Утилита make организует управление всеми отдельными файлами проекта, для чего она выполняет инструкции из текстового файла, называемого *make-файлом*. Если отредактировать некоторые файлы проекта и ввести команду make, утилита make выполняет инструкции make-файла и сравнивает даты модификации файлов, содержащих исходные тексты, с датами модификации соответствующих целевых файлов. Если файл с исходным текстом был модифицирован позже своего целевого файла, make вызывает для него компилятор. Утилита make перекомпилирует только те исходные файлы, которые были изменены, а также все файлы с исходными текстами, зависящими от измененных файлов. Благодаря утилите make вам не придется заново компилировать все файлы проекта при каждой модификации одного файла или убеждаться в том, что программа была построена верно. Make-файл содержит все команды сборки проекта. Умение пользоваться утилитой make сэкономит вам массу времени и усилий. Кроме того, make обычно используется при установке новых программ в системе Linux/Unix (хотя эти make-файлы гораздо сложнее тех, что представлены в книге, а make-файл для конкретного компьютера обычно генерируется автоматически в процессе установки).

Утилита make в той или иной форме поддерживается практически всеми компиляторами C++, но даже если она и не поддерживается, всегда можно найти бесплатную реализацию make для любого компилятора. По этой причине она упоминается в книге. Впрочем, разработчики компиляторов также создают собственные утилиты для построения проектов. Обычно такие утилиты получают информацию о файлах, входящих в проект, и самостоятельно определяют все связи между ними. В их работе используется некий аналог make-файла, чаще всего называемый *файлом проекта*, но содержимое этого файла находится под управлением среды программирования, поэтому программист может не беспокоиться о нем. Особенности конфигурирования и использования файла проекта зависят от конкретной среды программирования, поэтому вам придется самостоятельно найти нужную документацию. Впрочем, средства для работы с файлами проектов, предоставляемые разработчиками компиляторов, обычно настолько просты, что легко осваиваются простым методом проб и ошибок (любимый способ освоения новых пакетов автором).

Make-файлы, приведенные в книге, обычно работают даже при наличии специализированных инструментов построения файлов проектов.

Команда make

Когда вы вводите команду make (или то имя, под которым она известна в вашей системе), утилита make ищет в текущем каталоге файл с именем makefile. Этот файл

содержит информацию о зависимостях между файлами с исходными текстами. Утилита make проверяет дату модификации каждого файла. Если зависимый файл был создан раньше, чем файл, от которого он зависит, утилита выполняет *правило*, указанное в make-файле после описания зависимости.

Все комментарии в make-файлах начинаются со знака # и продолжаются до конца строки.

Рассмотрим простой пример. Make-файл программы «hello» может выглядеть так:

```
# Комментарий
hello.exe: hello.cpp
mycompiler hello.cpp
```

Это означает, что файл hello.exe (целевой файл) зависит от файла hello.cpp. Если hello.cpp имеет более позднюю дату модификации, чем hello.exe, утилита make выполняет «правило» mycompiler hello.cpp. Файл может содержать информацию о нескольких зависимостях или правилах. Многие версии make требуют, чтобы все правила начинались с символа табуляции. Остальные пропуски обычно игнорируются, чтобы make-файлы можно было форматировать для удобства чтения.

Правила не ограничиваются вызовом компилятора. Из make-файла можно запустить любую программу по вашему усмотрению. Определив группы взаимозависимых наборов правил, вы можете отредактировать файл с исходным текстом программы, ввести команду make и быть уверенным в том, что утилита сама построит все необходимые файлы.

Макросы

Make-файлы могут содержать *макросы* (которые не имеют ничего общего с препроцессорными макросами C/C++). Макросы в make-файлах упрощают операции замены строк. Например, в make-файлах настоящей книги используется макрос для вызова компилятора C++:

```
CPP = mycompiler
hello.exe: hello.cpp
$(CPP) hello.cpp
```

Знак = определяет макрос CPP, а знак $ и круглые скобки означают расширение макроса. В данном случае это говорит о том, что вызов макроса $(CPP) будет заменен строкой mycompiler. Если вдруг потребуется перейти на другой компилятор с именем cpp, достаточно отредактировать макрос и привести его к виду:

```
CPP = cpp
```

В макрос также можно добавить флаги компилятора и т. д. или же определить для них отдельные макросы.

Правила суффиксов

Было бы утомительно заново сообщать утилите make, как вызывать компилятор для каждого файла проекта с расширением cpp, если каждый раз это делается одним и тем же способом. Поскольку утилита make разрабатывалась для экономии времени, в ней также предусмотрена возможность сокращенной записи для операций, зависящих от расширения (суффикса) файла. Такие сокращения называются *правилами суффиксов*. Правило суффикса сообщает make, как преобра-

зовать файл с одним расширением (например, .cpp) в файл с другим расширением (.obj или .exe). После того как make научится строить файлы одного типа на основании файлов другого типа, остается лишь сообщить, как связаны между собой файлы проекта. Когда утилита make находит файл с более ранней датой модификации, чем у файла, от которого он зависит, новый файл строится при помощи правила.

Правила суффиксов сообщают make, что построение всех файлов проекта не требует явного задания правил и утилита может сама определить способ построения различных файлов на основании их расширения. В нашем примере правило суффикса означает следующее: «Чтобы построить файл с расширением exe на основании файла с расширением cpp, нужно вызвать следующую команду». Вот как выглядит это правило для предыдущего примера:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
$(CPP) $<
```

Директива .SUFFIXES сообщает утилите make, что та должна особо отслеживать все перечисленные расширения файлов, потому что они имеют особый смысл в данном make-файле. Далее следует правило суффикса .cpp.exe, которое означает: «Преобразование любого файла с расширением cpp в файл с расширением exe (если файл с расширением cpp модифицировался позже файла с расширением exe) должно производиться так». Как и прежде, в make-файле используется макрос $(CPP), но теперь в нем появилось нечто новое — $<. Раз эта последовательность символов начинается с $, она является макросом, но этот макрос принадлежит к числу специальных встроенных макросов make. Комбинация $< используется только в правилах суффиксов и означает «причину, по которой сработало правило», что в данном случае преобразуется в «файл с расширением cpp, который необходимо откомпилировать».

После настройки правил суффиксов достаточно ввести команду вида make Union.exe; утилита задействует правила, несмотря на то что имя Union вообще не упоминается в make-файле.

Цели по умолчанию

После макросов и правил суффиксов утилита make ищет в файле первую *цель* и строит ее, если в файле прямо не указано обратное. Для примера рассмотрим следующий make-файл:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
$(CPP) $<
target1.exe:
target2.exe:
```

Если пользователь просто введет команду make, будет построен файл target1.exe (на основе правила суффикса по умолчанию), поскольку именно эта цель будет встречена первой при обработке файла. Чтобы построить файл target2.exe, необходимо ввести команду make target2.exe. Но вводить несколько команд неудобно, поэтому в make-файлах обычно создается фиктивная цель, которая зависит от всех остальных целей:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
$(CPP) $<
all: target1.exe target2.exe
```

Файла с именем all не существует, поэтому при каждом вызове make программа находит в списке первую цель all, убеждается в том, что файла с именем all не существует, и приступает к проверке зависимостей. Она переходит к цели target1.exe и (по правилам суффиксов) проверяет, во-первых, существует ли файл target1.exe и, во-вторых, имеет ли файл target1.exe более позднюю дату модификации, чем target1.cpp. Если хотя бы одно условие не выполняется, утилита задействует правило суффикса (или специальные правила, если они были заданы для этой конкретной цели). Затем make переходит к следующему файлу в списке целей. Таким образом, создание фиктивной цели (которая обычно называется all, но может называться как угодно) позволяет построить все исполняемые файлы проекта простым вводом команды make. В файл также можно включить отдельные списки целей для решения других задач, например, чтобы команда make debug заново строила все файлы с включением отладочного режима.

Make-файлы данной книги

Все листинги программ этой книги были автоматически извлечены из текстовой версии книги и распределены по подкаталогам в соответствии с номерами глав. Для этого использовалась программа ExtractCode.cpp, описанная во втором томе. Кроме того, программа ExtractCode.cpp создает в каждом подкаталоге несколько make-файлов с разными именами, чтобы вы могли просто перейти в этот каталог и ввести следующую команду (mycompiler заменяется именем компилятора, а флаг -f означает «то, что следует дальше, следует интерпретировать как make-файл»):

```
make -f mycompiler.makefile
```

Наконец, программа ExtractCode.cpp создает в корневом каталоге примеров «главный» make-файл, который поочередно переходит к каждому каталогу и вызывает make с соответствующим make-файлом. Таким образом, вы можете откомпилировать *все* примеры настоящей книги одной командой make. Процесс прерывается в том случае, если компилятору не удастся обработать некоторый файл (хотя компилятор, соответствующий требованиям стандарта C++, должен успешно откомпилировать все файлы в книге). Поскольку возможности make различаются в зависимости от реализации, в сгенерированных make-файлах используются лишь самые простые и распространенные средства.

Пример make-файла

Как упоминалось ранее, программа ExtractCode.cpp автоматически генерирует make-файлы для каждой главы книги, поэтому в книге не будут приводиться make-файлы отдельных примеров (все make-файлы наряду с исходными текстами включены в архив примеров, который можно загрузить с сайта www.piter.com). И все же читателю будет полезно познакомиться с примером make-файла. Ниже приведена

сокращенная версия такого файла, автоматически сгенерированного для одного из примеров этой главы. Каждый подкаталог содержит несколько make-файлов с разными именами; нужный make-файл вызывается командой make -f. Следующий файл предназначен для компилятора GNU C++:

```
CPP = g++
OFLAG = -o
.SUFFIXES : .o .cpp .c
.cpp.o :
    $(CPP) $(CPPFLAGS) -c $<
.c.o :
    $(CPP) $(CPPFLAGS) -c $<

all: \
    Return \
Declare \
    Ifthen \
    Guess \
    Guess2 \
# Остальные файлы этой главы не показаны

Return: Return.o
    $(CPP) $(OFLAG) Return Return.o

Declare: Declare.o
    $(CPP) $(OFLAG) Declare Declare.o

Ifthen: Ifthen.o
    $(CPP) $(OFLAG)Ifthen Ifthen.o

Guess: Guess.o
    $(CPP) $(OFLAG)Guess Guess.o

Guess2: Guess2.o
    $(CPP) $(OFLAG)Guess2 Guess2.o

Return.o: Return.cpp
Declare.o: Declare.cpp
Ifthen.o: Ifthen.cpp
Guess.o: Guess.cpp
Guess2.o: Guess2.cpp
```

Макрос CPP содержит имя компилятора. Чтобы использовать другой компилятор, следует отредактировать make-файл или изменить значение макроса в командной строке:

```
make CPP=cpp
```

Впрочем, программа ExtractCode.cpp автоматически строит make-файлы для разных компиляторов.

Второй макрос OFLAG определяет имя выходного файла. Хотя многие компиляторы по умолчанию предполагают, что имя выходного файла совпадает с именем входного файла, существуют исключения (например, компиляторы для Linux/Unix по умолчанию создают файл с именем a.out).

В приведенном файле определены два правила суффиксов для файлов с расширениями .cpp и .c (на случай, если потребуется откомпилировать файлы с исходными текстами C). По умолчанию используется фиктивная цель all, а в конце

каждой следующей строки ставится знак продолжения \ до цели Guess2, которая завершает список и поэтому не сопровождается знаком \. Подкаталог примеров этой главы содержит еще много файлов, но они не приводятся в листинге ради экономии места.

Правила суффиксов обеспечивают создание объектных файлов (с расширением .o) из файлов .cpp, но обычно правила создания исполняемых файлов задаются явно, потому что исполняемый файл, как правило, компонуется из нескольких объектных файлов, а утилита make не знает, из каких именно. Кроме того, на платформе Linux/Unix исполняемые файлы не имеют стандартного расширения, поэтому правила суффиксов не подходят для этих простых ситуаций. Становится понятно, почему правила построения исполняемых файлов указываются в явном виде.

В этом make-файле выбран самый безопасный путь с минимальным набором возможностей make; в сущности, в нем задействованы только базовые концепции целей и зависимостей, а также макросы. Такой подход практически гарантирует совместимость с большинством существующих версий make. Обычно он приводит к увеличению объема make-файла, но это не так страшно, поскольку файл все равно автоматически генерируется программой ExtractCode.cpp.

Утилита make обладает многими другими возможностями, не используемыми в книге. Кроме того, появились более новые, усовершенствованные версии и разновидности make с сокращениями, способными сэкономить много времени программисту. За информацией о расширенных возможностях вашей версии make обращайтесь к документации. Если разработчик компилятора не включает make в комплект поставки или использует нестандартную версию, воспользуйтесь GNU-версией make, поддерживаемой практически на всех известных платформах. Для этого проведите поиск по многочисленным архивам GNU в Интернете.

Итоги

В этой главе довольно кратко были рассмотрены все основные синтаксические конструкции С++, большая часть которых унаследована из языка С (чем, собственно, и объясняется широко известная совместимость языков С++ и С). Этот краткий курс в первую очередь предназначался для читателей с опытом программирования, для которых достаточно вводного описания основ синтаксиса С и С++. Впрочем, даже программисты С могут узнать в нем нечто новое, не говоря уже о специфических (и потому почти наверняка им не известных) возможностях С++.

Упражнения

1. Создайте заголовочный файл с расширением .h. Объявите в нем группу функций с разными списками аргументов и возвращаемыми значениями типов void, char, int и float. Создайте файл .cpp, который включает заголовочный файл и определения всех функций. Каждое определение должно просто выводить имя функции, список аргументов и тип возвращаемого значе-

ния, чтобы вы могли убедиться в том, что функция была вызвана успешно. Создайте второй файл .cpp, включите в него заголовочный файл и определите функцию int main() с вызовами всех функций. Откомпилируйте и запустите программу.

2. Напишите программу, которая при помощи двух вложенных циклов for и оператора вычисления остатка (%) находит и выводит простые числа (то есть целые числа, которые нацело делятся только на себя и на 1).

3. Напишите программу, которая при помощи цикла while читает слова из стандартного ввода (cin) в объект string. Цикл while должен быть бесконечным, а выход из него осуществляться по команде break. Каждое прочитанное слово при помощи серии команд if сначала «отображается» на целое число, которое затем используется командой switch в качестве критерия выбора (не стоит полагать, что такая последовательность событий является хорошим стилем программирования; она нужна только для тренировки в работе с управляющими конструкциями). Внутри каждой секции case должно выводиться какое-нибудь осмысленное сообщение. Самостоятельно выберите «особые» слова и их интерпретацию. Также решите, какое слово должно сообщать о завершении программы. Проверьте работу программы и попробуйте перенаправить стандартный ввод команды в чтение из файла (чтобы вам не пришлось вводить лишние данные, воспользуйтесь файлом с исходным текстом программы).

4. Измените программу Menu.cpp так, чтобы в ней использовались команды switch вместо if.

5. Напишите программу для вычислений значений двух выражений из подраздела «Приоритет» в разделе «Знакомство с операторами».

6. Измените программу YourPets2.cpp так, чтобы в ней использовались разные типы данных (char, int, float, double и их производные). Запустите программу и постройте карту распределения памяти. Если у вас имеется доступ к другим компьютерам, компиляторам или операционным системам, повторите эксперимент в максимально возможном количестве вариантов.

7. Создайте две функции: первая должна получать аргумент типа string*, а вторая — аргумент типа string&. Каждая функция должна изменять внешний объект string своим способом. В функции main() создайте и инициализируйте объект string, выведите его содержимое, затем передайте каждой из двух функций и выведите результаты.

8. Напишите программу, в которой используются все триграфы, и проверьте, поддерживаются ли они вашим компилятором.

9. Откомпилируйте и запустите программу Static.cpp. Удалите из программы ключевое слово static, снова откомпилируйте и запустите ее. Объясните результат.

10. Попробуйте откомпилировать и скомпоновать программу FileStatic.cpp с программой FileStatic2.cpp.Что означает полученное сообщение об ошибке?

11. Измените программу Boolean.cpp так, чтобы она работала с типом double вместо int.

12. Измените программы Boolean.cpp и Bitwise.cpp так, чтобы в них использовались синонимы поразрядных и логических операторов (синонимы поддерживаются всеми компиляторами, соответствующими стандарту С++).
13. Измените программу Bitwise.cpp так, чтобы в ней использовались функции из программы Rotate.cpp. Представление результатов должно четко и наглядно показывать, что происходит при циклическом сдвиге.
14. Измените программу Ifthen.cpp так, чтобы в ней использовался тернарный условный оператор (?:).
15. Создайте структуру, содержащую два объекта string и переменную int. Определите псевдоним typedef для имени структуры. Создайте экземпляр структуры, инициализируйте все три поля экземпляра и выведите их значения. Получите адрес экземпляра и присвойте его переменной-указателю на тип вашей структуры. Измените значения полей в экземпляре и выведите их через указатель.
16. Напишите программу с перечисляемым типом, содержащим разные варианты цветов. Создайте переменную этого типа и выведите все числа, соответствующие названиям цветов, в цикле for.
17. Поэкспериментируйте с программой Union.cpp и посмотрите, как удаление тех или иных элементов влияет на размер объединения. Попробуйте присвоить значение одному элементу объединения, выведите разные элементы (то есть элементы других типов) и посмотрите, что при этом происходит.
18. Напишите программу, в которой сразу же друг за другом определяются два массива int. Сместите индекс от конца первого массива во второй массив и выполните присваивание. Выведите содержимое второго массива и убедитесь в том, что оно изменилось. Попробуйте определить переменную типа char между двумя определениями массивов и повторите эксперимент. Вероятно, для простоты стоит создать вспомогательную функцию вывода массива.
19. Измените программу ArrayAddresses.cpp так, чтобы она работала с данными типов char, long int, float и double.
20. Используя приемы, освоенные при написании программы ArrayAddresses.cpp, организуйте вывод размера структуры и адресов элементов в программе StructArray.cpp.
21. Создайте массив объектов string и присвойте строку каждому элементу. Выведите содержимое массива в цикле for.
22. Создайте на базе программы ArgsToInts.cpp две новые программы, использующие функции atol() и atof().
23. Измените программу PointerIncrement2.cpp так, чтобы вместо структуры в ней использовалось объединение.
24. Измените программу PointerArithmetic.cpp, чтобы она работала с типами long и long double.
25. Определите переменную типа float. Получите ее адрес, приведите его к типу unsigned char и присвойте указателю на unsigned char. При помощи этого ука-

зателя и конструкции [ ] организуйте смещение индекса по переменной типа float и при помощи функции printBinary(), определенной в этой главе, выведите значения отдельных байтов переменной (смещение должно происходить от 0 до sizeof(float)-1). Измените значение float и попробуйте разобраться во внутреннем представлении числа (данные типа float хранятся в специальном формате).

26. Определите массив типа int. Получите начальный адрес массива и приведите его к типу void* при помощи оператора static_cast. Напишите функцию, в аргументах которой передаются void*, число (количество байтов) и значение. Функция должна присваивать передаваемое значение каждому байту в заданном интервале. Протестируйте работу функции на массиве int.
27. Создайте const-массив double и volatile-массив double. Переберите элементы каждого массива в цикле, при помощи оператора const_cast отмените для каждого элемента атрибуты const и volatile и присвойте ему значение.
28. Напишите функцию, которая получает указатель на массив double и размер этого массива; функция должна выводить значения всех элементов массива. Создайте массив double, проинициализируйте все элементы нулями и выведите массив при помощи функции. При помощи оператора reinterpret_cast преобразуйте начальный адрес массива в unsigned char* и присвойте каждому байту массива значение 1 (количество байтов в double должно определяться оператором sizeof). Снова выведите содержимое массива. Как вы думаете, почему элементам не было присвоено значение 1.0?
29. (Задача повышенной трудности) Измените программу FloatingAsBinary.cpp так, чтобы она выводила каждую часть double в виде отдельной группы битов. Для этого вызовы printBinary() придется заменить вызовами специальной функции (для создания которой можно взять за основу printBinary()). Вам придется изучить формат представления вещественных чисел и разобраться с порядком байтов на вашем компьютере (собственно, в этом и заключается трудность).
30. Создайте make-файл, который не только компилирует программы YourPets1.cpp и YourPets2.cpp (для вашего компилятора), но и запускает обе программы в контексте определения цели по умолчанию. Используйте правила суффиксов.
31. Измените программу StringizingExpressions.cpp и организуйте условное определение P(A) (#ifdef), чтобы отладочный код можно было исключать установкой флага в командной строке. За информацией о том, как определять и отменять препроцессорные данные в командной строке компилятора, обращайтесь к документации по компилятору.
32. Определите функцию, которая получает аргумент типа double и возвращает int. Создайте и инициализируйте указатель на эту функцию, затем вызовите ее через указатель.

33. Объявите указатель на функцию, которая получает аргумент типа int и возвращает указатель на функцию с аргументом типа char и возвращаемым значением float.
34. Измените программу FunctionTable.cpp так, чтобы функции вместо вывода сообщения возвращали объекты string. Содержимое этих объектов должно выводиться в функции main().
35. Напишите для одного из предыдущих упражнений (по вашему выбору) make-файл. По команде make должна строиться рабочая версия программы, а по команде make debug — версия с включением отладочного кода.

Абстрактное представление данных

Язык C++ должен повышать производительность труда программиста. Спрашивается, почему вы прилагаете усилия (а изучение требует усилий, как бы мы ни старались упростить переход с одного языка на другой) и переходите с хорошо знакомого языка на другой язык да еще со *снижением* производительности, которое продолжается, пока новый язык в должной мере не освоен? Только потому, что вы знаете: затраты на освоение нового инструмента с лихвой окупятся.

Повышение производительности труда в контексте программирования означает, что меньшее количество программистов способно написать более сложные программы за меньший промежуток времени. Конечно, при выборе языка приходится учитывать и другие факторы, например эффективность (не приводит ли специфика языка к замедлению скорости выполнения программ и разрастанию их объема?), безопасность (поможет ли язык сделать так, чтобы программа всегда делала то, что должна делать, и корректно справлялась с ошибками?) и удобство сопровождения (позволяет ли язык написать программный код, который легко понять, изменить и дополнить?). Несомненно, эти факторы важны, и мы неоднократно вернемся к ним в книге.

Но на житейском уровне «повышение производительности» означает, что программа, над которой раньше трое программистов трудились в течение недели, теперь пишется одним программистом за пару дней. Выгода достигается сразу на нескольких уровнях. Доволен программист, потому что он чувствует гордость за успешно написанную программу. Доволен заказчик, потому что продукты создаются быстрее меньшим количеством людей. Довольны клиенты, потому что они получают продукты по более низкой цене. Но единственный способ добиться радикального повышения производительности заключается в широком применении готового кода, то есть библиотек.

Библиотека представляет собой набор фрагментов программного кода, специально упакованных и готовых к использованию. Обычно в минимальном варианте библиотека состоит из файла с расширением lib и одного или нескольких заголо-

вочных файлов, передающих компилятору информацию о содержимом библиотеки. Компоновщик умеет искать нужную информацию в библиотечном файле и извлекать из него нужные фрагменты откомпилированного кода. Впрочем, это лишь один из вариантов распространения библиотек. В некоторых архитектурах (например, Linux/Unix) библиотеки нередко приходится распространять на уровне исходных текстов, чтобы их можно было переконфигурировать и перекомпилировать для новой цели.

Итак, библиотеки относятся к числу важнейших инструментов повышения производительности, а удобство работы с библиотеками было одной из целей разработки языка C++. Напрашивается предположение, что использование библиотек в C было сопряжено с какими-то трудностями, и это действительно так. Хорошее понимание этой темы поможет вам лучше разобраться в архитектуре C++ и в том, как правильно пользоваться библиотеками.

Маленькая библиотека в стиле C

В исходном варианте библиотека обычно представляет собой просто набор функций. Но каждый программист, которому доводилось пользоваться библиотеками C, знает, что библиотека — это не только операции и функции, но и атрибуты (то есть данные). А при работе с наборами атрибутов в языке C очень удобно группировать их в структурах, особенно если в пространстве задачи нужно представить несколько сходных наборов. В таких случаях для каждого набора создается переменная структурного типа (struct).

Итак, многие библиотеки C содержат определения структур и функций, работающих с этими структурами. Чтобы вы лучше представляли, как устроены подобные системы, рассмотрим структуру данных, сходную с массивом, но с размером, который может задаваться при создании экземпляра структуры во время выполнения программы. Автор назвал эту структуру CStash. Хотя приведенная ниже программа написана на C++, стиль программирования полностью соответствует C.

```
//: C04:CLib.h
// Заголовочный файл для библиотеки в стиле C.
// Структура CStash является аналогом массива.
// но создается на стадии выполнения

typedef struct CStashTag {
  int size;      // Размер каждого элемента
  int quantity;  // Количество элементов
  int next;      // Следующий пустой элемент
  // Динамически выделяемый байтовый массив:
  unsigned char* storage;
} CStash;

void initialize(CStash* s, int size);
void cleanup(CStash* s);
int add(CStash* s, const void* element);
void* fetch(CStash* s, int index);
int count(CStash* s);
void inflate(CStash* s, int increase);
///:~
```

Синонимы типа CStashTag обычно определяются для структур на тот случай, если структура должна содержать ссылку на саму себя. Например, в *связанных списках* каждый элемент содержит указатель на следующий элемент, поэтому структура должна содержать указатель на следующую структурную переменную в списке, то есть программист должен иметь возможность сослаться на этот тип указателя в теле структуры. Кроме того, практически для всех структур в библиотеках С с помощью ключевого слова typedef определяются псевдонимы вроде приведенных выше. Это делается для того, что структуру можно было использовать как новый тип и определять структурные переменные в конструкциях вида

```
CStash A, B, C;
```

Указатель storage относится к типу unsigned char*. Тип unsigned char определяет минимальную единицу памяти, поддерживаемую компилятором С, хотя на некоторых компьютерах ее размер совпадает с размером максимальной единицы. Размер типа unsigned char зависит от реализации, но обычно он равен одному байту. Может показаться, что, поскольку структура CStash предназначена для хранения данных произвольного типа, в данной ситуации было бы уместно воспользоваться указателем void*. Однако в нашем случае память должна интерпретироваться не как блок неизвестного типа, а как блок смежных байтов.

Ниже приведен исходный текст файла реализации. Кстати, при покупке коммерческой библиотеки клиент не всегда получает исходные тексты — иногда разработчик предоставляет только откомпилированный файл с расширением obj, lib, dll и т. д.

```
//: C04:CLib.cpp {O}
// Реализация библиотеки в стиле C
// Объявление структуры и функций:
#include "CLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Количество элементов, добавляемых при увеличении размера буфера:
const int increment = 100;

void initialize(CStash* s, int sz) {
  s->size = sz;
  s->quantity = 0;
  s->storage = 0;
  s->next = 0;
}

int add(CStash* s, const void* element) {
  if(s->next >= s->quantity) // В буфере есть свободное место?
    inflate(s, increment);
  // Скопировать элемент в следующую свободную позицию буфера:
  int startBytes = s->next * s->size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < s->size; i++)
    s->storage[startBytes + i] = e[i];
  s->next++;
  return(s->next - 1); // Индекс
}

void* fetch(CStash* s, int index) {
```

```
  // Проверка индекса:
  assert(0 <= index);
  if(index >= s->next)
    return 0; // Признак конца
  // Указатель на запрашиваемый элемент:
  return &(s->storage[index * s->size]);
}

int count(CStash* s) {
  return s->next;  // Количество элементов в CStash
}

void inflate(CStash* s, int increase) {
  assert(increase > 0);
  int newQuantity = s->quantity + increase;
  int newBytes = newQuantity * s->size;
  int oldBytes = s->quantity * s->size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = s->storage[i]; // Копирование старого буфера в новый
  delete [](s->storage); // Освобождение старого буфера
  s->storage = b; // Перевод указателя на новый буфер
  s->quantity = newQuantity;
}

void cleanup(CStash* s) {
  if(s->storage != 0) {
   cout << "freeing storage" << endl;
   delete []s->storage;
  }
} ///:~
```

Функция initialize() инициализирует структуру CStash и назначает внутренним переменным исходные значения. Первоначально указателю storage присваивается нулевое значение (память еще не выделена).

Функция add() вставляет элемент в следующую свободную позицию CStash. Сначала она проверяет свободное место в буфере и, если буфер заполнен, расширяет его вызовом описанной далее функции inflate().

Компилятор не знает типа данных, хранящихся в буфере (функция получает только void*), поэтому сохранение не может быть выполнено простым присваиванием (что, конечно, было бы удобно). Простейший механизм копирования основан на индексировании массива. Обычно буфер storage уже содержит байты данных, на что указывает значение next. Смещение внутри буфера startBytes вычисляется умножением next на размер каждого элемента (в байтах). Затем аргумент element приводится к типу unsigned char*, чтобы его можно было адресовать по байтам, и копируется в свободную позицию storage. Далее функция увеличивает переменную next, чтобы она ссылалась на следующую свободную позицию, и возвращает «индекс», под которым было сохранено значение, для его последующей выборки функцией fetch().

Функция fetch() сначала проверяет, что индекс не выходит за границы массива, а затем возвращает адрес нужной переменной, вычисленный на основании аргумента index. Поскольку смещение index задается в количестве элементов, то для получения нужного числового смещения в байтах аргумент умножается на количество байтов, занимаемых одним элементом. Однако при индексировании

storage с вычисленным смещением (с применением стандартного механизма индексирования массивов) будет получен не адрес, а байт, находящийся по этому адресу. Для получения адреса необходимо использовать оператор получения адреса &.

С точки зрения опытного программиста C, функция count выглядит несколько странно. Кажется, будто она усложненным способом делает то, что легко сделать вручную. Например, если в программе имеется экземпляр структуры CStash с именем intStash, то для получения количества элементов гораздо проще воспользоваться конструкцией intStash.next, обойдясь без вызова функции count(&intStash) и связанных с этим вызовом издержек. Если в будущем неожиданно потребуется изменить внутреннее представление CStash и способ вычисления количества элементов, гибкий интерфейс с вызовом функции позволит это сделать. К сожалению, многие программисты совершенно не интересуются вашим мнением о «правильной» архитектуре библиотеки. Они видят определение структуры и напрямую получают значение next, а может быть, даже изменяют его без вашего ведома. Если бы проектировщик библиотеки мог хоть как-то контролировать подобные события!

Динамическое распределение памяти

Мы не знаем, сколько памяти может потребоваться для хранения CStash, поэтому блок памяти, на которую ссылается указатель storage, выделяется из *кучи*. Кучей (heap) называется большой блок памяти, используемый для выделения блоков меньшего размера во время выполнения программы. Куча задействуется в тех случаях, когда размер необходимой памяти неизвестен во время написания программы (например, если только во время выполнения программы выясняется, что в памяти должно храниться 200 переменных типа Airplane вместо 20). В стандарте C определены функции динамического выделения памяти malloc(), calloc(), realloc() и free(). Однако в C++ вместо вызова библиотечных функций используется более совершенный (и притом более простой) механизм динамического выделения памяти, интегрированный в язык в виде ключевых слов new и delete.

Функция inflate() при помощи ключевого слова new выделяет увеличенный блок памяти для структуры CStash. В нашем примере объем памяти только увеличивается и никогда не сокращается, поэтому вызов assert() проверяет, что в аргументе функции inflate() не было передано отрицательное число. Функция вычисляет значение newQuantity — новое количество элементов, которые поместятся в буфере после вызова inflate(), и умножает его на размер одного элемента в байтах. Полученное значение newBytes определяет общий размер блока памяти. Чтобы узнать, сколько байтов необходимо скопировать из старого буфера, мы вычисляем oldBytes по старому значению quantity.

Непосредственное выделение памяти выполняется в *выражении new* (так называется выражение, в котором находится ключевое слово new):

```
new unsigned char[newBytes];
```

В общем виде выражение new выглядит так:

```
new Type;
```

Здесь Type — тип переменной, память для которой выделяется из кучи. В нашем примере память выделяется для массива unsigned char, длина которого равна newBytes;

именно этот тип указан в приведенном выше выражении. Впрочем, выражения new могут использоваться даже для выделения памяти под встроенные типы:

```
new int;
```

На практике это делается редко, но зато подтверждает логическое единство синтаксиса.

Выражение new возвращает *указатель* на объект того типа, который был указан в выражении. Следовательно, если программа содержит выражение new Type;, вы получите указатель на Type (выражение new int вернет указатель на int и т. д.). Следовательно, если память выделяется для массива new unsigned char, вы получите указатель на первый элемент массива. Компилятор проследит за тем, чтобы возвращаемое значение выражения new было присвоено указателю правильного типа.

Конечно, любой запрос на выделение дополнительной памяти может завершиться неудачей, если в системе не осталось свободной памяти. Как вы вскоре узнаете, в C++ имеются специальные механизмы, которые активизируются в случае неудачных операций выделения памяти.

После выделения нового блока памяти необходимо скопировать существующие данные из старого буфера в новый. Как и прежде, задача решается индексированием массива и последовательным копированием байтов в цикле. Когда все данные будут скопированы, старую память следует освободить, чтобы она могла повторно использоваться программой. Ключевое слово delete составляет пару с new и применяется для освобождения любой памяти, выделенной ключевым словом new (если забыть о вызовах delete, память останется недоступной; при достаточно большом количестве *утечек памяти* вся свободная память в системе будет исчерпана). При освобождении памяти массивов используется специальный вариант синтаксиса. В сущности, вы напоминаете компилятору, что указатель ссылается не на один объект, а на массив объектов; для этого перед указателем на освобождаемую память ставятся квадратные скобки:

```
delete []myArray;
```

После освобождения старого буфера остается лишь присвоить указатель на новый буфер переменной storage и изменить переменную quantity. На этом работа функции inflate() заканчивается.

Учтите, что система динамического распределения памяти устроена весьма примитивно. Она предоставляет блоки памяти по запросу и забирает их обратно при вызове delete. В C++ не существует встроенных средств *уплотнения кучи*, которые бы производили дефрагментацию и освобождали свободные блоки большего размера. В результате многочисленных операций выделения и освобождения памяти может произойти *фрагментация*: куча будет содержать много свободной памяти, но ни одного блока, размер которого был бы достаточен для удовлетворения текущего запроса. Уплотнение усложняет программы, так как в результате перемещения блоков памяти указатели теряют свои исходные значения. В некоторых средах программирования имеются встроенные средства уплотнения динамической памяти, но вместо указателей в них используются специальные *манипуляторы* (которые могут быть временно преобразованы в указатели — для этого нужно *зафиксировать* блок памяти, чтобы он не перемещался при уплотнении). Вы также можете самостоятельно реализовать схему уплотнения памяти, но эта задача не из простых.

При создании стековой переменной во время компиляции программы память для нее автоматически выделяется и освобождается компилятором. Компилятору точно известен объем необходимой памяти и время существования переменной, определяемое по правилам видимости. Однако при динамическом выделении памяти компилятор не знает, сколько памяти *и* на какой срок потребуется. Это означает, что автоматическое освобождение памяти невозможно. Программист должен сам вызвать delete и тем самым сообщить подсистеме управления динамической памятью, что память данного блока может выделяться при следующих вызовах new. В библиотеках логичнее всего делать это в функции cleanup(), где выполняются все завершающие операции.

Чтобы протестировать библиотеку, мы создадим два экземпляра структуры CStash. В первом экземпляре будут храниться данные типа int, а во втором — массивы из 80 символов:

```
//: C04:CLibTest.cpp
//{L} CLib
// Тестирование библиотеки в стиле С
#include "CLib.h"
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
using namespace std:

int main() {
  // Переменные определяются в начале блока. как в языке С:
  CStash intStash. stringStash:
  int i:
  char* cp:
  ifstream in:
  string line:
  const int bufsize = 80:
  // Не забудьте инициализировать переменные:
  initialize(&intStash. sizeof(int)):
  for(i = 0: i < 100: i++)
    add(&intStash. &i):
  for(i = 0: i < count(&intStash): i++)
    cout << "fetch(&intStash. " << i << ") = "
         << *(int*)fetch(&intStash. i)
         << endl:
  // Строки из 80 символов:
  initialize(&stringStash. sizeof(char)*bufsize):
  in.open("CLibTest.cpp"):
  assert(in):
  while(getline(in. line))
    add(&stringStash. line.c_str()):
  i = 0:
  while((cp = (char*)fetch(&stringStash.i++))!=0)
    cout << "fetch(&stringStash. " << i << ") = "
         << cp << endl:
  cleanup(&intStash):
  cleanup(&stringStash):
} ///:~
```

В соответствии с правилами языка С все переменные определяются в начале блока main(). Не забывайте, что переменные CStash необходимо инициализировать

вызовом initialize(). Одна из проблем с библиотеками С заключается именно в том, что пользователь должен полностью осознать всю важность функций инициализации и зачистки. Если он забудет вызвать эти функции, это приведет к крайне неприятным последствиям. К сожалению, пользователи часто забывают о важности инициализации и зачистки. Они склонны думать лишь о том, что нужно *им*, и не обращают внимания на отчаянные призывы проектировщика библиотеки: «Постой, сначала нужно сделать *это*!» Более того, некоторые пользователи даже пытаются самостоятельно инициализировать элементы структур! В С не существует средств, которые бы предотвратили подобное поведение.

В структурной переменной intStash хранятся целые числа, а в переменной stringStash — символьные массивы. Чтобы получить эти символьные массивы, мы открываем файл с кодом программы CLibTest.cpp, читаем его содержимое в строковую переменную line, а затем создаем указатель на символьное представление line при помощи функции c_str().

После загрузки обеих структур программа выводит их содержимое. Структура intStash выводится в цикле for, при этом верхняя граница счетчика определяется функцией count(). Структура stringStash() выводится в цикле while(); цикл завершается, когда функция fetch() возвращает 0 (признак выхода за границу массива).

Также обратите внимание на дополнительную операцию приведения типа в команде

```
cp = (char*)fetch(&stringStash.i++)
```

Это преобразование необходимо из-за более жесткой проверки типов в C++, не позволяющей просто присвоить void* любому другому типу (в С это разрешено).

Неверные предположения

Существует еще одно важное обстоятельство, о котором необходимо узнать перед рассмотрением общих проблем создания библиотек С. Заголовочный файл CLib.h *должен* включаться во все файлы, использующие структуру CStash, поскольку компилятор понятия не имеет, как она выглядит. С другой стороны, он *может* предположить, как выглядит функция. На первый взгляд кажется, что такие предположения только помогают программисту, но на самом деле это один из основных недостатков С.

Хотя функции всегда следует заранее объявлять в заголовочных файлах, в языке С эти объявления не так важны. Язык С (но не C++!) позволяет вызвать функцию, которая не была объявлена ранее. Хороший компилятор предупредит о том, что функцию стоило бы предварительно объявить, но стандарт языка С не заставляет выполнять это требование. Возникают опасные ситуации: компилятор С может предположить, что если при вызове функции было передано значение типа int, то ее список аргументов содержит аргумент типа int, тогда как в действительности в нем находится аргумент типа float. Как будет показано далее, такие предположения приводят к трудноуловимым ошибкам.

Каждый отдельный файл реализации С (то есть файл с расширением .c) представляет собой *единицу трансляции*. Компилятор запускается для каждой единицы трансляции и во время своей работы знает о существовании только этой единицы. Вся информация, передаваемая в заголовочных файлах, чрезвычайно важна, поскольку она определяет представления компилятора об остальных частях про-

граммы. Особенно важны объявления в заголовочных файлах, сообщающие компилятору, как он должен действовать в той или иной ситуации. Например, если в заголовочный файл включено объявление void func(float), то компилятор знает, что при вызове функции с аргументом типа int нужно в процессе передачи привести int к типу float. Без объявления компилятор С просто решит, что функция func(int) определяется где-то в другом месте, оставит аргумент без изменений, и в итоге func() получит неверные данные.

Для каждой единицы трансляции компилятор создает объектный файл с расширением .o, .obj или что-нибудь в этом роде. Объектные файлы вместе с необходимым кодом запуска собираются компоновщиком в исполняемую программу. В процессе компоновки должны быть разрешены все внешние ссылки. Например, в файле CLibTest.cpp функции initialize(), fetch() и др. объявляются (то есть компилятор получает информацию о том, как выглядят функции) и используются, но не определяются. Их определения находятся в другом месте, а именно в файле CLib.cpp. Следовательно, вызовы функций из CLib.cpp представляют собой внешние ссылки. В процессе сборки объектных файлов компоновщик должен перебрать неразрешенные внешние ссылки и найти адреса, которым они соответствуют. Эти адреса заменяют внешние ссылки в исполняемой программе. Важно понимать, что в С внешние ссылки, которые ищет компоновщик, представляют собой простые имена функций (обычно снабженные префиксом в виде символа подчеркивания). Компоновщику остается лишь сопоставить имя функции, указанное при вызове, с телом функции в объектном файле, и на этом его работа завершается. Если некий вызов ошибочно интерпретируется компилятором как func(int), а в другом объектном файле существует тело функции func(float), то компоновщик в обоих случаях видит имя _func и решает, что все в порядке. В точке вызова func() в стек заносится int, тогда как тело функции func() ожидает, что в стеке находится float. Если функция только читает значение и изменяет его, структура стека не нарушится. Более того, значение, прочитанное из стека и интерпретированное как float, даже может выглядеть осмысленно. Но это лишь ухудшает ситуацию, поскольку такие ошибки труднее найти.

Конфликты имен

Человек быстро привыкает ко всему... даже к тому, к чему привыкать бы не следовало. Стиль библиотеки CStash выглядит абсолютно естественно для программистов С, но если присмотреться к программе, она начинает казаться несколько... неудобной. Например, при вызове всех функций библиотеки им приходится передавать адрес структуры. А при чтении программы внутренние механизмы работы библиотеки перемешиваются с логикой вызовов функций, и разобраться в происходящем оказывается нелегко.

Однако одна из главных проблем с использованием библиотек в С связана с *конфликтами имен*. В языке С существует единое пространство имен функций; иначе говоря, в процессе поиска имени функции компоновщик просматривает только главный список имен. Кроме того, когда компилятор обрабатывает единицу трансляции, он не может работать с двумя одноименными функциями.

Допустим, вы решили приобрести две библиотеки у двух разных производителей. В каждой библиотеке имеется своя структура, которую необходимо инициа-

лизировать и деинициализировать. Оба разработчика решили, что соответствующие функции должны называться initialize() и cleanup(). Что сделает компилятор C, если включить оба заголовочных файла в одну единицу трансляции? К счастью, он выдаст сообщение об ошибке, уведомив о несовпадении списков аргументов объявленных функций. Но даже если заголовочные файлы не включаются в одну единицу трансляции, у компоновщика все равно возникнут проблемы. Хороший компоновщик обнаружит конфликт имен, но некоторые компоновщики просто используют первое имя функции, найденное при просмотре объектных файлов в порядке их перечисления в списке компоновки (некоторые программисты даже ухитряются задействовать это обстоятельство, поскольку оно позволяет заменять библиотечные функции собственными «доморощенными» версиями).

Так или иначе, вы не сможете использовать две библиотеки C с одноименными функциями. Для решения этой проблемы поставщики библиотек C часто включают в имена функций уникальные последовательности символов. Например, функции initialize() и cleanup() могут превратиться в CStash_initialize() и CStash_cleanup(). Это простое и логичное решение — имя функции, работающей со структурой, «украшается» именем структуры.

Пора сделать наш первый шаг к созданию классов C++. Имена переменных, входящих в структуры, не конфликтуют с именами глобальных переменных. Так почему бы не распространить этот принцип на имена функций, работающих с конкретной структурой? Другими словами, почему бы не сделать функции элементами структур наряду с переменными?

Базовый объект

Именно в этом и заключается «первый шаг» — функции C++ могут принадлежать структурам. Вот как выглядит в языке C++ C-версия CStash, преобразованная в структуру Stash:

```
//: C04:CppLib.h
// Библиотека в стиле C, переведенная на C++

struct Stash {
  int size;      // Размер каждого элемента
  int quantity;  // Количество элементов
  int next;      // Следующий пустой элемент
   // Динамически выделяемый байтовый массив:
  unsigned char* storage;
  // Функции!
  void initialize(int size);
  void cleanup();
  int add(const void* element);
  void* fetch(int index);
  int count();
  void inflate(int increase);
}; ///:~
```

Для начала обратите внимание на отсутствие ключевого слова typedef. Компилятор C++ не требует typedef и сам преобразует в программе имя структуры в имя нового типа (наряду с int, char, float и double).

Все переменные структуры остались точно такими же, как раньше, но теперь функции также оказались в теле структуры. Кроме того, исчез первый аргумент,

передававшийся функциям в C-версии. Вместо того чтобы заставлять программиста передавать адрес структуры в аргументах всех функций, работающих с этой структурой, C++ скрыто делает это за него. Теперь аргументы функций имеют отношение только к тому, что *делает* эта функция, а не к внутренним механизмам ее работы.

Код функций почти не изменился по сравнению с C-версией библиотеки. Количество аргументов осталось прежним (хотя передаваемый адрес структуры остается невидимым, он не исчез), а у каждой функции существует только одно тело. Иначе говоря, следующая конструкция не означает, что за каждой переменной закрепляется отдельная функция add():

```
Stash A, B, C;
```

Итак, сгенерированный код почти не отличается от того, который был бы сгенерирован для C-версии. Интересно заметить, что в нем реализовано «украшение имен», к которому, вероятно, прибегли бы и вы, породив конструкции вида Stash_initialize(), Stash_cleanup() и т. д. Если имя функции находится внутри структуры, компилятор фактически присоединяет имя структуры к имени функции. Это означает, что функция initialize() в структуре Stash не конфликтует ни с функцией initialize() внутри любой другой структуры, ни даже с глобальной функцией initialize(). В большинстве случаев вам вообще не придется думать об «украшении имен» — в программах задействуются самые обычные имена. Но иногда бывает нужно явно указать, что функция (например, initialize()) принадлежит именно структуре Stash, а не какой-либо другой структуре. В частности, такая необходимость возникает при определении функции. Для этой цели в C++ существует оператор из двух двоеточий (::), называемый *оператором уточнения области видимости* (такое название выбрано из-за того, что оператор позволяет определять имена с разной видимостью — глобально или внутри структуры). Например, определить функцию initialize(), принадлежащую структуре Stash, можно с помощью выражения Stash::initialize(int size). Пример использования оператора :: в определениях функций:

```
//: C04:CppLib.cpp {O}
// Библиотека в стиле C, переведенная на C++
// Объявление структуры и функций:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Количество элементов, добавляемых при увеличении размера буфера:
const int increment = 100;

void Stash::initialize(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(const void* element) {
  if(next >= quantity) // В буфере есть свободное место?
    inflate(increment);
  // Скопировать элемент в следующую свободную позицию буфера:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
```

```
    for(int i = 0; i < size; i++)
      storage[startBytes + i] = e[i];
    next++;
    return(next - 1); // Индекс
  }

  void* Stash::fetch(int index) {
    // Проверка индекса:
    assert(0 <= index);
    if(index >= next)
      return 0; // Признак конца
    // Указатель на запрашиваемый элемент:
    return &(storage[index * size]);
  }

  int Stash::count() {
    return next; // Количество элементов в Stash
  }

  void Stash::inflate(int increase) {
    assert(increase > 0);
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
      b[i] = storage[i]; // Копирование старого буфера в новый
    delete []storage; // Освобождение старого буфера
    storage = b; // Перевод указателя на новый буфер
    quantity = newQuantity;
  }

  void Stash::cleanup() {
    if(storage != 0) {
      cout << "freeing storage" << endl;
      delete []storage;
    }
  } ///:~
```

Между C и C++ существует целый ряд принципиальных различий. Во-первых, объявления в заголовочных файлах *обязательны*. C++ не позволит вызвать функцию без ее предварительного объявления, в этом случае компилятор выдаст сообщение об ошибке. Тем самым обеспечивается согласованность вызовов функций с их определениями. Заставляя программиста объявлять функцию перед вызовом, компилятор C++ практически вынуждает использовать заголовочный файл для включения объявления. Если тот же заголовочный файл будет включен в месте определения функций, то компилятор проверит соответствие объявления в заголовочном файле определению функции. Таким образом, заголовочный файл превращается в хранилище объявлений функций и обеспечивает их согласованность во всех единицах трансляции проекта.

Конечно, глобальные функции по-прежнему можно объявлять вручную всюду, где они определяются и используются (впрочем, дело это настолько утомительное, что вряд ли кому-нибудь захочется им заниматься). Однако структуры всегда должны объявляться перед их определением или использованием, а самым удобным местом для размещения определений структур является заголовочный файл (кроме структур, намеренно скрываемых в других файлах).

Функции, принадлежащие структуре Stash, внешне очень похожи на свои прототипы из библиотеки C; они отличаются только наличием оператора :: и скрытой передачей первого аргумента. Конечно, сам аргумент по-прежнему передается, потому что функция должна иметь возможность работать с конкретным экземпляром структуры. Но обратите внимание — уточнения переменных внутри функций тоже исчезли! Вместо s->size = sz; мы используем команду size = sz; и избавляемся от лишнего уточнения s->, которое все равно не делало происходящее более содержательным. Теперь это делает компилятор C++: он берет «скрытый» первый аргумент (адрес структуры, который раньше передавался вручную) и автоматически выполняет уточнение при каждой ссылке на переменную структуры. Это означает, что внутри функции, принадлежащей структуре, на любые члены этой структуры (в том числе и на другие функции) можно ссылаться просто по имени. Компилятор сначала просмотрит имена локальной структуры, а потом продолжит поиск глобальной версии указанного имени. Данная особенность упрощает не только программирование, но и чтение готовых программ.

А что, если вы по какой-либо причине *хотите* узнать адрес структуры? В C-версии библиотеки это делалось просто, поскольку в первом аргументе каждой функции передавался указатель Stash* с именем s. В C++ существует специальное ключевое слово this, которое возвращает адрес текущего экземпляра структуры и является аналогом переменной s в C-версии библиотеки. Так что в принципе можно вернуться к стилистике C и использовать команды вида

```
this->size = Size;
```

Присутствие ключевого слова this совершенно не изменит кода, сгенерированного компилятором, поэтому использовать this в таком контексте не обязательно. Иногда встречаются программы, в которых автор постоянно задействует this->, однако это уточнение не делает программу более содержательной и часто говорит о неопытности программиста. На практике ключевое слово this требуется относительно редко, но если требуется — оно в вашем распоряжении (ключевое слово this будет встречаться в некоторых примерах этой книги).

Остается упомянуть еще об одном отличии. Язык C позволял присвоить указатель void* любому другому указателю без протестов со стороны компилятора:

```
int i = 10;
void* vp = &i;  // Разрешено в C и C++
int* ip = vp;   // Разрешено только в C
```

В C++ такое присваивание запрещено. Почему? Потому что в C информации о типах отведена второстепенная роль, и компилятор позволяет присвоить указатель с неопределенным типом указателю с заданным типом. В C++ дело обстоит иначе. Типы в этом языке очень важны, и компилятор пресекает любые нарушения — особенно если учесть, что в C++ появилась возможность включения функций в структуры. Неосторожная передача указателей на структуры в C++ могла бы привести к тому, что программа вызовет функцию структуры, которая логически не определена в этой структуре! В такой ситуации возможны любые бедствия. Поэтому, хотя C++ позволяет присвоить типу void* указатель на любой тип (собственно, для этого и предназначался тип void*), он *запрещает* присваивать указатель void* указателям на другие типы. Перед присваиванием всегда должно выполняться приведение типа, чтобы читатель программы и компилятор знали, что указатель на void* действительно должен интерпретироваться как указатель на итоговый тип.

Однако при этом возникает другой интересный вопрос. Одной из целей, поставленных при разработке C++, была максимальная совместимость с существующим кодом C для упрощения перехода на новый язык. Тем не менее это не означает, что любой код, допустимый в C, автоматически будет допустим в C++. Многие операции, разрешенные с точки зрения компилятора C, опасны и чреваты ошибками (вы познакомитесь с ними по мере изложения материала). Компилятор C++ в таких случаях выдает предупреждения и сообщения об ошибках, причем в большинстве случаев он скорее помогает программисту, чем мешает ему. Нередко программист долго и безуспешно отлаживает программу на C, но стоит перекомпилировать программу на C++, как компилятор сам обнаруживает ошибку! В C часто выясняется, что программа успешно компилируется, но ее еще нужно заставить правильно работать. В C++ нормально компилируемые программы работают гораздо чаще! Это объясняется тем, что в языке используется гораздо более жесткий механизм проверки типов.

Следующая тестовая программа демонстрирует ряд новых аспектов в использовании версии структуры Stash для C++:

```
//: C04:CppLibTest.cpp
//{L} CppLib
// Тестирование библиотеки C++
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash;
  intStash.initialize(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  // Строки из 80 символов:
  Stash stringStash;
  const int bufsize = 80;
  stringStash.initialize(sizeof(char) * bufsize);
  ifstream in("CppLibTest.cpp");
  assure(in, "CppLibTest.cpp");
  string line;
  while(getline(in, line))
    stringStash.add(line.c_str());
  int k = 0;
  char* cp;
  while((cp =(char*)stringStash.fetch(k++)) != 0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
  intStash.cleanup();
  stringStash.cleanup();
} ///:~
```

Во-первых, обратите внимание на то, что переменные определяются непосредственно перед использованием (см. с. 115). Другими словами, переменные опреде-

ляются в произвольной точке области видимости, а не только в ее начале (как в языке C).

Программа в целом похожа на CLibTest.cpp, но функции структур теперь вызываются при помощи оператора точка (.), перед которым указывается имя переменной. Такой синтаксис удобен, поскольку он имитирует синтаксис выбора полей в структурах. Поскольку обращение к переменной заменяется вызовом функции, после имени следует список аргументов.

Вызовы функций, *фактически* сгенерированные компилятором, гораздо ближе к исходным функциям из библиотеки C. После «украшения имени» и передачи указателя this вызов функции C++ intStash.initialize(sizeof(int),100) заменяется чем-то вроде Stash_initialize(&intStash,sizeof(int),100). Более того, первый компилятор C++ cfront, разработанный в AT&T, генерировал код C, который затем обрабатывался компилятором C. Такой подход позволял легко перенести cfront на любую машину с компилятором C, что способствовало быстрому распространению технологии C++. Но раз компилятор C++ способен генерировать код C (а некоторые компиляторы до сих пор поддерживают такую возможность), значит, должен существовать какой-то способ представления синтаксиса C++ в C.

Еще одно изменение в CLibTest.cpp — включение заголовочного файла require.h, специально написанного для этой книги. Он обеспечивает гораздо более совершенную проверку ошибок по сравнению с той, которую обеспечивает assert(). Файл содержит несколько функций, в том числе встречающуюся в программе функцию assure(), используемую при работе с файлами. Эта функция проверяет, был ли файл успешно открыт, и если нет — выводит в стандартный поток сообщение об ошибке открытия файла (имя которого определяется вторым аргументом функции) и завершает программу. Функции из файла require.h будут часто использоваться в книге, особенно для проверки количества аргументов командной строки и результатов открытия файлов. Они заменяют собой монотонный и малосодержательный код проверки ошибок, но при этом выводят чрезвычайно полезную информацию. Эти функции будут полностью описаны далее.

Понятие объекта

После знакомства с вводным примером стоит сделать шаг назад и разобраться с терминологией. Включение функций в структуры составляет подлинную суть того, что язык C++ добавил в C, и поднимает представление программиста о структурах на принципиально новый уровень. В C структура — не более чем простой конгломерат данных; способ упаковки переменных, позволяющий работать с ними как с единым целым. Трудно рассматривать структуры C как нечто большее, чем простое синтаксическое удобство. Функции, работающие со структурами, определяются отдельно от них. Но при введении функций в структуры последние претерпевают качественное изменение; отныне они обладают не только характеристиками (как структуры C), *но и* поведением. Так вполне естественно возникает концепция объекта — автономной логически обособленной сущности, которая способна запоминать свое состояние *и* действовать.

В C++ объект представляет собой обычную переменную, а в наиболее формальном определении — «имя, связанное с областью памяти» (это более точная формулировка для требования «объект должен обладать уникальным идентификатором»,

что в случае C++ соответствует уникальному адресу памяти). Область памяти содержит данные объекта, а также используется операциями, выполняемыми с этими данными.

К сожалению, в части терминологии между языками не существует полного единства, хотя смысл терминов стал более или менее общепринятым. Также иногда встречаются разногласия относительно того, какие языки можно назвать объектно-ориентированными, хотя сейчас этот вопрос уже достаточно хорошо проработан. Некоторые языки относят к категории *объектно-базированных*; в них, как и в C++, существуют объекты (уже знакомые вам «структуры с функциями»). Однако поддержка объектов не исчерпывает всех требований, предъявляемых к объектно-ориентированным языкам. Языки, ограничивающиеся упаковкой функций в структуры данных, называются объектно-базированными, а не объектно-ориентированными.

Абстрактные типы данных

Упаковка данных вместе с функциями позволяет создавать новые типы данных. Это часто называется *инкапсуляцией*[1]. Существующие типы данных тоже могут состоять из нескольких компонентов, упакованных вместе. Например, тип float состоит из мантиссы, экспоненты и знакового бита. С ним можно выполнять различные операции, например прибавить к другому числу типа float или int и т. д. Следовательно, тип float обладает атрибутами и поведением.

Определение Stash создает новый тип данных. С ним можно выполнять операции add(), fetch() и inflate(). Переменные этого типа создаются конструкцией Stash s, подобно тому, как переменная типа float создается конструкцией float f. Тип Stash тоже обладает характеристиками и поведением. Несмотря на все сходство с «настоящими» встроенными типами данных, мы называем его *абстрактным типом данных* — прежде всего потому, что он позволяет абстрагировать концепцию из пространства задачи в пространство решения. Кроме того, компилятор C++ интерпретирует его как новый тип данных, и если функция при вызове должна получать Stash, то компилятор проследит за тем, чтобы ей передавался именно тип Stash. Таким образом, для абстрактных типов данных (также называемых *пользовательскими типами*) реализуется такой же уровень проверки типов, как для встроенных.

Тем не менее наряду со сходством существуют и различия, которые немедленно проявляются при выполнении операций с объектами. При вызове функции объекта используется синтаксис *объект.функция*(*аргументы*). Но в терминологии объектно-ориентированного программирования это также называется «отправкой сообщения объекту». Следовательно, для объекта Stash s команда s.add(&i) «отправляет сообщение объекту s» и приказывает ему: «вызови для этого значения и самого себя функцию add()». Фактически все объектно-ориентированное программирование сводится к одной фразе: *отправка сообщений объектам*. В сущности, объектно-ориентированная программа всего лишь создает набор объектов и орга-

[1] По поводу этого термина тоже нет единого мнения. Одни используют этот термин в том же контексте, в котором он употребляется в тексте; другие предпочитают обозначать им механизм *управления доступом*, о котором речь пойдет в следующей главе.

низует обмен сообщениями между ними. Конечно, нужно понять, *какими* должны быть объекты и сообщения, но если как следует разобраться в этом вопросе, то дальнейшая реализация на C++ на удивление прямолинейна.

Подробнее об объектах

На семинарах часто задают вопрос: «Сколько памяти занимает объект и как он выглядит»? Ответ: «Примерно так же, как структура в языке C». Более того, код, сгенерированный компилятором C для структуры C (без расширений C++), обычно *в точности* совпадает с кодом, сгенерированным компилятором C++. Вероятно, это успокоит тех программистов C, которые используют в своих программах низкоуровневую информацию о размере и формате структур и почему-то напрямую работают с байтами структур вместо идентификаторов (хотя зависимость от конкретных размеров и формата структуры нарушает переносимость программ).

Размер структуры определяется суммарным размером всех ее членов. Иногда при размещении структуры в памяти компилятор добавляет дополнительные байты выравнивания — это может повысить эффективность выполнения программы. В главе 15 будет показано, что иногда в структуры добавляются «секретные» указатели, но пока об этом можно не думать.

Для получения размеров структуры применяется оператор sizeof. Небольшой пример:

```
//: C04:Sizeof.cpp
// Размеры структур
#include "CLib.h"
#include "CppLib.h"
#include <iostream>
using namespace std;

struct A {
  int i[100];
};

struct B {
  void f();
};

void B::f() {}

int main() {
  cout << "sizeof struct A = " << sizeof(A)
       << " bytes" << endl;
  cout << "sizeof struct B = " << sizeof(B)
       << " bytes" << endl;
  cout << "sizeof CStash in C = "
       << sizeof(CStash) << " bytes" << endl;
  cout << "sizeof Stash in C++ = "
       << sizeof(Stash) << " bytes" << endl;
} ///:~
```

На компьютере автора для первой структуры выводится размер 200, потому что каждая переменная int занимает 2 байта. Структура B является аномальным случаем, потому что она не содержит ни одной переменной. В языке C такие структуры запре-

щены, но в C++ необходимо иметь возможность создавать структуры, предназначенные только для определения видимости имен функций, поэтому в C++ вполне допустимы структуры без данных. Однако результат второй команды выглядит несколько неожиданно: он отличен от нуля. В ранних версиях языка этот размер был равен нулю, но тогда при создании таких объектов возникала странная ситуация: адрес структуры совпадал с адресом объекта, созданного сразу же после нее, поэтому структуру невозможно было отличить от объекта. Одно из основных правил работы с объектами гласит, что каждый объект должен обладать уникальным адресом, поэтому даже структуры без переменных всегда имеют минимальный ненулевой размер.

Два последних вызова sizeof показывают, что размер структуры C++ совпадает с размером эквивалентной структуры C. Язык C++ стремится избежать любых лишних затрат.

Этикет использования заголовочных файлов

Создавая структуру, содержащую функции, вы определяете в программе новый тип данных. В общем случае этот тип должен быть доступным для вас и других программистов. Кроме того, интерфейс (объявления) должен отделяться от реализации (определений функций), чтобы реализацию можно было изменить без перекомпиляции всей системы. Задача решается включением объявлений нового типа в заголовочный файл.

Когда автор только начинал программировать на C, заголовочные файлы были для него полной загадкой. Многие книги по C не объясняли, зачем они нужны, а компилятор не требовал обязательного объявления функций, поэтому обычно казалось, что без заголовочных файлов вполне можно обойтись (кроме программ, в которых объявлялись структуры). В C++ стало абсолютно ясно, для чего нужны заголовочные файлы. Они практически обязательны для упрощения разработки программ, и в них включается предельно конкретная информация — объявления. Заголовочный файл сообщает компилятору, что доступно в вашей библиотеке. Использование библиотеки возможно даже в том случае, если вместе с объектным или библиотечным файлом у вас имеется только заголовочный файл; исходные тексты в файлах с расширением .cpp не нужны. Вся спецификация интерфейса содержится в заголовочном файле.

Оптимальный (хотя и не обязательный с точки зрения компилятора) подход к построению больших проектов на C основан на использовании библиотек. Логически связанные функции собираются в объектный модуль или библиотеку, а все объявления функций включаются в заголовочный файл. В этом проявляется специфика C++; в библиотеки C можно включать любые функции, а в C++ абстрактные типы данных определяют функции, логические связанные по признаку общего доступа к данным структуры. Любая функция, принадлежащая структуре, должна быть объявлена в объявлении структуры; объявить ее в другом месте нельзя. Если язык C поощрял использование библиотек функций, то в C++ оно стало почти обязательным.

О важности заголовочных файлов

При использовании функций, входящих в библиотеку, C позволяет проигнорировать заголовочный файл и объявить функцию вручную. В прошлом программи-

сты иногда поступали так, чтобы немного ускорить работу компилятора и избавить его от необходимости открывать и включать файлы (проблема, которая обычно отсутствует в современных компиляторах). Например, минимальное объявление функции printf() языка C из библиотеки <stdio.h> выглядит так:

```
printf(...);
```

Многоточие указывает на *переменный список аргументов*, а в целом объявление означает следующее: «функция printf() при вызове получает аргументы, каждый аргумент относится к некоторому типу, но на это не нужно обращать внимания; просто прими те аргументы, которые будут переданы». В результате проверка типов аргументов фактически приостанавливается.

Подобные объявления способствуют появлению нетривиальных проблем. Объявляя функции вручную, вы можете допустить ошибку. Поскольку компилятор видит в файле только ваше объявление, возможно, он сможет как-то приспособиться к ошибке. Программа будет компоноваться правильно, но в этом конкретном файле функция будет использоваться неправильно. Такие ошибки очень трудно обнаружить, а заголовочные файлы позволяют легко избежать их.

Размещая все объявления функций в заголовочном файле и включая этот заголовочный файл во все файлы, в которых используются или определяются функции, вы обеспечиваете логическую согласованность объявлений в системе. Кроме того, включение заголовка в файл определения гарантирует соответствие объявлений и определений.

При объявлении структуры в заголовочном файле C++ вы *обязаны* включать заголовочный файл всюду, где используется структура и определяются ее функции. Компилятор C++ выдаст сообщение об ошибке, если вы попытаетесь вызвать обычную функцию или вызвать/определить функцию, принадлежащую структуре, без предварительного объявления. Заставляя правильно работать с заголовочными файлами, язык гарантирует согласованность библиотек и сокращает количество ошибок благодаря всеобщему использованию одного и того же интерфейса.

Заголовочный файл определяет *контракт* между разработчиком библиотеки и ее пользователем. Контракт описывает структуры данных, утверждает типы аргументов и возвращаемых значений при вызовах функций. Фактически он говорит: «Моя библиотека должна применяться так». Часть этой информации нужна пользователю для разработки приложения, а компилятору она нужна полностью, чтобы сгенерировать правильный код. Пользователь структуры просто включает заголовочный файл, создает объекты (экземпляры) структурного типа и подключает на стадии компоновки объектный модуль или библиотеку (то есть откомпилированный код).

Компилятор гарантирует соблюдение контракта, заставляя пользователя объявлять все структуры и функции перед использованием, а в случае функций-членов — перед их определением. Следовательно, пользователь будет вынужден поместить объявления в заголовочный файл и включить его в тот файл, где определяются функции-члены, и в те файлы, где они задействуются. Поскольку во всей системе имеется единственный заголовочный файл, описывающий библиотеку, компилятор может удостовериться в логической согласованности операций с библиотекой и предотвратить ошибки.

Для правильной организации кода и эффективного написания заголовочных файлов необходимо учитывать некоторые обстоятельства. Прежде всего, нужно

подумать над тем, что включать в заголовочные файлы. Основное правило рекомендует ограничиться одними объявлениями (то есть информацией для компилятора) и исключить все, что связано с выделением памяти посредством создания переменных или генерирования программного кода. Дело в том, что заголовочный файл обычно включается в нескольких единицах трансляции одного проекта, и если память для идентификатора выделяется в нескольких местах, компоновщик выдаст ошибку множественного определения (принцип *единственного определения* в C++: объявлений может быть сколько угодно, но определение должно быть только одно).

Впрочем, у этого правила имеются исключения. Если в заголовочном файле определяется переменная с *файловой видимостью* (то есть переменная, видимая только в данном файле), то проект будет содержать несколько экземпляров этих данных, но компоновщик не обнаружит конфликта[1]. Более общее правило гласит, что заголовочные файлы не должны содержать ничего, что могло бы вызвать неоднозначность во время компоновки.

Проблема многократного объявления

Второе обстоятельство, которое должно учитываться при использовании заголовочных файлов: при размещении объявления структуры в заголовочном файле может оказаться, что файл многократно включается в сложную программу. Хорошим примером служит библиотека потоков ввода-вывода. Любая структура, выполняющая операции ввода-вывода, может включать один из заголовочных файлов iostream. Если в cpp-файл, с которым вы работаете, включено несколько таких структур (и для каждой структуры включается свой заголовочный файл), возникает риск многократного включения заголовочного файла <iostream> и множественного объявления его содержимого.

Компилятор считает повторное объявление структур и классов ошибкой, поскольку иначе появилась бы возможность использования одного имени для разных типов. Чтобы предотвратить ошибки при многократном включении заголовочного файла, необходимо наделить его «интеллектом» при помощи директив препроцессора (в стандартных заголовочных файлах C++, таких как <iostream>, такой «интеллект» уже есть).

Языки C и C++ разрешают заново определять функции, если объявления совпадают, но ни один из этих языков не разрешает заново определять структуры. В C++ это правило особенно важно. Если компилятор разрешит переопределять структуры и два объявления будут различаться, какое из объявлений ему выбрать?

Проблема переобъявления возникает в C++ довольно часто, поскольку каждый тип данных (структура с функциями) обычно имеет собственный заголовочный файл. Если вы создаете новый тип данных, который использует первый тип, вам придется включить один заголовочный файл в другой. В любом cpp-файле может возникнуть ситуация с включением нескольких заголовочных файлов, в которые, в свою очередь, включен один и тот же заголовочный файл. В процессе компиляции заголовочный файл многократно встретится компилятору. Если не принять никаких мер, компилятор воспримет происходящее как повторное объявление

[1] Стандарт C++ не рекомендует использовать файловую видимость, эта концепция считается устаревшей.

структуры и сообщит об ошибке. Чтобы решить эту проблему, необходимо чуть больше узнать о препроцессоре.

Директивы препроцессора #define, #ifdef и #endif

Препроцессорная директива #define может потребоваться для создания флагов компиляции. Она существует в двух вариантах. Можно просто сообщить препроцессору об определении флага без указания значения:

```
#define FLAG
```

А можно связать имя со значением (типичный способ определения констант в C):

```
#define PI 3.14159
```

В обоих вариантах существование имени может быть проверено другой директивой препроцессора:

```
#ifdef FLAG
```

Если проверка дает истинный результат, то код, следующий за директивой #ifdef, будет включен в пакет, переданный компилятору. Включение прекращается в тот момент, когда препроцессор встретит одну из следующих директив:

```
#endif
#endif // FLAG
```

Любые продолжения директивы #endif в той же строке, кроме комментариев, недопустимы, хотя некоторые компиляторы могут поддерживать их. Пары #ifdef/ #endif могут быть вложенными.

Действие директивы #define отменяется директивой #undef. В результате ее выполнения директива #ifdef с той же переменной возвращает ложный результат. Еще директива #undef заставляет препроцессор прекратить использование макроса. У директивы #ifdef тоже имеется парная директива #ifndef, которая считается истинной, если имя не определено (в частности, эта директива входит в пример рассматриваемого далее заголовочного файла).

Препроцессор C поддерживает немало других полезных возможностей. За полной информацией обращайтесь к документации компилятора.

Стандартное устройство заголовочного файла

В каждом заголовочном файле, содержащем объявление структуры, следует сначала проверить, не был ли этот заголовочный файл уже включен в текущий cpp-файл. Это делается при помощи препроцессорного флага. Если флаг не установлен, значит, файл еще не включался; установите флаг (чтобы предотвратить возможное переобъявление структуры в будущем) и объявите структуру. Если флаг уже установлен, значит, тип уже был объявлен ранее, и код с его объявлением просто игнорируется. Примерный формат заголовочного файла:

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
 // Объявление типа...
#endif // HEADER_FLAG
```

Как видите, при первом включении заголовочного файла его содержимое (вместе с определением типа) будет обработано препроцессором. При всех последующих включениях (в рамках той же единицы трансляции) объявление типа игнорируется.

Имя HEADER_FLAG можно заменить любым уникальным именем, но существует надежная схема выбора имен: записать имя заголовочного файла символами верхнего регистра и заменить точки символами подчеркивания (впрочем, символы подчеркивания в начале имени резервируются для системных имен). Пример:

```
//: C04:Simple.h
// Простой заголовочный файл, предотвращающий повторное объявление
#ifndef SIMPLE_H
#define SIMPLE_H

struct Simple {
  int i,j,k;
  initialize() { i = j = k = 0; }
};
#endif // SIMPLE_H ///:~
```

Хотя имя SIMPLE_H после #endif закомментировано и игнорируется препроцессором, оно делает программу более понятной.

Директивы препроцессора, предотвращающие многократное включение, часто называются *стражами включений* (include guards).

Пространства имен в заголовках

Возможно, вы обратили внимание на *директивы using*, присутствующие почти во всех cpp-файлах книги, обычно в форме

```
using namespace std;
```

Пространство имен std содержит всю стандартную библиотеку C++, поэтому эта директива позволяет обращаться к именам стандартной библиотеки C++ без уточнения. Однако директивы using практически никогда не встречаются в заголовочных файлах (по крайней мере, за пределами области видимости). Дело в том, что директива using снимает защиту с указанного пространства имен и продолжает действовать до конца текущей единицы компиляции. Если за пределами области видимости включить директиву using в заголовочный файл, это будет означать, что потеря «защиты пространства имен» распространится на все файлы, в которые включен данный заголовок, а среди них могут оказаться другие заголовочные файлы. Следовательно, включение директив using в заголовочные файлы легко приводит к практически повсеместной «отмене» пространств имен.

Короче говоря, не используйте директивы using в заголовочных файлах.

Использование заголовков в проектах

Разработка проекта в C++ обычно сопровождается объединением множества разных типов (структур данных с ассоциированными функциями). Как правило, объявления каждого типа или группы логически связанных типов находятся в отдельном заголовочном файле, после чего функции этого типа определяются в единице трансляции. В случае использования типа заголовочный файл также включается в программу для правильной передачи объявлений.

Некоторые из приводимых в книге примеров строятся по этому образцу, но чаще примеры являются чрезвычайно компактными, поэтому все компоненты программы — объявления структур, определения функций и функция main() — ока-

зываются в одном файле. Тем не менее следует помнить, что на практике обычно используются отдельные файлы и заголовки.

Вложенные структуры

Удобный механизм выделения данных и имен функций из глобального пространства имен распространяется на структуры. Структуру можно объявить внутри другой структуры, чтобы сгруппировать логически связанные элементы. Синтаксис объявления выглядит так, как следовало ожидать; в этом убеждает следующий пример структуры, в котором стек реализуется на базе простого связанного списка, чтобы в нем «никогда» не кончалась свободная память:

```
//: C04:Stack.h
// Вложенная структура в связанном списке
#ifndef STACK_H
#define STACK_H

struct Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK_H ///:~
```

Вложенная структура с именем Link содержит указатель next на следующую структуру Link в списке и указатель на данные, хранящиеся в Link. Если указатель next равен нулю, это означает, что текущий узел находится в конце списка.

Обратите внимание на определение указателя head сразу же после объявления структуры Link (вместо отдельного определения Link* head). Этот синтаксис, позаимствованный из C, подчеркивает важность символа точки с запятой (;) после объявления структуры; этот символ обозначает конец списка определений переменных данного структурного типа (обычно список пуст).

Как и все структуры, встречавшиеся до настоящего момента, вложенная структура для инициализации имеет собственную функцию initialize(). Структура Stack содержит функции initialize() и cleanup(), а также функцию push(), получающую указатель на сохраняемые данные (предполагается, что память под эти данные была выделена из кучи), и функцию pop(), которая возвращает указатель на верхний блок данных и удаляет верхний элемент из стека (при удалении элемента функцией pop() необходимо уничтожить объект, на который ссылается указатель data). Функция peek() также возвращает указатель на данные верхнего элемента, но оставляет верхний элемент стека без изменений.

Определения функций выглядят так:

```
//: C04:Stack.cpp {O}
// Связанный список с вложением
#include "Stack.h"
```

```
#include "../require.h"
using namespace std;

void
Stack::Link::initialize(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

void Stack::initialize() { head = 0; }

void Stack::push(void* dat) {
  Link* newLink = new Link;
  newLink->initialize(dat, head);
  head = newLink;
}

void* Stack::peek() {
  require(head != 0, "Stack empty");
  return head->data;
}

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

void Stack::cleanup() {
  require(head == 0, "Stack not empty");
} ///:~
```

Особенно интересно первое определение, которое показывает, как определяются члены вложенных структур. Мы просто используем дополнительный уровень уточнения видимости с указанием имени внешней структуры. Функция Stack::Link::initialize() получает аргументы и присваивает их переменным вложенной структуры.

Функция Stack::initialize() присваивает переменной head нулевое значение — в исходном состоянии список пуст.

Функция Stack::push() получает указатель на переменную, которая заносится в стек. Сначала оператор new выделяет память для структуры Link, которая должна быть занесена в верхнюю позицию стека. Указателю next присваивается текущее значение head, после чего переменной head присваивается новый указатель на Link. Тем самым Link фактически заносится в начало списка.

Функция Stack::pop() получает указатель data из текущей вершины стека, смещает указатель head вниз и удаляет старую вершину стека, после чего возвращает сохраненный указатель. Когда функция pop() удаляет последний элемент, переменной head присваивается ноль (признак пустого стека).

Функция Stack::cleanup() в действительности не выполняет никаких завершающих действий. Вместо этого она помогает твердо соблюдать правило: прикладной программист, использующий объект Stack, отвечает за извлечение всех элементов

из стека и их удаление. Функция require() выводит сообщение об ошибке, если стек не пуст.

Почему бы в деструкторе Stack не удалить все объекты, не извлеченные вызовами pop()? Дело в том, что Stack содержит указатели void*, а как показано в главе 13, вызов delete для void* не обеспечивает правильного удаления. Впрочем, вопрос «кто отвечает за освобождение памяти?» на самом деле *еще* сложнее (вы подробнее узнаете об этом в следующих главах).

Пример программы для тестирования объекта Stack:

```
//: C04:StackTest.cpp
//{L} Stack
//{T} StackTest.cpp
// Тестирование связанного списка с вложенной структурой
#include "Stack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Аргумент - имя файла
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  textlines.initialize();
  string line;
  // Чтение файла и сохранение строк в стеке:
  while(getline(in, line))
    textlines.push(new string(line));
  // Извлечение строк из стека и вывод:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
  textlines.cleanup();
} ///:~
```

Этот пример похож на приведенный выше, но он заносит строки из файла (в виде указателей на string) в стек и извлекает их, в результате чего файл выводится в обратном порядке. Функция pop() возвращает указатель void*, который перед использованием необходимо преобразовать обратно в string*. Вывод строки осуществляется разыменованием этого указателя.

По мере заполнения textlines содержимое line «клонируется» для каждого вызова push() вызовом new string(line). Выражение new возвращает указатель на новый объект string, в который была скопирована информация из line. Если бы мы просто передали адрес line функции push(), то стек был бы заполнен одинаковыми адресами, указывающими на line. Процесс «клонирования» более подробно описан в следующих главах книги.

Имя файла передается в командной строке. Количество аргументов командной строки проверяется при помощи функции requireArgs() из заголовочного файла require.h. Эта функция сравнивает желаемое количество аргументов с argc и, если значения различаются, выводит соответствующее сообщение об ошибке и завершает программу.

Глобальная видимость

Оператор :: помогает выходить из ситуаций, в которых имя, выбранное компилятором по умолчанию (то есть «ближайшее» имя), вам не подходит. Предположим, в программе определена структура с локальным идентификатором a, но вы хотите из функции этой структуры сослаться на глобальный идентификатор a. По умолчанию компилятор выберет локальную переменную, поэтому необходимо специально указать ему, какая переменная вам нужна. Чтобы сослаться на глобальное имя, введите оператор :: без указания префикса. В следующем примере используются ссылки на глобальные переменную и функцию:

```
//: C04:Scoperes.cpp
// Глобальная видимость
int a;
void f() {}

struct S {
  int a;
  void f();
};

void S::f() {
  ::f();  // Иначе произойдет рекурсивный вызов!
  ::a++;  // Глобальная переменная a
  a--;    // Переменная a из области видимости структуры
}
int main() { S s; f(); } ///:~
```

Без уточнения видимости в S::f() компилятор по умолчанию выберет версии f() и a, принадлежащие структуре.

Итоги

В этой главе была представлена «фирменная» особенность C++: возможность включения функций в структуры. Новый тип структуры называется *абстрактным типом данных*, а структурные переменные называются *объектами*, или *экземплярами*, этого типа. Вызов функции, принадлежащей объекту, называется *отправкой сообщения* объекту. Суть объектно-ориентированного программирования заключается в отправке сообщений объектам.

Хотя группировка данных и функций существенно улучшает организацию кода и упрощает работу с библиотеками (поскольку скрытие имен предотвращает возможные конфликты), вам еще нужно много узнать, прежде чем ваши программы на C++ станут более надежными и безопасными. В следующей главе будет показано, как защитить некоторые члены структуры, чтобы никто другой не мог напрямую работать с ними. Механизм управления доступом проводит четкую границу между данными, которые могут изменяться пользователем структуры, и данными, доступными только разработчик.

Упражнения

1. Функция puts() стандартной библиотеки C выводит на консоль символьный массив (то есть используется в виде puts("hello")). Напишите програм-

му С, которая задействует функцию puts(), но не включает файл <stdio.h> или другие объявления функции. Откомпилируйте программу компилятором С (у некоторых компиляторов С++ существует специальный флаг командной строки для переключения в режим компиляции С). Затем откомпилируйте программу компилятором С++ и проанализируйте различия.

2. Создайте объявление структуры с одной функцией-членом, затем создайте определение для этой функции. Создайте объект нового типа данных и вызовите функцию.
3. Измените решение упражнения 2 так, чтобы структура объявлялась в «защищенном» заголовочном файле, при этом определение и функция main() должны находиться в разных срр-файлах.
4. Создайте структуру, содержащую одну переменную типа int, и две глобальные функции, каждая из которых получает указатель на структуру. Первая функция вызывается со вторым аргументом int и присваивает переменной int структуры значение своего аргумента, а вторая функция выводит значение структурной переменной int. Протестируйте обе функции.
5. Повторите упражнение 4, но переопределите функции так, чтобы они были членами структуры. Повторите тестирование.
6. Создайте класс, который бы при обращениях к своим переменным и вызове функций-членов использовал указатель this (избыточный), ссылающийся на адрес текущего объекта.
7. Создайте структуру Stash для хранения данных типа double. Заполните ее 25 значениями double и выведите их на консоль.
8. Повторите упражнение 7 для структуры Stack.
9. Создайте файл с функцией f(), которая получает аргумент int и выводит его на консоль при помощи функции printf() из файла <stdio.h>. Команда вывода имеет вид printf("%d\n",i), где i — выводимое значение. Создайте отдельный файл с функцией main(), в котором функция f() объявлялась бы с аргументом типа float. Вызовите f() из main(). Попробуйте откомпилировать программу компилятором С++ и посмотрите, к чему это приведет. Затем откомпилируйте программу компилятором С, скомпонуйте ее и посмотрите, что произойдет при запуске. Объясните, в чем дело.
10. Узнайте, как получить ассемблерный код в ваших компиляторах С и С++. Напишите функцию на языке С и структуру с одной функцией-членом на С++. Сгенерируйте ассемблерный код и найдите имена, сгенерированные для обеих функций. Посмотрите, какие изменения внесены компилятором.
11. Напишите программу с условной компиляцией фрагмента main(). Если некоторое препроцессорное имя определено, программа должна выводить одно сообщение, а если не определено — другое. Откомпилируйте программу и поэкспериментируйте с директивой #define. Затем выясните, как в вашем компиляторе организована передача препроцессорных определений в командной строке, и поэкспериментируйте с ней.

12. Напишите программу, в которой используется директива assert() с нулевым аргументом (то есть всегда ложным условием). Посмотрите, что происходит при запуске. Теперь откомпилируйте программу с директивой #define NDEBUG, запустите снова и проанализируйте различия.
13. Создайте абстрактный тип данных Video для представления кассеты в пункте видеопроката. Попробуйте определить все данные и операции, которые могут понадобиться для типа Video в системе учета видеокассет. Включите функцию print() для вывода информации о Video.
14. Создайте объект Stack для хранения объектов Video из упражнения 13. Определите несколько объектов Video, сохраните их в стеке и выведите при помощи функции Video::print().
15. Напишите программу, которая бы выводила размеры всех встроенных типов данных на вашем компьютере с использованием оператора sizeof.
16. Измените структуру Stash, чтобы данные хранились в объекте типа vector<char>.
17. С использованием оператора new выполните динамическое выделение памяти для следующих типов: int, long, массив из 100 символов (char), массив из 100 чисел float. Выведите адреса созданных объектов и освободите память оператором delete.
18. Напишите функцию с аргументом типа char*. Функция должна динамически (оператором new) выделять память для массива char, размер которого соответствует размеру переданного символьного массива. Используя механизм индексирования, скопируйте символы из аргумента в динамически созданный массив (не забудьте о завершающем нулевом символе) и верните указатель на копию. Протестируйте функцию в main(), передав ей статический символьный массив, получив результат и снова передав его функции. Выведите обе строки и оба указателя, чтобы убедиться, что они относятся к разным областям памяти. Освободите динамическую память при помощи оператора delete.
19. Напишите программу с объявлением структуры внутри другой структуры (*вложенные структуры*). Объявите переменные в обеих структурах, затем объявите и определите в них функции. Напишите функцию main() для тестирования новых типов.
20. Сколько памяти занимает структура? Напишите фрагмент программы, в котором бы выводились размеры разных структур. Создайте структуры, содержащие только переменные, а также структуры, содержащие переменные и функции. Создайте пустую структуру вообще без членов. Выведите размеры этих структур. Объясните результат для случая пустой структуры.
21. Как было показано в этой главе, C++ автоматически создает для структур тип, что эквивалентно применению оператора typedef. То же самое делается для перечисляемых типов и объединений. Напишите небольшую программу для демонстрации этого факта.
22. Создайте структуру Stack для хранения структур Stash. Каждая структура Stash должна содержать пять строк из входного файла. Экземпляры Stash

следует создавать оператором new. Загрузите файл в Stack, затем снова выведите его в исходной форме, извлекая данные из Stack.

23. Измените упражнение 22 и создайте структуру, которая бы инкапсулировала структуру Stack для хранения Stash. Операции добавления и получения строк должны выполняться только через функции этой структуры.
24. Создайте структуру с двумя переменными: int и указателем на другой экземпляр этой же структуры. Напишите функцию, которая получает адрес одной из структур и значение типа int (длину списка, который требуется построить). Функция строит цепочку структур (*связанный список*), начиная с первого аргумента (начала списка), при этом каждая структура должна содержать указатель на следующую структуру в списке. Новые экземпляры структур создаются оператором new, а в переменной int хранится счетчик (номер создаваемого объекта). В последней структуре списка присвойте указателю нулевое значение (признак конца списка). Напишите вторую функцию, которая получает начало списка, перебирает его до конца и выводит для каждого элемента значение указателя и переменную int.
25. Повторите упражнение 24, но вместо использования «свободных» структур и функций поместите их внутрь структуры.

Скрытие реализации

Типичная библиотека С содержит структуру и функции, выполняющие операции с этой структурой. До настоящего момента вы видели, как C++ берет *концептуально* связанные функции и *буквально* связывает их со структурой, для чего объявления функций перемещаются в область видимости структуры. При этом изменяется способ вызова функций структуры, пропадает необходимость в передаче адреса структуры в первом аргументе, а в программе появляется новое имя типа (чтобы вам не приходилось создавать для struct псевдоним оператором typedef).

Конечно, все эти возможности удобны — они улучшают организацию программы, упрощают ее написание и чтение. Однако существуют и другие важные факторы, упрощающие работу с библиотеками в C++, особенно в области защиты данных и контроля доступа. Настоящая глава посвящена ограничению доступа в структуры.

Установка ограничений

В любых отношениях важно установить границы, которые должны соблюдаться всеми участниками этих отношений. При создании библиотеки вы устанавливаете отношения с *прикладным программистом*, который использует вашу библиотеку для построения приложения или другой библиотеки.

В структурах С, как и во многом другом, что относится к С, никаких ограничений не существует. Прикладной программист может делать со структурой все, что захочет, и вы не сможете заставить его соблюдать какие-либо правила. Например, хотя в предыдущей главе неоднократно подчеркивалась важность функций initialize() и cleanup(), прикладной программист может попросту забыть о вызове этих функций (в следующей главе будет показано более надежное решение). И хотя, с вашей точки зрения, прикладной программист не должен напрямую изменять значения переменных в структурах, запретить такие изменения в языке С невозможно. Все данные полностью открыты для всех желающих.

Необходимость ограничения доступа к членам структур объясняется двумя причинами. Во-первых, нужно оградить от прикладных программистов то, что им трогать не положено, — инструменты, необходимые для внутренней работы типа данных, но не входящие в его интерфейс, используемый клиентами для решения своих задач. На самом деле такие ограничения только помогают прикладным программистам разобраться, что действительно важно для их работы, а от чего лучше держаться подальше.

Во-вторых, ограничение доступа позволяет разработчику библиотеки изменять внутренние механизмы работы структуры, не беспокоясь о том, как это отразится на прикладных программистах. Так, в примере со структурой Stack из предыдущей главы можно для скорости перейти на выделение памяти большими блоками (вместо выделения памяти при каждом добавлении элемента). Четкое разделение интерфейса и реализации позволяет решить эту задачу, после чего прикладному программисту останется только заново скомпоновать программу.

Управление доступом в С++

В С++ появились три новых ключевых слова для ограничения доступа в структурах: public, private и protected. Эти *спецификаторы доступа* используются в объявлениях структур и определяют уровень доступа для всех объявлений, следующих за ними. После любого спецификатора доступа должно стоять двоеточие.

Спецификатор public означает, что все дальнейшие объявления членов делают их *открытыми*, то есть доступными для всех желающих. В этом отношении они не отличаются от членов структур в языке С. Например, следующие объявления структур эквивалентны:

```
//: C05:Public.cpp
// Открытый доступ, как в структурах C

struct A {
  int i;
  char j;
  float f;
  void func();
};

void A::func() {}

struct B {
public:
  int i;
  char j;
  float f;
  void func();
};

void B::func() {}

int main() {
  A a; B b;
  a.i = b.i = 1;
  a.j = b.j = 'c';
```

```
  a.f = b.f = 3.14159;
  a.func();
  b.func();
} ///:~
```

С другой стороны, ключевое слово private означает, что объявляемые члены являются *закрытыми*, то есть обращения к ним возможны только из функций-членов этого типа (иначе говоря, закрытые члены недоступны для всех, кроме разработчика структуры). Фактически оно разделяет «зоны ответственности» создателя структуры и прикладного программиста. При попытке обратиться извне к закрытому члену структуры происходит ошибка компиляции. Например, в структуре B из предыдущего примера некоторые переменные можно скрыть от постороннего доступа:

```
//: C05:Private.cpp
// Ограничение доступа

struct B {
private:
  char j;
  float f;
public:
  int i;
  void func();
};

void B::func() {
  i = 0;
  j = '0';
  f = 0.0;
};

int main() {
  B b;
  b.i = 1;    // Можно, открытая переменная
//! b.j = '1'; // Нельзя, закрытая переменная
//! b.f = 1.0; // Нельзя, закрытая переменная
} ///:~
```

Хотя функция func() может обращаться ко всем членам B (потому что функция func() также является членом структуры B, что автоматически дает ей полные права доступа), обычные глобальные функции, в том числе и main(), этого делать не могут. Естественно, доступ из функций других структур тоже ограничен. Доступ к закрытым членам предоставляется только функциям, перечисленным при объявлении структуры.

Спецификаторы доступа не обязаны следовать в каком-либо обязательном порядке и могут присутствовать в объявлении более одного раза. Действие спецификатора распространяется на все дальнейшие объявления до следующего спецификатора.

Защищенные члены

Последний спецификатор доступа, protected, работает, как private, за одним исключением, о котором пока говорить рано: доступ к защищенным (protected) членам предоставляется «наследующим» структурам, для которых закрытые члены недо-

ступны. Сказанное станет более понятным в главе 14 при описании наследования. А пока считайте, что protected работает так же, как private.

Друзья

А что, если вдруг потребуется специально предоставить доступ к закрытым членам из функции, не принадлежащей структуре? Для этого следует *внутри* объявления структуры объявить эту функцию *другом* (friend) структуры. Очень важно, чтобы объявление friend находилось внутри объявления структуры, чтобы компилятор в процессе обработки объявления располагал полной информацией о размере и поведении этого типа данных, включая ответ на очень важный вопрос: «Кто получает доступ к моей закрытой реализации?»

Структура сама управляет тем, кому предоставляется доступ к ее членам. Нет ни малейшей возможности получить доступ извне, если об этом ничего не сказано в самой структуре. Нельзя объявить новую структуру, заявив: «Привет, я являюсь другом структуры Bob», и рассчитывать на получение доступа к закрытым и защищенным членам структуры Bob.

Дружественными могут объявляться глобальные функции, функции-члены других структур и даже целые структуры. Пример:

```
//: C05:Friend.cpp
// Ключевое слово friend предоставляет особый уровень доступа

// Объявление (неполная спецификация типа):
struct X;

struct Y {
  void f(X*);
};

struct X { // Определение
private:
  int i;
public:
  void initialize();
  friend void g(X*, int); // Глобальная дружественная функция
  friend void Y::f(X*);  // Дружественная функция другой структуры
  friend struct Z; // Дружественной является вся структура
  friend void h();
};

void X::initialize() {
  i = 0;
}

void g(X* x, int i) {
  x->i = i;
}

void Y::f(X* x) {
  x->i = 47;
}

struct Z {
```

```
private:
  int j;
public:
  void initialize();
  void g(X* x);
};

void Z::initialize() {
  j = 99;
}

void Z::g(X* x) {
  x->i += j;
}

void h() {
  X x;
  x.i = 100; // Прямое обращение к данным
}

int main() {
  X x;
  Z z;
  z.g(&x);
} ///:~
```

Структура Y содержит функцию f(), которая модифицирует объект типа X. Возникает запутанная ситуация: компилятор C++ требует, чтобы все объявлялось перед первой ссылкой, поэтому структура Y должна быть объявлена до того, как ее функция Y::f(x*) будет объявлена дружественной в структуре X. Но чтобы объявить Y::f(X*), необходимо сначала объявить структуру X!

Однако у этой проблемы есть решение. Обратите внимание: функция Y::f(X*) получает *адрес* объекта X. Это очень важно — компилятор всегда умеет правильно передавать адреса, поскольку их размер не зависит от передаваемого объекта, даже если на данный момент полная информация о размере типа отсутствует. Но если бы передавался весь объект, то перед объявлением функций вида Y::g(X) компилятору потребовалось бы полное определение структуры X для вычисления размера и способа передачи.

Благодаря передаче адреса X мы можем использовать ***неполную спецификацию типа*** X перед объявлением Y::f(X*). Задача решается объявлением

```
struct X;
```

Такое объявление просто сообщает компилятору о существовании структуры с указанным именем, что позволяет ссылаться на нее во всех ситуациях, не требующих дополнительной информации об этой структуре.

Теперь в структуре X функция Y::f(X*) объявляется дружественной безо всяких проблем. Если попытаться объявить ее до того, как компилятор получит полную спецификацию Y, произойдет ошибка компиляции. Это дополнительная мера безопасности, которая обеспечивает логическое согласование объявлений и предотвращает ошибки.

Обратите внимание на два других объявления дружественных функций. Первое объявляет дружественной обычную глобальную функцию g(). Но ведь функция g() не объявлялась ранее в глобальной области видимости! Оказывается, клю-

чевое слово friend позволяет одновременно объявить функцию *и* предоставить ей статус дружественной. Принцип распространяется на целые структуры. Следующее объявление содержит неполную спецификацию типа Z и объявляет всю структуру как дружественную:

```
friend struct Z;
```

Вложенные друзья

Структура, объявленная как вложенная, не получает автоматического доступа к закрытым членам внешней структуры. Для таких случаев существует конкретная процедура: сначала объявляется (без определения) вложенная структура, затем она объявляется дружественной и только после этого определяется. Определение структуры должно быть отделено от объявления friend, в противном случае компилятор не воспримет ее как вложенную. Пример:

```
//: C05:NestFriend.cpp
// Вложенные дружественные структуры
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;

struct Holder {
private:
  int a[sz];
public:
  void initialize();
  struct Pointer;
  friend struct Pointer;
  struct Pointer {
  private:
    Holder* h;
    int* p;
  public:
    void initialize(Holder* h);
    // Перемещение по массиву:
    void next();
    void previous();
    void top();
    void end();
    // Обращение к данным:
    int read();
    void set(int i);
  };
};

void Holder::initialize() {
  memset(a, 0, sz * sizeof(int));
}

void Holder::Pointer::initialize(Holder* rv) {
  h = rv;
  p = rv->a;
}

void Holder::Pointer::next() {
  if(p < &(h->a[sz - 1])) p++;
```

```
}

void Holder::Pointer::previous() {
  if(p > &(h->a[0])) p--;
}

void Holder::Pointer::top() {
  p = &(h->a[0]);
}

void Holder::Pointer::end() {
  p = &(h->a[sz - 1]);
}

int Holder::Pointer::read() {
  return *p;
}

void Holder::Pointer::set(int i) {
  *p = i;
}

int main() {
  Holder h;
  Holder::Pointer hp, hp2;
  int i;

  h.initialize();
  hp.initialize(&h);
  hp2.initialize(&h);
  for(i = 0; i < sz; i++) {
    hp.set(i);
    hp.next();
  }
  hp.top();
  hp2.end();
  for(i = 0; i < sz; i++) {
    cout << "hp = " << hp.read()
         << ", hp2 = " << hp2.read() << endl;
    hp.next();
    hp2.previous();
  }
} ///:~
```

После объявления структуры Pointer мы предоставляем ей доступ к закрытым членам Holder объявлением:

```
friend Pointer;
```

Структура Holder содержит массив чисел int и объект Pointer для работы с массивом. Поскольку структура Pointer тесно связана с Holder, есть смысл объявить ее вложенной по отношению к Holder. Но структура Pointer относится к отдельному типу, и это позволяет создать в функции main() несколько экземпляров этого типа и использовать их для работы с разными частями массива. Pointer является структурой, а не обычным указателем C, а выполнение операций с Pointer через интерфейс гарантирует, что указатель всегда будет ссылаться на Holder.

В этой программе используется стандартная библиотечная функция C memset() (из библиотеки <cstring>). Функция заполняет n байтов памяти начиная с некото-

рого адреса заданным значением. Конечно, с таким же успехом можно было перебрать байты в цикле, но функция memset() уже существует, она хорошо протестирована (что снижает вероятность случайных ошибок) и, вероятно, более эффективна по сравнению с самостоятельно запрограммированной версией.

Покушение на «чистоту» языка

Определение класса создает «контрольный след», по которому можно узнать, какие функции обладают правом на модификацию закрытых частей класса. Если функция объявлена дружественной, это означает, что она не принадлежит классу, но вы все равно хотите разрешить ей модификацию закрытых данных. Поэтому функция включается в определение класса, чтобы все знали о ее привилегированном статусе.

C++ считается гибридным, а не чистым объектно-ориентированным языком, в связи с чем и появилось ключевое слово friend. Оно несколько снижает «чистоту» языка, но C++ проектировался для решения практических проблем, а не для воплощения абстрактных идеалов.

Строение объекта

В главе 4 было показано, что структура, написанная для компилятора C и позднее обработанная компилятором C++, остается без изменений. Это относилось в первую очередь к строению объекта, то есть к размещению отдельных переменных в памяти, выделенной для объекта. Если бы компилятор C++ изменял строение структур C, это привело бы к нарушению работы всех программ C, которые использовали информацию о внутреннем представлении переменных в структурах.

Когда ограничение доступа относится только к специфике C++, ситуация меняется. Внутри некоторого «блока доступа» (группы объявлений, относящихся к одному спецификатору доступа) переменные заведомо занимают смежные участки памяти, как в C. Но нельзя гарантировать, что блоки доступа будут следовать в объекте в порядке их объявления. Хотя компилятор *обычно* размещает блоки в том порядке, в котором они следуют в программе, это не обязательно, потому что в конкретной архитектуре компилятора и/или рабочей среды может быть предусмотрена специальная поддержка спецификаторов private и protected, для которой эти блоки должны размещаться в особых областях памяти. Спецификация языка не должна препятствовать подобным возможностям.

Спецификаторы доступа являются частью структуры и не влияют на объекты, созданные на базе этой структуры. Вся информация об ограничении доступа исчезает перед запуском программы; обычно это происходит на стадии компиляции. В работающей программе объект является «областью памяти» и ничем более. Впрочем, при желании вы можете нарушить все правила и обращаться к памяти напрямую, как в C. Язык C++ не рассчитан на то, чтобы оградить вас от подобных неразумных поступков, — он всего лишь предоставляет гораздо более простую и удобную альтернативу.

В общем случае при написании программы не рекомендуется использовать какие-либо возможности, зависящие от реализации. А если это неизбежно, инкапсулируйте их в структуре, чтобы все изменения при переносе на другую платформу были сосредоточены в одном месте.

Класс

Управление доступом часто называется *скрытием реализации*. Включение функций в структуры (инкапсуляция[1]) создает тип данных, обладающий атрибутами и поведением, а механизм управления доступом позволяет ограничить доступ к этому типу. Основных причин такого ограничения две. Прежде всего, путем управления доступом разработчик указывает, что в структуре может или не может использоваться прикладными программистами. Он строит внутренние механизмы структуры и не беспокоится о том, что прикладной программист примет эти механизмы за часть интерфейса для работы со структурой.

Отсюда непосредственно следует вторая причина — отделение интерфейса от реализации. Если структура используется в программах, а прикладной программист работает с ней только посредством отправки сообщений через открытый интерфейс, то разработчик может по своему усмотрению изменять все закрытые части — это не потребует изменения существующих программ.

Сочетание инкапсуляции и управления доступом создает нечто большее, чем просто структуру С. В мире объектно-ориентированного программирования структуры описывают категории объектов подобно тому, как мы объединяем в категории однородные объекты (например, птиц или рыб). Все объекты, принадлежащие к одному классу, обладают некоторыми атрибутами и поведением. Так объявление структуры превращается в описание строения и поведения всех объектов данного типа.

В первом объектно-ориентированном языке Simula-67 для определения нового типа данных было использовано ключевое слово class. Видимо, этот факт вдохновил Страуструпа и заставил его выбрать то же ключевое слово для С++, чтобы подчеркнуть важнейшую особенность языка: новые типы данных представляют собой нечто большее, чем просто структуры С с функциями. Бесспорно, это полностью оправдывает введение нового ключевого слова.

Тем не менее ключевое слово class в С++ почти избыточно. Оно практически полностью идентично ключевому слову struct, существует только одно отличие: по умолчанию члены класса являются закрытыми, а члены структуры — открытыми. Следующие два объявления эквивалентны:

```
//: C05:Class.cpp
// Сходство между классами и структурами

struct A {
private:
  int i, j, k;
public:
  int f();
  void g();
};

int A::f() {
  return i + j + k;
}

void A::g() {
  i = j = k = 0;
}
```

[1] Как упоминалось выше, под термином «инкапсуляция» иногда понимают управление доступом.

```
// Следующий класс дает идентичный результат:

class B {
  int i, j, k;
public:
  int f();
  void g();
};

int B::f() {
  return i + j + k;
}

void B::g() {
  i = j = k = 0;
}

int main() {
  A a;
  B b;
  a.f(); a.g();
  b.f(); b.g();
} ///:~
```

Классы являют собой основополагающую концепцию ООП в C++. Переход на классы был настолько важен, что, по мнению автора, Страуструп охотно исключил бы ключевое слово struct из языка, но требование совместимости с C не позволило это сделать.

Многие программисты предпочитают создавать классы в стиле, более близком к структурам, — они переопределяют характерную для классов «закрытость по умолчанию», начиная с определения открытых элементов:

```
class X {
public:
  void interface_function();
private:
  void private_function();
  int internal_representation;
};
```

Такой подход по-своему логичен: читатель программы начинает с открытых членов, которые интересуют его в первую очередь, и не обращает внимания на объявления, следующие после спецификатора private. В самом деле, объявление всех остальных членов в классе необходимо лишь для того, чтобы компилятор знал размеры объектов и мог правильно выделить для них память, а также обеспечить логическую согласованность разных частей программы.

Однако в примерах этой книги классы будут начинаться с закрытых членов:

```
class X {
  void private_function();
  int internal_representation;
public:
  void interface_function();
};
```

Некоторые программисты даже изобретают собственные системы обозначений для имен закрытых членов:

```
class Y {
public:
  void f();
private:
  int mX;  //  Специальное имя
};
```

Переменная mX уже скрыта в области видимости Y, поэтому префикс m оказывается лишним. Тем не менее в проектах с большим количеством глобальных переменных (что вообще-то нежелательно, но иногда неизбежно) это может быть удобно: внутри определения функции класса сразу видно, какие данные являются глобальными, а какие принадлежат классу.

Версия Stash с ограничением доступа

Давайте возьмем примеры из главы 4 и изменим их так, чтобы в них использовались классы и механизмы управления доступом. Обратите внимание: после изменения часть интерфейса, предназначенная для прикладных программистов, оказывается четко выделенной, поэтому им не удастся случайно изменить часть класса, для них не предназначенную.

```
//: C05:Stash.h
// Версия Stash с ограничением доступа
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;      // Размер каждого элемента
  int quantity;  // Количество элементов
  int next;      // Следующий пустой элемент
  // Динамически выделяемый байтовый массив:
  unsigned char* storage;
  void inflate(int increase);
public:
  void initialize(int size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH_H ///:~
```

Функция inflate() стала закрытой (private), потому что она используется только функцией add() и, следовательно, принадлежит к базовой реализации, а не к внешнему интерфейсу класса. Это означает, что когда-нибудь в будущем вы сможете изменить базовую реализацию и перейти на другую систему управления памятью.

Если не считать имени включаемого файла, в этом примере не изменилось ничего, кроме приведенного выше заголовка. Файл реализации и тестовая программа остались прежними.

Версия Stack с ограничением доступа

Во втором примере структура Stack будет преобразована в класс. Вложенная структура данных объявлена закрытой, и это к лучшему: прикладной программист не увидит внутреннего представления Stack и не будет зависеть от него:

```
//: C05:Stack2.h
// Вложенные структуры в связанном списке
#ifndef STACK2_H
#define STACK2_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
public:
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK2_H ///:~
```

Как и в предыдущем примере, файл реализации не изменился, поэтому он здесь не приводится. Тестовая программа тоже осталась прежней. Изменилась только защищенность интерфейса класса. Настоящая ценность механизма управления доступом заключается в том, что он предотвращает нежелательные нарушения границ в ходе разработки программ. В сущности, уровень защиты членов класса известен только компилятору. Информация об уровне доступа не входит в «украшенное» имя члена класса, которое передается компилятору. Вся проверка системы защиты выполняется компилятором, а на стадии выполнения ограничения доступа исчезают.

Обратите внимание: интерфейс, доступный для прикладного программиста, теперь больше напоминает нормальный стек. Он реализован в виде связанного списка, но базовую реализацию можно в любой момент изменить, причем это не повлияет ни на те части класса, с которыми работает прикладной программист, ни (что еще важнее) на одну из строк клиентского кода.

Классы-манипуляторы

Управление доступом в C++ позволяет отделить интерфейс от реализации, но скрытие реализации выполняется лишь частично. Компилятор все равно должен видеть объявления всех составляющих объекта, чтобы правильно создать объект и оперировать им. В принципе, можно представить язык программирования, в котором обязательно объявляется только открытый интерфейс объекта, а закрытая реализация может быть скрыта, но C++ стремится по возможности выполнять статическую проверку типов (на стадии компиляции). В результате разработчик узнает об ошибках на возможно более ранней стадии, а программы становятся более эффективными. Тем не менее включение закрытой реализации приводит к двум последствиям: во-первых, реализация остается видимой даже в том случае, если к ней нельзя легко получить доступ; во-вторых, это может стать причиной лишних перекомпиляций.

Скрытие реализации

В некоторых проектах тот факт, что реализация остается видимой для прикладного программиста, оказывается неприемлемым. Например, заголовочный файл

библиотеки может содержать информацию, которую фирма не желает открывать своим конкурентам. Или вы работаете над системой, относящейся к сфере безопасности (скажем, алгоритмом шифрования), и не хотите, чтобы сведения из заголовочного файла использовались для раскрытия шифра. Или библиотека будет задействована в «агрессивной среде», в которой программисты все равно попытаются напрямую работать с закрытыми компонентами с применением указателей и операций приведения типа. Во всех перечисленных ситуациях реально используемая структура должна компилироваться в файле реализации без раскрытия информации через заголовочный файл.

Лишние перекомпиляции

Подсистема управления проектами в вашей среде программирования требует перекомпилировать файл в случае модификации этого файла *или* других файлов, от которых он зависит, — в том числе и включаемых заголовочных файлов. Это означает, что любые изменения класса, как в открытом интерфейсе, так и в объявлениях закрытых членов, приведут к принудительной перекомпиляции всех файлов, в которые включен этот заголовочный файл. Данная ситуация также часто называется проблемой *неустойчивости базовых классов*. В больших проектах на ранней стадии разработки базовые реализации изменяются довольно часто, и лишние перекомпиляции могут стать источником серьезных проблем. В очень больших проектах затраты времени на компиляцию основательно тормозят разработку.

Из этой ситуации поможет выйти методика, основанная на использовании так называемых *классов-манипуляторов*, или *«чеширских котов»*[1]. При этом все, что относится к реализации, исчезает, и остается только указатель («улыбка чеширского кота»). Указатель ссылается на структуру, определяемую в файле реализации вместе со всеми определениями функций. Таким образом, при условии неизменности интерфейса заголовочный файл тоже остается неизменным. Программист может свободно изменять базовую реализацию, и ему останется лишь перекомпилировать и перекомпоновать файл реализации.

Следующий простой пример демонстрирует эту методику. Заголовочный файл содержит только описание открытого интерфейса и единственный указатель на класс с неполной спецификацией:

```
//: C05:Handle.h
// Классы-манипуляторы
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
  struct Cheshire; // Только объявление класса
  Cheshire* smile;
public:
  void initialize();
  void cleanup();
  int read();
```

[1] Термин обязан своим происхождением Джону Кэролану (John Carolan), одному из первых экспертов C++... ну и, конечно, Льюису Кэрроллу (Lewis Carroll), автору незабываемой «Алисы». Прием можно рассматривать как частный случай архитектурного эталона «мост», о котором речь пойдет во втором томе.

```
  void change(int);
};
#endif // HANDLE_H ///:~
```

Вот и все, что положено видеть прикладному программисту. Следующая строка содержит *неполную спецификацию типа*, или *объявление класса* (*определение класса* включает его тело):

```
struct Cheshire;
```

Эта строка говорит компилятору, что идентификатор Cheshire является именем структуры, но не сообщает никаких подробностей об этой структуре. Такая информация позволяет создать только указатель на структуру; создать сам объект невозможно до тех пор, пока компилятор не встретит определение структуры. Но, согласно этой методике, определение структуры скрывается в файле реализации:

```
//: C05:Handle.cpp {O}
// Реализация класса-манипулятора
#include "Handle.h"
#include "../require.h"

// Определение реализации Handle:
struct Handle::Cheshire {
  int i;
};

void Handle::initialize() {
  smile = new Cheshire;
  smile->i = 0;
}

void Handle::cleanup() {
  delete smile;
}

int Handle::read() {
  return smile->i;
}

void Handle::change(int x) {
  smile->i = x;
} ///:~
```

Структура Cheshire является вложенной, поэтому при ее определении должна быть явно указана область видимости:

```
struct Handle::Cheshire {
```

Функция Handle::initialize() выделяет память для структуры Cheshire, а в структуре Handle::cleanup() эта память освобождается. Она заменяет все элементы данных, которые обычно размещаются в секции private. При компиляции Handle.cpp определение структуры скрывается в объектном файле, где его никто не видит. Если изменить элементы Cheshire, придется перекомпилировать только файл Handle.cpp, потому что заголовочный файл остается без изменений.

По правилам использования класс Handle ничем не отличается от любого другого класса: включите заголовок, создайте объекты и организуйте передачу сообщений.

```
//: C05:UseHandle.cpp
//{L} Handle
// Использование класса Handle
#include "Handle.h"

int main() {
  Handle u;
  u.initialize();
  u.read();
  u.change(1);
  u.cleanup();
} ///:~
```

Для прикладного программиста доступен только открытый интерфейс, поэтому до тех пор пока изменения происходят только в базовой реализации, приведенный выше файл не потребует перекомпиляции. Хотя скрытие реализации получается не идеальным, налицо бесспорное улучшение.

Итоги

Механизм управления доступом в C++ открывает перед создателем класса ряд полезных возможностей. Пользователь класса четко видит, что он может задействовать в своей работе, а на что можно не обращать внимания. Но существует другая, еще более важная сторона — благодаря управлению доступом прикладной программист перестает зависеть от базовой реализации класса. Создатель класса может изменить базовую реализацию, зная, что эти изменения никак не отразятся на работе прикладного программиста — эта часть класса для него недоступна.

Возможность изменения базовой реализации не только позволяет усовершенствовать архитектуру приложения, но и предотвращает фатальные последствия от ошибок. Даже при самом тщательном планировании и проектировании возможны ошибки. Если программист знает, что ошибки можно относительно безопасно исправить в будущем, он более склонен к экспериментам, быстрее учится и быстрее завершает работу над проектом.

В своих программах прикладной программист видит только открытый интерфейс класса, поэтому именно эту часть класса нужно в первую очередь «наладить» в процессе анализа и проектирования. Но даже в этом случае у вас остается определенная свобода для изменений. Если в первом варианте интерфейс оказался неудачным, *добавьте* в него новые функции, — главное, чтобы из интерфейса не исчезли те функции, которые уже используются прикладными программистами.

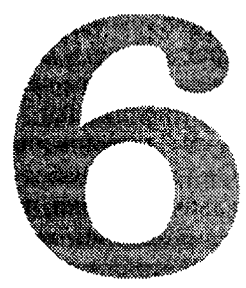
Упражнения

1. Создайте класс с открытыми (public), закрытыми (private) и защищенными (protected) переменными и функциями. Создайте объект класса и посмотрите, какие сообщения выводятся компилятором при попытках обращения ко всем категориям членов класса.

2. Создайте структуру Lib, содержащую три переменные типа string: a, b и c. В функции main() создайте объект Lib с именем x и присвойте значения x.a, x.b и x.c. Выведите присвоенные значения. Замените a, b и c массивом string s[3]. Убедитесь, что сделанное изменение нарушает работу функции main(). Создайте класс Libc, содержащий закрытые объекты string с именами a, b и c, а также функции seta(), geta(),setb(), getb(), setc() и getc() для присваивания/чтения значений этих переменных. Напишите функцию main(), как прежде, и замените закрытые объекты a, b и c закрытым массивом string s[3]. Убедитесь, что сделанное изменение *не нарушает* работу функции main().
3. Создайте класс и глобальную дружественную функцию, которая изменяет значения закрытых переменных класса.
4. Напишите два класса, в каждом из которых имеется функция, получающая указатель на объект другого класса. Создайте экземпляры обоих классов в main() и вызовите упомянутые функции в каждом классе.
5. Создайте три класса. Первый класс должен содержать закрытые данные, но предоставлять дружественный доступ всему второму классу и одной из функций третьего класса. Покажите в функции main(), что все три класса работают правильно.
6. Создайте класс Hen, определите в нем вложенный класс Nest. В классе Nest определите вложенный класс Egg. Каждый класс должен содержать функцию display(). В функции main() создайте экземпляр каждого класса и вызовите функции display() для всех классов.
7. Измените упражнение 6 так, чтобы классы Nest и Egg содержали закрытые данные. Объявите внешние классы дружественными по отношению к вложенным, чтобы предоставить им доступ к закрытым данным.
8. Создайте класс с переменными, распределенными по нескольким секциям public, private и protected. Добавьте функцию showMap() для вывода всех переменных и их адресов. По возможности откомпилируйте и протестируйте программу с разными компьютерами и/или компиляторами и/или операционными системами. Проверьте, существуют ли различия в формате представления объекта.
9. Скопируйте файлы реализации и тестовой программы для структуры Stash из главы 4. Откомпилируйте и протестируйте файл Stash.h из этой главы.
10. Поместите объекты класса Hen из упражнения 6 в структуру Stash. Прочитайте и выведите эти объекты (добавьте функцию Hen::print(), если это не было сделано ранее).
11. Скопируйте файлы реализации и тестовой программы для структуры Stack из главы 4. Откомпилируйте и протестируйте файл Stack2.h из этой главы.
12. Поместите объекты класса Hen из упражнения 6 в структуру Stack. Прочитайте и выведите эти объекты (добавьте функцию Hen::print(), если это не было сделано ранее).

13. Измените структуру Cheshire из программы Handle.cpp. Убедитесь в том, что подсистема управления проектами заново компилирует и компонует только этот файл, без перекомпиляции файла UseHandle.cpp.
14. Создайте класс StackOfInt (стек, содержащий значения int) с применением методики «чеширского кота»; низкоуровневая структура данных, непосредственно используемая для хранения элементов, должна быть скрыта в классе StackImp. Реализуйте две версии StackImp: в одной должен быть задействован массив int фиксированной длины, а в другой — vector<int>. Ограничьте максимальный размер стека, чтобы вам не пришлось беспокоиться о расширении массива в первой версии. Обратите внимание: изменение StackImp не обязательно приведет к изменению StackOfInt.h.

Инициализация и зачистка



В главе 4 разрозненные компоненты типичной библиотеки С были инкапсулированы в структуру. В результате появился абстрактный тип данных, в дальнейшем именуемый *классом*. Однако классы не только предоставляют единую точку входа для обращения к компонентам библиотеки, но и позволяют скрывать имена функций. В главе 5 был представлен механизм управления доступом (то есть скрытия реализации), при помощи которого разработчик класса может провести четкую границу между компонентами, доступными для прикладного программиста, и теми, с которыми ему работать не положено. Таким образом, разработчик класса распоряжается внутренними механизмами, связанными с типом данных, по своему усмотрению, а прикладной программист знает, с какими членами класса ему разрешено работать.

Сочетание инкапсуляции с управлением доступом значительно упрощает работу с библиотеками. Концепция «нового типа данных», обеспечиваемая ими, во многих отношениях превосходит встроенные типы данных С. Компилятор С++ реализует проверку нового типа, что повышает безопасность его использования.

Вообще в области безопасности компилятор С++ способен сделать гораздо больше, чем компилятор С. В этой и следующей главах будут описаны новые возможности С++, благодаря которым ошибки буквально «лезут на глаза» — иногда даже до компиляции программы, но чаще в форме предупреждений и сообщений от компилятора. По этой причине вы будете часто сталкиваться с непривычной ситуацией, когда компилируемая программа С++ правильно работает с первого запуска.

Среди аспектов безопасности стоит особо упомянуть об инициализации и зачистке (завершающих действиях с объектом). Очень многие ошибки С возникают тогда, когда программист забывает правильно инициализировать или деинициализировать переменную. Это особенно справедливо для библиотек С, когда прикладной программист не знает, как инициализировать структуру, или вообще не подозревает о том, что это нужно сделать (в некоторых библиотеках функция инициализации отсутствует, и прикладному программисту приходится заполнять

структуры вручную). Особенно много проблем возникает с зачисткой, поскольку программисты C склонны забывать о своих переменных, завершив работу с ними, из-за чего завершающие операции, необходимые для структур библиотеки, часто не выполняются.

В C++ концепции инициализации и зачистки играют чрезвычайно важную роль. Они делают работу с библиотеками более удобной и искореняют многие нетривиальные ошибки, происходящие из-за того, что прикладной программист забывает о выполнении этих операций.

В этой главе рассматриваются средства C++, гарантирующие правильное выполнение инициализации и зачистки.

Гарантированная инициализация с помощью конструктора

В рассмотренных классах Stash и Stack присутствует функция initialize(), которая, как напрямую следует из ее названия, должна вызываться перед первым использованием объекта. К сожалению, из этого следует, что прикладной программист должен помнить о необходимости инициализации. На практике прикладные программисты склонны забывать о подобных мелочах, стараясь поскорее применить вашу замечательную библиотеку для решения своих задач. Однако в C++ инициализация слишком важна, чтобы ее можно было доверить прикладным программистам. Чтобы обеспечить должную инициализацию каждого объекта, разработчик класса включает в него специальную функцию, которая называется *конструктором*. Если у класса имеется свой конструктор, компилятор автоматически вызывает его в точке создания объекта, до того как прикладные программисты успеют с ним что-нибудь сделать. Вызов конструктора совершенно не зависит от прикладного программиста и происходит автоматически в точке определения объекта.

Однако возникает проблема с выбором имени этой функции. Приходится учитывать два аспекта. Во-первых, любое фиксированное имя теоретически может конфликтовать с именами, присваиваемыми членам класса. Во-вторых, за вызов конструктора отвечает компилятор, поэтому он всегда должен знать, какую функцию следует вызвать. Предложенное Страуструпом решение выглядит предельно просто и логично: имя конструктора должно совпадать с именем класса.

Рассмотрим простой пример класса с конструктором:

```
class X {
  int i;
public:
  X();  // Конструктор
};
```

При определении объекта в следующем фрагменте происходит то же самое, что происходило бы при создании переменной a типа int, — для объекта выделяется память:

```
void f() {
  X a;
  // ...
}
```

Когда программа достигает точки выполнения, в которой определяется объект a, происходит автоматический вызов конструктора. Другими словами, компилятор незаметно вставляет вызов X::X() в точке определения объекта a. Конструктор, как и все остальные функции классов, получает в первом (секретном) аргументе this адрес того объекта, для которого он вызывается. Однако в случае конструктора указатель this ссылается на неинициализированный блок памяти, и задача конструктора как раз и заключается в правильной инициализации этой памяти.

Конструктор, как и любая другая функция, при вызове может получать аргументы с информацией о создании объекта, исходных значениях переменных класса и т. д. Аргументы конструктора позволяют обеспечить инициализацию всех компонентов объекта нужными значениями. Например, если в классе Tree определен конструктор с одним аргументом типа int, задающим высоту дерева, то создание объекта tree должно выглядеть примерно так:

```
Tree t(12); // Дерево высотой 12 метров
```

Если в классе нет других конструкторов, кроме Tree(int), то компилятор не позволит создать объект каким-то иным способом (возможности определения нескольких конструкторов и различные способы их вызова будут рассматриваться в следующей главе).

Собственно, вот и все, что можно сказать о конструкторе, — это обычная функция со специальным именем, автоматически вызываемая компилятором для каждого объекта в точке его создания. Однако, несмотря на простоту, конструкторы чрезвычайно ценны, поскольку они позволяют избавиться от целого класса ошибок и упрощают как написание, так и чтение программ. Например, в предыдущем фрагменте кода отсутствует явный вызов аналога функции initialize(), концептуально отделенный от определения объекта. В C++ концепции определения и инициализации тесно связаны и одно невозможно без другого.

Конструкторы и деструкторы обладают одной уникальной особенностью: они не имеют возвращаемого значения. В этом они принципиально отличаются от функций, возвращающих пустое значение (значение типа void), — такие функции не возвращают значения, но при желании можно заставить их что-нибудь вернуть. Конструкторы и деструкторы ничего не возвращают, и вы не в силах что-либо изменить. Появление и уход объекта из программы — такие же особые события, как рождение и смерть, поэтому компилятор всегда сам вызывает соответствующие функции, не доверяясь программисту. Если бы у этих функций имелось возвращаемое значение, было бы непонятно, что с ним делать компилятору. Другой вариант — возможность явного вызова конструкторов и деструкторов — сделал бы их вызов менее надежным.

Гарантированная зачистка с помощью деструктора

Программисты C часто сознают важность инициализации, но гораздо реже думают о зачистке. Допустим, что нужно сделать после освобождения объекта int? Ничего, просто забыть о нем. Но при работе с библиотеками «просто забыть» об объекте после завершения работы с ним может быть небезопасно. А что, если этот объект переключает устройство в другой режим работы, выводит данные на экран или

выделяет память из кучи? Если забыть о нем, то последствия существования объекта еще долго будут отражаться на работе системы. В C++ зачистка не уступает по важности инициализации, поэтому ее выполнение гарантируется при помощи деструктора.

Синтаксис деструктора в целом схож с синтаксисом конструктора: имя функции тоже определяется именем класса. Но чтобы деструктор отличался от конструктора, его имя начинается с префикса ~ (тильда). Кроме того, деструктор всегда вызывается без аргументов, потому что его поведение не нуждается в модификации. Пример объявления деструктора:

```
class Y {
public:
  ~Y();
};
```

Деструктор автоматически вызывается компилятором при выходе объекта из области видимости. Момент вызова конструктора всегда известен, это происходит в точке определения объекта, но единственным признаком вызова деструктора является закрывающая фигурная скобка того блока, в котором находится объект. И все же деструктор вызывается автоматически даже в том случае, если для выхода из блока использовалась команда goto (она сохранена в C++ для совместимости с языком C и иногда бывает удобной). Учтите, что деструкторы не вызываются при *нелокальных переходах*, реализуемых с использованием функций стандартной библиотеки C setjmp() и longjmp(). (Во всяком случае, так сказано в спецификации, даже если ваш компилятор реализует нелокальные переходы другим способом. Применение средств, не описанных в спецификации, нарушает переносимость программы.)

Пример использования конструкторов и деструкторов:

```
//: C06:Constructor1.cpp
// Конструкторы и деструкторы
#include <iostream>
using namespace std;

class Tree {
  int height;
public:
  Tree(int initialHeight);  // Конструктор
  ~Tree();  // Деструктор
  void grow(int years);
  void printsize();
};

Tree::Tree(int initialHeight) {
  height = initialHeight;
}

Tree::~Tree() {
  cout << "inside Tree destructor" << endl;
  printsize();
}

void Tree::grow(int years) {
  height += years;
}

void Tree::printsize() {
```

```
    cout << "Tree height is " << height << endl;
  }

  int main() {
    cout << "before opening brace" << endl;
    {
      Tree t(12);
      cout << "after Tree creation" << endl;
      t.printsize();
      t.grow(4);
      cout << "before closing brace" << endl;
    }
    cout << "after closing brace" << endl;
  } ///:-
```

Результат выполнения программы:

```
before opening brace
after Tree creation
Tree height is 12
before closing brace
inside Tree destructor
Tree height is 16
after closing brace
```

Как нетрудно убедиться, деструктор автоматически вызывается, когда в ходе выполнения программа «добирается» до закрывающей фигурной скобки.

Исключение блока определений

В языке С все переменные всегда должны определяться в начале блока после открывающей фигурной скобки. Это требование характерно для многих языков программирования и обычно считается «хорошим стилем программирования». На взгляд автора, такое обоснование выглядит сомнительно. Вряд ли программисту удобно возвращаться к началу блока каждый раз, когда в программе требуется создать новую переменную. Кроме того, при определении переменных рядом с местом их использования программа становится более понятной.

Возможно, эти аргументы относятся к области стилистики. Тем не менее в С++ обязательное определение всех объектов в начале блока стало бы источником серьезных проблем. Если конструктор существует, он должен быть вызван при создании объекта. Но если конструктор получает один или несколько аргументов, кто гарантирует, что данные инициализации будут доступны в начале блока? В общем случае это неизвестно. Поскольку в С не поддерживается концепция закрытых членов, отделение определения от инициализации не создает проблем. В то же время С++ гарантирует, что объект будет инициализирован одновременно с созданием, — тем самым предотвращается существование неинициализированных объектов в системе. Язык С просто игнорирует эту проблему; более того, он *способствует* появлению неинициализированных объектов, заставляя вас определять переменные в начале блока, перед тем как у вас появятся данные инициализации[1].

---

[1] С99, обновленная версия стандарта С, следует примеру С++, позволяя определять переменные в произвольной точке блока.

В общем случае C++ позволит создать объект лишь после того, как у вас появятся все необходимые данные для конструктора. Из-за этого определение переменных в начале блока неприемлемо. В сущности, стиль программирования C++ подталкивает программиста к тому, чтобы объекты определялись как можно ближе к точке их использования. В C++ любое правило, применяемое к «объекту вообще», автоматически распространяется на объекты встроенных типов. Таким образом, любые объекты класса и переменные встроенных типов могут определяться в любой точке блока. Кроме того, это означает, что определение переменной можно отложить до появления всех необходимых данных, поэтому определение всегда может выполняться одновременно с инициализацией:

```
//: C06:DefineInitialize.cpp
// Определение переменных в произвольной точке
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class G {
  int i;
public:
  G(int ii);
};

G::G(int ii) { i = ii; }

int main() {
  cout << "initialization value? ";
  int retval = 0;
  cin >> retval;
  require(retval != 0);
  int y = retval + 3;
  G g(y);
} ///:~
```

Сначала выполняется фрагмент программы, затем переменная retval определяется, инициализируется и используется для сохранения введенных данных, после чего определяются переменные y и g. Что касается языка C, он не позволяет определять переменные нигде, кроме начала блока.

В общем случае переменные следует определять как можно ближе к точке использования и всегда инициализировать их при создании (для встроенных типов инициализация не обязательна, но желательна по правилам хорошего стиля программирования). Данная рекомендация повышает безопасность: сокращение периода времени, в течение которого переменная доступна в блоке, снижает вероятность ее ошибочного использования в другой части блока. Кроме того, программа лучше читается, поскольку читателю не приходится постоянно переходить к началу блока и обратно, чтобы узнать тип переменной.

Циклы for

В C++ счетчики цикла for часто определяются прямо внутри управляющего выражения:

```
for (int j = 0; j < 100; j++) {
    cout << "j = " << j << endl;
```

```
}
for (int i = 0; i < 100; i++) {
    cout << "i = " << i << endl;
}
```

Эти команды позволяют обратить внимание на важные частные случаи, которые новички в C++ иногда понимают неправильно.

Переменные i и j определяются непосредственно внутри выражения for (в С такое невозможно). Затем они становятся доступными в цикле for. Такой синтаксис чрезвычайно удобен, поскольку смысл i и j полностью понятен из контекста и вам не приходится дополнительно уточнять его при помощи неудобочитаемых имен вида i_loop_counter.

Однако не стоит полагать, что переменные i и j продолжают существовать после завершения цикла for, — это не так[1].

В главе 3 упоминается, что команды while и switch тоже позволяют определять объекты в своих управляющих выражениях, однако эта возможность используется гораздо реже, чем в циклах for.

Остерегайтесь локальных переменных, которые скрывают переменные из внешней области видимости. В общем случае использование одного и того же имени для двух переменных — вложенной и глобальной по отношению к области видимости вложенной — создает путаницу и способствует появлению ошибок[2].

Автор считает, что небольшие области видимости являются признаком хорошего проектирования. Если одна функция занимает несколько страниц, вероятно, вы хотите от нее слишком многого. Наличие множества мелких функций не только удобнее, но и упрощает поиск ошибок.

Выделение памяти

Поскольку переменные могут определяться в любой точке блока, может показаться, что выделение памяти для переменных также откладывается до точки определения. В действительности более вероятно, что компилятор в традициях С выделит всю память блока при входе в него (то есть на открывающей фигурной скобке). Впрочем, это несущественно, поскольку вы все равно не сможете обратиться к памяти (то есть объекту) до момента определения[3]. Хотя память выделяется в начале блока, вызов конструктора происходит только в точке определения объекта, поскольку идентификатор до этого момента остается недоступным. Компилятор даже проверяет, что определение объекта (а следовательно, и вызов конструктора) не происходит в точке, мимо которой компилятор может пройти при некоторых условиях, например в команде switch или внутри перехода по команде goto. Если снять комментарии в следующей программе, компилятор выдаст предупреждение или сообщение об ошибке:

```
//: C06:Nojump.cpp
// Обход конструкторов невозможен
```

[1] В более раннем проекте стандарта C++ отмечалось, что срок жизни переменной продлевается до конца блока, в котором находится цикл for. В некоторых компиляторах эта возможность по-прежнему поддерживается, но это неправильно. Ваша программа будет переносимой только в том случае, если вы исключите все обращения к переменной за пределами цикла for.

[2] В языке Java компилятор выдает для подобного кода сообщение об ошибке.

[3] В принципе, вы можете это сделать при помощи фокусов с указателями, но, действуя таким образом, вы поступаете очень и очень плохо.

```
class X {
public:
  X();
};

X::X() {}

void f(int i) {
  if(i < 10) {
   //! goto jump1; // Ошибка: goto обходит инициализацию
  }
  X x1;  // Вызов конструктора
 jump1:
  switch(i) {
    case 1 :
      X x2;  // Вызов конструктора
      break;
  //! case 2 : // Ошибка: case обходит инициализацию
      X x3;  // Вызов конструктора
      break;
  }
}

int main() {
  f(9);
  f(11);
}///:~
```

В приведенной программе команды goto и switch теоретически могут обойти точку вызова конструктора. В этом случае объект войдет в область видимости без вызова конструктора, поэтому компилятор выдает сообщение об ошибке. Эта проверка лишний раз гарантирует, что объекты не смогут создаваться без инициализации.

Конечно, вся память, о которой здесь рассказывалось, выделяется из стека. Для этого компилятор смещает указатель стека «вниз» (относительный термин, который может обозначать как увеличение, так и уменьшение фактического значения указателя стека в зависимости от компьютера). Память под объекты также может выделяться из кучи оператором new; эта тема будет рассматриваться в главе 13.

Класс Stash с конструктором и деструктором

В примерах предыдущей главы присутствовали функции, очевидно подходящие на роль конструкторов и деструкторов: initialize() и cleanup(). Вот как выглядит заголовок класса Stash с конструктором и деструктором:

```
//: C06:Stash2.h
// Версия с конструктором и деструктором
#ifndef STASH2_H
#define STASH2_H

class Stash {
  int size;      // Размер каждого элемента
  int quantity;  // Количество элементов
```

```
  int next;      // Следующий пустой элемент
  // Динамически выделяемый байтовый массив:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH2_H ///:~
```

В реализации изменяются только определения функций initialize() и cleanup(), которые заменяются соответственно конструктором и деструктором:

```
//: C06:Stash2.cpp {O}
// Конструктор и деструктор
#include "Stash2.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(void* element) {
  if(next >= quantity) // В буфере есть свободное место?
    inflate(increment);
  // Скопировать элемент в следующую свободную позицию буфера:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Индекс
}

void* Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // Признак конца
  // Указатель на запрашиваемый элемент:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Количество элементов в Stash
}

void Stash::inflate(int increase) {
```

```
    require(increase > 0,
      "Stash::inflate zero or negative increase");
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
      b[i] = storage[i]; // Копирование старого буфера в новый
    delete [](storage); // Освобождение старого буфера
    storage = b; // Перевод указателя на новый буфер
    quantity = newQuantity;
  }

  Stash::~Stash() {
    if(storage != 0) {
     cout << "freeing storage" << endl;
     delete []storage;
    }
  } ///:~
```

В этом примере вместо assert() для выявления ошибок программирования также используются функции из файла require.h. Результат неудачной проверки assert() несет меньше полезной информации, чем функции из файла require.h, которые будут рассматриваться позднее в этой главе.

Поскольку функция inflate() объявлена закрытой, сбой в ней возможен только в том случае, если другая функция класса случайно передаст в функцию inflate() неправильное значение. Если вы твердо уверены, что такого быть не может, попробуйте удалить require(), но помните, что до полной настройки класса всегда существует вероятность того, что в нем появится новый код, из-за которого в программе возникнут ошибки. Вызов require() обходится недорого и может быть автоматически удален при помощи препроцессора, а выигрыш от повышения надежности кода велик.

В следующей тестовой программе обратите внимание на определение объектов Stash непосредственно перед их использованием, а также на выполняемую при этом инициализацию на основании аргументов конструктора:

```
//: C06:Stash2Test.cpp
//{L} Stash2
// Конструктор и деструктор
#include "Stash2.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
```

```
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize);
  ifstream in("Stash2Test.cpp");
  assure(in, " Stash2Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

Также обратите внимание на то, что вызовы cleanup() исчезли из программы, но деструкторы вызываются автоматически при выходе объектов intStash и stringStash из области видимости.

И еще одно обстоятельство, о котором следует помнить в примерах Stash: автор специально ограничился встроенными типами, не имеющими деструкторов. Если вы попытаетесь скопировать в Stash объекты классов, возникнут всевозможные проблемы, и программа будет работать неправильно. На самом деле стандартная библиотека C++ позволяет правильно организовать копирование объектов в контейнерах, но это довольно сложный и запутанный процесс. В следующем примере Stack показано, как эта проблема обходится при помощи указателей, а в одной из следующих глав класс Stash будет переработан с применением указателей.

Класс Stack с конструктором и деструктором

Реализация связанного списка (внутри класса Stack) с применением конструкторов и деструкторов показывает, как хорошо конструкторы и деструкторы работают в сочетании с операторами new и delete. Измененный заголовочный файл выглядит так:

```
//: C06:Stack3.h
// Стек с конструктором и деструктором
#ifndef STACK3_H
#define STACK3_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt);
    ~Link();
  }* head;
public:
  Stack();
  ~Stack();
  void push(void* dat);
  void* peek();
  void* pop();
};
#endif // STACK3_H ///:~
```

Теперь конструктор и деструктор определяются не только в классе **Stack**, но и во вложенном классе **Link**:

```
//: C06:Stack3.cpp {O}
// Конструкторы и деструкторы
#include "Stack3.h"
#include "../require.h"
using namespace std;

Stack::Link::Link(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

Stack::Link::~Link() { }

Stack::Stack() { head = 0; }

void Stack::push(void* dat) {
  head = new Link(dat,head);
}

void* Stack::peek() {
  require(head != 0, "Stack empty");
  return head->data;
}

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

Stack::~Stack() {
  require(head == 0, "Stack not empty");
} ///:~
```

Конструктор **Link::Link()** просто инициализирует указатели **data** и **next**, поэтому в функции **Stack::push()** следующая строка не только создает новый объект **Link** (путем динамического выделения памяти оператором **new**, о чем рассказывалось в главе 4), но и инициализирует указатели на этот объект:

```
head = new Link(dat,head);
```

Возможно, у вас возник вопрос: почему деструктор класса **Link** ничего не делает, в частности почему он не вызывает оператор **delete** для указателя **data**? По двум причинам. В главе 4 при описании класса **Stack** было отмечено, что **delete** не может вызываться для указателя **void***, который ссылается на объект (это утверждение будет доказано в главе 13). Но, кроме того, если бы деструктор класса **Link** удалял указатель **data**, то функция **pop()** возвращала бы указатель на удаленный объект, что заведомо является ошибкой. Иногда это называется *принадлежностью*, или *владением*: класс **Link**, а следовательно, и класс **Stack**, только содержит указатели, но не отвечает за их освобождение. То есть вы должны очень хорошо понимать, кто за это *отвечает*. Например, если не вызвать

pop() и delete для всех указателей в объекте Stack, они не будут автоматически освобождены деструктором класса Stack. В результате возможны самые серьезные последствия и утечка памяти, а незнание того, кто отвечает за освобождение объектов, может порождать ошибки в программе. Вот почему вызов Stack::~Stack() выводит сообщение об ошибке, если объект Stack на момент уничтожения содержит элементы.

Выделение и освобождение памяти объектов Link скрыто внутри класса Stack и является частью базовой реализации. В тестовой программе эти операции не видны, хотя вы *обязаны* освободить указатели, возвращаемые функцией pop():

```
//: C06:Stack3Test.cpp
//{L} Stack3
//{T} Stack3Test.cpp
// Конструкторы и деструкторы
#include "Stack3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Аргумент - имя файла
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Чтение файла и сохранение строк в стеке:
  while(getline(in, line))
    textlines.push(new string(line));
  // Извлечение строк из стека и вывод:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
} ///:~
```

В данном примере все строки textlines извлекаются из стека и удаляются, а если бы этого не произошло, функция require() вернула бы сообщение об утечке памяти.

Агрегатная инициализация

Агрегатом (или составным значением) называется совокупность значений, объединенных по общему признаку. Под это определение подходят агрегаты смешанных типов, такие как структуры и классы. Массив представляет собой агрегат одного типа.

Инициализация агрегатов утомительна и чревата ошибками. Механизм *агрегатной инициализации* C++ значительно повышает ее надежность. При создании агрегатного объекта от вас требуется лишь выполнить присваивание, а компилятор сам позаботится об инициализации. Существует несколько разновидностей присваивания в зависимости от типа агрегата, но во всех случаях список значений,

используемых при присваивании, заключается в фигурные скобки. Для массива встроенных типов конструкция выглядит очень просто:

```
int a[5] = { 1, 2, 3, 4, 5 };
```

Если количество инициализирующих значений больше количества элементов в массиве, компилятор выдает сообщение об ошибке. Но что произойдет, если инициализирующих значений будет *меньше*? Пример:

```
int b[6] = {0};
```

В данном случае компилятор использует первое значение для первого элемента массива, а остальные элементы будут заданы без инициализирующих значений. Учтите, что этот способ не применяется, если массив определяется без списка инициализирующих значений. Таким образом, приведенное выше выражение является удобным инструментом инициализации массивов нулями без применения цикла for с его теоретически возможным выходом за границу массива (в зависимости от компилятора подобное выражение может также превосходить цикл for по эффективности).

Во второй форме сокращенной записи выполняется *автоматический подсчет* элементов массива. В этом случае компилятор определяет размер массива по количеству инициализирующих значений:

```
int c[] = { 1, 2, 3, 4 };
```

Если позднее потребуется добавить в массив новый элемент, достаточно включить в список дополнительное значение. Программа, в которой все изменения локализуются в одном месте, снижает вероятность внесения ошибок при последующих модификациях. Но как узнать фактический размер массива в программе? Выражение sizeof c / sizeof *c (размер всего массива, разделенный на размер первого элемента) возвращает нужное значение при любых изменениях размеров массива:

```
for (int i = 0; i < sizeof c / sizeof *c; i++)
  c[i]++;
```

Структуры также являются агрегатами, поэтому описанная выше схема инициализации применима и к ним. В структурах С все переменные являются открытыми, что позволяет напрямую присваивать им значения:

```
struct X {
  int i;
  float f;
  char c;
};

X x1 = { 1, 2.2, 'c' };
```

Если у вас имеется массив таких объектов, то при его инициализации данные каждого объекта перечисляются в отдельной паре вложенных фигурных скобок:

```
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
```

В данном примере третий объект инициализируется нулями.

Но если какие-либо переменные класса объявлены закрытыми (что характерно для хорошо спроектированных классов C++), и даже если все переменные являются открытыми, но у класса есть конструктор, ситуация меняется. В приведенных выше примерах инициализирующие значения присваиваются непосред-

ственно элементам агрегата, но конструкторы дают возможность выполнить инициализацию через формальный интерфейс класса. В этом случае инициализация выполняется вызовом конструкторов. Предположим, в программе определена структура

```
struct Y {
  float f;
  int i;
  Y(int a);
};
```

В список инициализации включаются вызовы конструкторов. Лучше всего воспользоваться явными вызовами вида

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

Команда создает три объекта тремя вызовами конструктора. При наличии конструктора вся инициализация должна производиться через конструктор, даже если инициализация агрегатная. Это относится как к структурам, содержащим только открытые переменные, так и к классам с закрытыми переменными.

Второй пример демонстрирует вызов конструктора с несколькими аргументами:

```
//: C06:Multiarg.cpp
// Вызов конструктора с несколькими аргументами
// при агрегатной инициализации
#include <iostream>
using namespace std;

class Z {
  int i, j;
public:
  Z(int ii, int jj);
  void print();
};

Z::Z(int ii, int jj) {
  i = ii;
  j = jj;
}

void Z::print() {
  cout << "i = " << i << ", j = " << j << endl;
}

int main() {
  Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
  for(int i = 0; i < sizeof zz / sizeof *zz; i++)
    zz[i].print();
} ///:~
```

Обратите внимание: все выглядит так, словно каждый объект в массиве инициализируется явным вызовом конструктора.

Конструктор по умолчанию

Конструктор по умолчанию вызывается без аргументов. Он создает простейший, «типовой», объект и используется в ситуациях, когда компилятор должен построить

объект, не располагая никакой информацией о нем. Например, если взять приведенное выше определение структуры Y и использовать ее в следующем определении, компилятор пожалуется на отсутствие конструктора по умолчанию:

```
Y y2[2] = { Y(1) };
```

Второй объект массива должен создаваться без аргументов, а для этого компилятору нужен конструктор по умолчанию. Более того, если просто определить в программе массив объектов Y, как показано ниже, то компилятор все равно выдаст сообщение об ошибке, потому что для инициализации каждого объекта в массиве ему необходим конструктор по умолчанию:

```
Y y3[7];
```

Аналогичная проблема возникает и при создании отдельных объектов:

```
Y y4;
```

Помните: если в классе определен конструктор, то компилятор следит за тем, чтобы этот конструктор вызывался в *любом* случае, независимо от ситуации.

Конструктор по умолчанию настолько важен, что при полном отсутствии конструкторов в структуре или классе (и только в этом случае!) компилятор автоматически генерирует их за вас. Это означает, что следующий фрагмент будет нормально работать:

```
//: C06:AutoDefaultConstructor.cpp
// Автоматически сгенерированный конструктор по умолчанию

class V {
  int i;  // Закрытая переменная
}; // Конструктора нет

int main() {
  V v, v2[10];
} ///:~
```

Однако если бы в классе был определен какой-либо конструктор, а конструктор по умолчанию отсутствовал, то попытка создания экземпляра V привела бы на этапе компиляции к выдаче сообщения об ошибке.

Возникает предположение, что конструктор, сгенерированный компилятором, выполняет некую разумную инициализацию, например заполняет всю память объекта нулями. Однако в действительности этого не происходит, поскольку такая инициализация увеличивала бы затраты, но не могла контролироваться программистом. Если вы хотите, чтобы память инициализировалась нулями, вам придется самостоятельно написать конструктор по умолчанию.

Хотя компилятор генерирует конструктор по умолчанию, поведение этого конструктора редко соответствует нашим ожиданиям. Рассматривайте эту возможность как дополнительную меру страховки, но старайтесь воздерживаться от ее использования. В общем случае следует определять конструкторы явно и не позволять компилятору делать это за вас.

Итоги

Тщательно проработанные механизмы инициализации и зачистки в C++ дают представление о том, сколь важная роль отводится им в этом языке. В процессе

разработки C++ Страуструп пришел к выводу, что источником значительной части ошибок программирования является некорректная инициализация переменных. Такие ошибки выявляются с большим трудом, причем из-за некорректной зачистки возникают аналогичные проблемы. Поскольку конструкторы и деструкторы *гарантируют* корректное выполнение инициализации и зачистки (компилятор не позволит создать или уничтожить объект без вызова конструктора или деструктора), программист в полной мере контролирует происходящие события, а программа становится более надежной.

Агрегатная инициализация преследует ту же цель — она предотвращает типичные ошибки инициализации агрегатов встроенных типов и делает программу более компактной.

В C++ большое внимание уделяется безопасности программирования. Инициализация и зачистка играют важную роль в этой области, а по мере изложения материала вы познакомитесь и с другими аспектами безопасности.

Упражнения

1. Напишите простой класс Simple с конструктором, который должен выводить сообщение о вызове. Создайте в функции main() объект этого класса.
2. Включите в класс из упражнения 1 деструктор, который будет выводить сообщение о вызове.
3. Добавьте в класс из упражнения 2 переменную типа int. Измените конструктор так, чтобы при вызове ему передавался аргумент int, значение которого должно сохраняться в переменной класса. И конструктор, и деструктор должны выводить значение int в своих сообщениях; это позволит отслеживать моменты создания и уничтожения объектов.
4. Докажите, что деструкторы вызываются даже при выходе из цикла с помощью команды goto.
5. Напишите два цикла for для вывода значений от 0 до 10. В первом цикле определите счетчик до начала цикла, а во втором — в управляющем выражении. После проверки циклов назначьте счетчику второго цикла такое же имя, как и счетчику первого, а затем посмотрите на реакцию компилятора.
6. Измените файлы Handle.h, Handle.cpp и UseHandle.cpp (см. конец главы 5) так, чтобы в них использовались конструкторы и деструкторы.
7. Создайте массив типа double с использованием агрегатной инициализации. Размер массива должен быть задан, но количество инициализирующих значений в списке должно быть меньше количества элементов. Выведите содержимое массива, определив его размер при помощи оператора sizeof. Затем создайте массив double с использованием агрегатной инициализации *и* автоматическим подсчетом элементов. Выведите содержимое массива.
8. Используйте агрегатную инициализацию для создания массива объектов string. Создайте объект Stack для хранения объектов string и в цикле перебе-

рите элементы массива; при каждой итерации текущий объект string должен заноситься в стек. Затем извлеките объекты string из стека с выводом их значений.

9. Продемонстрируйте механизмы автоматического подсчета элементов и агрегатной инициализации на примере массива объектов из упражнения 3. Включите в этот класс функцию для вывода сообщения. Вычислите размер массива и переберите его элементы, вызывая новую функцию.
10. Создайте класс без конструкторов и убедитесь в том, что в программе можно создавать объекты конструктором по умолчанию. Затем определите для класса конструктор, которому при вызове передается аргумент, и попробуйте снова откомпилировать программу. Объясните, что происходит.

Перегрузка функций и аргументы по умолчанию

Удобная система именования относится к числу важнейших аспектов любого языка программирования. Создавая объект (переменную), вы связываете имя с некоторой областью памяти, а функция является именем операции, выполняемой с этой областью памяти. Если система описывается с помощью разумно выбранных и понятных имен, программа будет легко читаться и редактироваться. Программист вообще сродни писателю — тот и другой стремятся донести свою мысль до читателей.

При переводе естественного языка на язык программирования возникает проблема. Одно и то же слово может обладать разным смыслом в зависимости от контекста. Иначе говоря, смысл слов *перегружается*. Смысловая перегрузка очень удобна, особенно если речь идет о тривиальных различиях. Мы говорим «помыть машину, помыть пол»; глупо было бы говорить «машина::помыть машину, пол::помыть пол» только для того, чтобы слушатель точно понимал, какая именно операция применяется в данном случае. Естественным языкам изначально присуща некоторая избыточность, так что даже если пропустить пару слов, смысл все равно останется понятным. Нам не нужна абсолютная точность в определениях — смысл часто удается определить по контексту.

Однако в большинстве языков программирования каждая функция должна снабжаться уникальным идентификатором. Если программа работает с тремя разными типами данных int, char и float, то для их вывода обычно приходится создавать три разные функции: print_int, print_char и print_float. Лишние функции усложняют жизнь как программисту, пишущему программу, так и читателям, которые пытаются в ней разобраться.

В С++ также существует дополнительный фактор, требующий перегрузки имен функций. Речь идет о конструкторах. Поскольку имя конструктора определяется именем класса, возникает впечатление, что конструктор может быть только один. А что, если объекты могут создаваться несколькими способами? Предположим, вы построили класс, объекты которого могут инициализироваться либо стандарт-

ным способом, либо на основании данных, загруженных из файла. В этом случае необходимы два конструктора: первый будет вызываться без аргументов (конструктор по умолчанию), а второй получит аргумент типа string — имя файла с данными инициализации. Обе функции являются конструкторами, поэтому они должны иметь одинаковые имена, совпадающие с именем класса. Следовательно, перегрузка функций абсолютно необходима для того, чтобы одно имя функции (в данном случае конструктор) могло использоваться с разными типами аргументов.

Хотя механизм перегрузки функций абсолютно необходим для конструкторов, он имеет глобальный характер и может использоваться с любыми функциями, не только с функциями классов. Кроме того, перегрузка функций означает, что наличие одноименных функций в двух библиотеках не приведет к конфликту, если эти функции вызываются с разными типами аргументов. Все эти вопросы подробно здесь рассматриваются.

Основная тема этой главы — удобство использования имен функций. Перегрузка позволяет задействовать одно имя для нескольких функций, но существует другая возможность сделать вызов функции более удобным. Если функция вызывается с большим количеством аргументов, многие из которых имеют одни и те же значения, повторять их неудобно и утомительно, не говоря уже о неудобстве чтения. В таких случаях в C++ применяются *аргументы по умолчанию*, то есть значения, которые автоматически подставляются компилятором, если аргумент не был указан при вызове. Таким образом, вызовы f("hello"), f("hi",1) и f("howdy",2,'c') могут быть вызовами одной и той же функции. В принципе, эти вызовы с тем же успехом могут быть вызовами трех перегруженных функций, но похожие списки аргументов обычно соответствуют сходному поведению, что более естественно реализовать в виде одной функции.

Механизмы перегрузки функций и передачи аргументов по умолчанию не вызывают особых сложностей. К концу этой главы вы поймете, как ими пользоваться и как они реализуются во время компиляции и компоновки.

Снова об украшении имен

В главе 4 была представлена концепция *украшения имен*. В следующем фрагменте функция f() в области видимости класса X не конфликтует с глобальной версией f():

```
void f();
class X { void f(); };
```

Чтобы различать эти функции, компилятор присваивает разные внутренние имена глобальной версии f() и X::f(). В главе 4 упоминалось о том, что эти внутренние имена состоят из имени класса, объединенного с именем функции, то есть из конструкций вида _f и _X_f. Однако на практике выясняется, что украшение имен функций не сводится к включению в идентификатор имени класса.

Допустим, вы хотите перегрузить два имени функции:

```
void print(char);
void print(float);
```

Неважно, где определяются эти функции, в классе или в глобальной области видимости. Только имена функций не позволят компилятору построить уникальные внутренние идентификаторы. В обоих случаях будет сгенерировано имя _print.

Принцип перегрузки функций заключается в использовании одинаковых имен с разными списками аргументов. Следовательно, чтобы этот принцип работал, компилятор должен включить в имя функции информацию о типах аргументов. Для функций из предыдущего примера, определенных в глобальной области видимости, будут сгенерированы внутренние имена вида _print_char и _print_float. Учтите, что правила украшения имен компилятором не оговорены в стандарте, поэтому результаты различаются в зависимости от компилятора (чтобы увидеть, как выглядят сгенерированные имена, с помощью компилятора получите и изучите ассемблерный код). Конечно, это вызовет проблемы, если вы захотите приобрести библиотеку, откомпилированную для конкретного компилятора и компоновщика. Впрочем, даже если бы способ украшения имен был стандартизирован, это лишь породило бы новые проблемы, поскольку разные компиляторы генерируют разный исполняемый код.

Собственно, вот и все, что необходимо знать о перегрузке функций: одно имя может назначаться разным функциям, если различаются списки аргументов. На основании имени функции, области видимости и типов аргументов компилятор строит внутренние имена, используемые в работе самого компилятора и компоновщика.

Перегрузка по типу возвращаемого значения

Часто приходится слышать вопрос: «Почему только область видимости и списки аргументов? Почему не возвращаемые значения?» На первый взгляд кажется, что во внутреннее имя функции было бы логично включить тип возвращаемого значения. Это позволило бы выполнять перегрузку и по возвращаемым значениям:

```
void f();
int f();
```

Такая перегрузка отлично работает, если компилятор может однозначно выбрать версию функции по контексту:

```
int x=f();
```

Однако в языке C программист всегда мог вызвать функцию и проигнорировать возвращаемое значение (то есть вызвать функцию ради *побочных эффектов* ее выполнения). Как компилятор узнает, какой вызов функции имеется в виду в таких случаях? Но еще хуже то, что читателю программы тоже будет трудно понять, о какой функции идет речь. Перегрузка только по типу возвращаемого значения создает слишком много проблем, поэтому в C++ она не поддерживается.

Безопасность типов при компоновке

У механизма украшения имен есть еще одно дополнительное преимущество. В языке C существует крайне неприятная разновидность ошибок, когда прикладной программист ошибается при объявлении функции (или что еще хуже, функция вызывается без предварительного объявления), и компилятор вычисляет объявление функции по способу ее вызова. Иногда полученное таким образом объявление оказывается правильным, но в противном случае в программе возникает ошибка, которую очень трудно обнаружить.

В C++ все функции *должны* объявляться перед использованием, поэтому вероятность ошибок такого рода значительно меньше. Поскольку компилятор C++

отказывается автоматически объявлять функцию за программиста, скорее всего, вы включите в программу нужный заголовочный файл. А если по какой-либо причине функция все же будет объявлена неправильно — либо вручную, либо из-за включения ошибочного (например, старого) заголовочного файла, — механизм украшения имен предоставляет дополнительную страховку, иногда называемую *компоновкой, безопасной по отношению к типам*, или, для краткости, *безопасной компоновкой*.

Рассмотрим следующую ситуацию. Имеется файл с определением функции:

```
//: C07:Def.cpp {O}
// Определение функции
void f(int) {}
///:~
```

Во втором файле содержится ошибочное объявление функции и ее вызов:

```
//: C07:Use.cpp
//{L} Def
// Неправильное объявление функции
void f(char);

int main() {
//! f(1); // Происходит ошибка компоновки
} ///:~
```

Мы видим, что функция в действительности определена в виде f(int), но компилятор этого не знает, поскольку в программе присутствует явное объявление f(char). Компиляция проходит успешно. В С компоновка тоже проходит успешно, а вот в C++ нет. Поскольку компилятор преобразует имена, определение приобретает вид f_int, тогда как в программе используется функция f_char. Компоновщик пытается разрешить ссылку на f_char, но находит только f_int и выдает сообщение об ошибке. В этом и заключается суть безопасной компоновки. Хотя данная ошибка возникает не часто, обнаружить ее невероятно сложно, особенно в большом проекте. Это один из тех случаев, когда неуловимая ошибка в программе С обнаруживается простой обработкой программы компилятором C++.

Пример перегрузки

Давайте изменим примеры из предыдущих глав так, чтобы в них перегружались функции. Как упоминалось выше, самое очевидное применение перегрузки имеет место в случае конструкторов. Примером такого рода является очередная версия класса Stash:

```
//: C07:Stash3.h
// Перегрузка функций
#ifndef STASH3_H
#define STASH3_H

class Stash {
  int size;      // Размер каждого элемента
  int quantity;  // Количество элементов
  int next;      // Следующий пустой элемент
  // Динамически выделяемый байтовый массив:
  unsigned char* storage;
  void inflate(int increase);
```

```
public:
  Stash(int size); // Нулевое количество элементов
  Stash(int size, int initQuantity);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH3_H ///:~
```

Первый конструктор Stash() остался без изменений, но наряду с ним появился второй конструктор с аргументом **quantity**, показывающим исходное количество элементов в контейнере. Из определения видно, что внутреннее значение **quantity** устанавливается равным нулю вместе с указателем **storage**. Во втором конструкторе вызов **inflate(initQuantity)** увеличивает **quantity** до заданной величины:

```
//: C07:Stash3.cpp {O}
// Перегрузка функций
#include "Stash3.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuantity);
}

Stash::~Stash() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
    delete []storage;
  }
}

int Stash::add(void* element) {
  if(next >= quantity) // В буфере есть свободное место?
    inflate(increment);
  // Скопировать элемент в следующую свободную позицию буфера:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Индекс
}

void* Stash::fetch(int index) {
```

```
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // Признак конца
  // Указатель на запрашиваемый элемент:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Количество элементов в CStash
}

void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Копирование старого буфера в новый
  delete [](storage); // Освобождение старого буфера
  storage = b; // Перевод указателя на новый буфер
  quantity = newQuantity; // Изменение размера
} ///:~
```

При использовании первого конструктора память для **storage** не выделяется. Выделение памяти происходит при первом вызове **add()** для объекта, а также при переполнении буфера в результате вызова **add()**.

Примеры использования обоих конструкторов приводятся в следующей тестовой программе:

```
//: C07:Stash3Test.cpp
//{L} Stash3
// Перегрузка функций
#include "Stash3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash3Test.cpp");
  assure(in, "Stash3Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
```

```
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

При вызове конструктора для stringStash передается второй аргумент, причем считается, что вы располагаете дополнительной информацией, позволяющей выбрать начальный размер Stash.

Объединения

Как мы выяснили в предыдущих главах, единственное различие между структурой и классом в C++ заключается в том, что по умолчанию члены структуры считаются открытыми (public), тогда как члены класса по умолчанию считаются закрытыми (private). Вполне естественно, что и у структур имеются свои конструкторы и деструкторы. Кроме того, оказывается, что помимо структур обладать конструкторами, деструкторами, функциями и даже спецификаторами доступа могут также объединения (union). Следующий пример демонстрирует и снова подтверждает преимущества перегрузки:

```
//: C07:UnionClass.cpp
// Объединение с конструктором и функциями
#include<iostream>
using namespace std;

union U {
private: // Управление доступом тоже поддерживается!
  int i;
  float f;
public:
  U(int a);
  U(float b);
  ~U();
  int read_int();
  float read_float();
};

U::U(int a) { i = a; }

U::U(float b) { f = b;}

U::~U() { cout << "U::~U()\n"; }

int U::read_int() { return i; }

float U::read_float() { return f; }

int main() {
  U X(12), Y(1.9F);
  cout << X.read_int() << endl;
  cout << Y.read_float() << endl;
} ///:~
```

При виде этого листинга возникает предположение, что объединения отличаются от классов только способом хранения данных (то есть тем, что данные int и float хранятся в общем участке памяти). Но в действительности существуют и другие различия: объединения не могут использоваться в качестве базовых классов при

наследовании, что является существенным ограничением с точки зрения объектно-ориентированного программирования (наследование рассматривается в главе 14).

Хотя функции-члены делают доступ к объединениям чуть более «цивилизованным», прикладному программисту по-прежнему ничто не мешает выбрать неверный тип элемента после инициализации объединения. Так, в предыдущем примере можно вызвать функцию X.read_float(), даже если она не соответствует фактическому типу хранимых данных. Впрочем, «безопасность» объединения может быть обеспечена за счет инкапсуляции в классе. В следующем примере обратите внимание на то, как применение ключевого слова enum проясняет смысл программы, а также на удобную перегрузку конструкторов:

```
//: C07:SuperVar.cpp
// Универсальная переменная
#include <iostream>
using namespace std;

class SuperVar {
  enum {
    character,
    integer,
    floating_point
  } vartype;  // Определение типа
  union {  // Анонимное объединение
    char c;
    int i;
    float f;
  };
public:
  SuperVar(char ch);
  SuperVar(int ii);
  SuperVar(float ff);
  void print();
};

SuperVar::SuperVar(char ch) {
  vartype = character;
  c = ch;
}

SuperVar::SuperVar(int ii) {
  vartype = integer;
  i = ii;
}

SuperVar::SuperVar(float ff) {
  vartype = floating_point;
  f = ff;
}

void SuperVar::print() {
  switch (vartype) {
    case character:
      cout << "character: " << c << endl;
      break;
    case integer:
      cout << "integer: " << i << endl;
      break;
    case floating_point:
```

```
      cout << "float: " << f << endl;
      break;
    }
  }

int main() {
  SuperVar A('c'), B(12), C(1.44F);
  A.print();
  B.print();
  C.print();
} ///:~
```

В этой программе перечисление (enum) определяется без указания имени типа; это вполне приемлемо, если вы собираетесь немедленно использовать перечисляемый тип для определения экземпляра, как это сделано в программе. В будущем нам не придется ссылаться на перечисление по имени, поэтому имя в данном случае не обязательно.

Объединение объявляется без имени типа и без имени переменной. *Анонимные объединения* резервируют память для хранения данных, но не требуют указания экземпляра при обращении к членам. Для примера рассмотрим следующее анонимное объединение:

```
//: C07:AnonymousUnion.cpp
int main() {
  union {
    int i;
    float f;
  };
  // Обращения к членам выполняются без указания экземпляра:
  i = 12;
  f = 1.22;
} ///:~
```

Обратите внимание: мы обращаемся к членам анонимного объединения так, словно они являются обычными переменными. Единственное различие состоит в том, что переменные занимают одну область памяти. Если анонимное объединение имеет файловую видимость, то есть находится за пределами функций и классов, то оно должно быть объявлено статическим, чтобы для него выполнялось внутреннее связывание.

Хотя класс SuperVar стал безопасным, его полезность несколько сомнительна. Объединения используются прежде всего для экономии памяти, но добавление vartype требует памяти, так что весь выигрыш фактически теряется. Существует пара альтернативных решений, которые могли бы сделать эту схему более практичной. Если бы перечисление vartype управляло сразу несколькими экземплярами объединений, то на всю группу было бы достаточно одного перечисления. В другом, более эффективном решении весь код vartype заключается в директивы #ifdef, которые гарантируют его правильное использование на стадии разработки и тестирования. В окончательной версии программы дополнительные затраты памяти и времени будут исключены.

Аргументы по умолчанию

Присмотритесь к конструкторам Stash() в заголовке Stash3.h. Они не так уж сильно отличаются, не правда ли? В сущности, первый конструктор выглядит

как частный случай второго с нулевым исходным значением size. Создание и сопровождение двух разных версий одной функции выглядит несколько неэффективно.

В C++ для таких ситуаций предусмотрен механизм определения *аргументов по умолчанию*. Аргумент по умолчанию представляет собой значение, которое задается в объявлении и автоматически подставляется компилятором, если аргумент не был задан при вызове функции. Вспомним конструкторы в примере с классом Stash:

```
Stash(int size); // Нулевое количество элементов
Stash(int size, int initQuantity);
```

Эти две функции можно заменить одной функцией:

```
Stash(int size, int initQuantity = 0);
```

Определение Stash(int) просто удаляется — достаточного одного определения Stash(int,int).

В новой версии следующие два определения объектов дают в точности одинаковые результаты:

```
Stash A(100), B(100,0);
```

В обоих случаях вызывается один и тот же конструктор, но для A компилятор автоматически подставляет второй аргумент, когда обнаруживает, что первый аргумент относится к типу int, а второй аргумент не указан. Компилятор осведомлен об аргументе по умолчанию, поэтому знает, что правильный вызов функции может быть получен в результате подстановки второго аргумента, для чего и был объявлен аргумент по умолчанию.

Аргументы по умолчанию являются вспомогательным средством, как и перегрузка функций. Обе возможности позволяют использовать одно имя функции в разных ситуациях. Различие состоит в том, что при определении аргументов по умолчанию компилятор сам подставляет аргументы, когда это не хотите делать вы. В предыдущем примере вместо перегрузки функций лучше воспользоваться аргументами по умолчанию, иначе в программе появится несколько функций со сходными сигнатурами и сходным поведением. Как правило, если поведение функций сильно различается, нет смысла задействовать аргументы по умолчанию (более того, стоит подумать, нужно ли присваивать двум столь разным функциям одинаковые имена).

Используя аргументы по умолчанию, необходимо помнить о двух правилах. Во-первых, значения по умолчанию могут назначаться только для аргументов, завершающих список. Нельзя объявить аргумент по умолчанию, за которым будет следовать аргумент с явно задаваемым значением. Во-вторых, после первого аргумента по умолчанию все последующие аргументы в списке также должны быть аргументами по умолчанию (следствие из первого правила).

Аргументы по умолчанию задаются только в объявлении функции (которое обычно находится в заголовочном файле). Компилятор должен знать о существовании значения по умолчанию, прежде чем он сможет его использовать. Некоторые программисты в целях документирования кода указывают закомментированные значения по умолчанию в определениях функций:

```
void fn(int x /* = 0 */) { //...
```

Заполнители в списке аргументов

Аргументы в объявлении функции могут объявляться без указания идентификатора. В сочетании с аргументами по умолчанию результат выглядит довольно забавно. В программе появляются конструкции вида

```
void f(int x, int = 0, float = 1.1);
```

В C++ определение функции также не требует обязательного указания всех идентификаторов:

```
void f(int x, int, float flt) { /* ... */ }
```

Тело функции может содержать ссылки на x и flt, но не на средний аргумент, у которого нет имени. Однако при вызовах функции необходимо передать значение аргумента-заполнителя: f(1) или f(1,2,3.0). Синтаксис заполнителей позволяет включить аргумент в список, не используя его. Это делается для того, чтобы в будущем определение функции можно было изменять без модификации кода, содержащего вызовы этой функции. Конечно, то же самое можно сделать при помощи именованного аргумента, но если определить аргумент и не задействовать его в теле функции, многие компиляторы выдадут предупреждение, полагая, что в программе допущена логическая ошибка. Намеренно опуская имя аргумента, вы подавляете это предупреждение.

Но еще важнее другое: если вы начнете использовать аргумент функции, а позднее решите, что он вам не нужен, этот аргумент можно легко исключить без предупреждений и без нарушения работы клиентского кода, в котором вызывается предыдущая версия функции.

Выбор между перегрузкой и аргументами по умолчанию

Как перегрузка, так и аргументы по умолчанию представляют собой вспомогательные механизмы для многократного использования имен функций. Тем не менее в отдельных случаях бывает трудно выбрать лучший в данной ситуации механизм. Для примера рассмотрим следующую программу, предназначенную для автоматизации управления блоками памяти:

```
//: C07:Mem.h
#ifndef MEM_H
#define MEM_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem();
  Mem(int sz);
  ~Mem();
  int msize();
  byte* pointer();
  byte* pointer(int minSize);
};
#endif // MEM_H ///:~
```

Объект Mem содержит байтовый буфер и следит за тем, чтобы объем буфера был достаточным. Конструктор по умолчанию вообще не выделяет память, а второй конструктор выделяет блок объемом sz. Деструктор освобождает память, функция msize() сообщает текущий размер буфера в байтах, а функция pointer() возвращает указатель на начальный адрес буфера (объект Mem работает на достаточно низком уровне). Также существует перегруженная версия pointer(), при помощи которой прикладной программист указывает, что ему нужен указатель на буфер объемом не менее minSize; перегруженная версия функции pointer() автоматически обеспечивает выполнение этого условия.

И конструктор, и функция pointer() увеличивают размер буфера при помощи закрытой функции ensureMinSize() (обратите внимание, что из-за возможного перераспределения памяти сохранять результат pointer() небезопасно).

Ниже представлена реализация класса:

```
//: C07:Mem.cpp {O}
#include "Mem.h"
#include <cstring>
using namespace std;

Mem::Mem() { mem = 0; size = 0; }

Mem::Mem(int sz) {
  mem = 0;
  size = 0;
  ensureMinSize(sz);
}

Mem::~Mem() { delete []mem; }

int Mem::msize() { return size; }

void Mem::ensureMinSize(int minSize) {
  if(size < minSize) {
    byte* newmem = new byte[minSize];
    memset(newmem + size, 0, minSize - size);
    memcpy(newmem, mem, size);
    delete []mem;
    mem = newmem;
    size = minSize;
  }
}

byte* Mem::pointer() { return mem; }

byte* Mem::pointer(int minSize) {
  ensureMinSize(minSize);
  return mem;
} ///:~
```

Вы видите, что выделение памяти выполняется только функцией ensureMinSize(), причем эта функция используется вторым конструктором и второй перегруженной версией функции pointer(). Если при вызове ensureMinSize() значение size достаточно велико, делать ничего не нужно. Если для увеличения блока требуется выделить новую память (что также происходит после создания блока нулевого размера конструктором по умолчанию), новая «добавочная» часть заполняется

нулями при помощи стандартной функции C memset(), упоминавшейся в главе 5. Затем вызывается функция memcpy() стандартной библиотеки C, которая в данном случае копирует существующие байты из mem в newmem (обычно копирование выполняется весьма эффективно).

Класс Mem спроектирован так, чтобы другие классы могли его использовать для упрощения операций с памятью (также он может применяться для инкапсуляции нетривиальных инструментов управления памятью, например, предоставляемых операционной системой). Для тестирования Mem создадим простой «псевдостроковый» класс:

```
//: C07:MemTest.cpp
// Тестирование класса Mem
//{L} Mem
#include "Mem.h"
#include <cstring>
#include <iostream>
using namespace std;

class MyString {
  Mem* buf;
public:
  MyString();
  MyString(char* str);
  ~MyString();
  void concat(char* str);
  void print(ostream& os);
};

MyString::MyString() {  buf = 0; }

MyString::MyString(char* str) {
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}

void MyString::concat(char* str) {
  if(!buf) buf = new Mem;
  strcat((char*)buf->pointer(
    buf->msize() + strlen(str) + 1), str);
}

void MyString::print(ostream& os) {
  if(!buf) return;
  os << buf->pointer() << endl;
}

MyString::~MyString() { delete buf; }

int main() {
  MyString s("My test string");
  s.print(cout);
  s.concat(" some additional stuff");
  s.print(cout);
  MyString s2;
  s2.concat("Using default constructor");
  s2.print(cout);
} ///:~
```

Возможности тестового класса сводятся к созданию объекта MyString, конкатенации и выводу текста в поток ostream. Класс содержит указатель на объект Mem. Обратите внимание на отличия между конструктором по умолчанию, который обнуляет этот указатель, и вторым конструктором, который создает объект Mem и копирует в него данные. Конструктор по умолчанию удобен тем, что он, например, позволяет создать большой массив пустых объектов MyString с минимальными затратами, поскольку размер каждого объекта соответствует размеру указателя, а затраты на вызов конструктора по умолчанию сводятся к присваиванию нуля. Сколь-нибудь существенные затраты при работе с MyString возникают только при выполнении конкатенации; на этой стадии объект Mem создается, если он не был создан ранее. Даже если объект был создан конструктором по умолчанию, а в дальнейшем конкатенация не выполнялась, вызов деструктора все равно безопасен, поскольку вызов оператора delete для нуля определен таким образом, что он не пытается освобождать память и не создает других проблем.

На первый взгляд эти два конструктора хорошо подходят для определения аргументов по умолчанию. Но если удалить конструктор по умолчанию и, как показано ниже, записать оставшийся конструктор с аргументом по умолчанию, программа будет нормально работать, но весь выигрыш по эффективности окажется потерянным, потому что объект Mem будет создаваться всегда:

```
MyString(char* str = "");
```

Чтобы вернуть прежнюю эффективность, в конструктор придется внести изменения:

```
MyString::MyString(char* str) {
  if(!*str) { // Указатель на пустую строку
    buf = 0;
    return;
  }
  buf = new Mem(strlen(str) + 1);
  strcpy(char*)buf->pointer(), str);
```

Фактически передача значения по умолчанию превращается в условие, при нарушении которого выполняется отдельный фрагмент кода. Хотя при небольших размерах конструктора, как здесь, все выглядит достаточно безобидно, в общем случае такая практика создает проблемы. Вам приходится *проверять* значение по умолчанию, вместо того чтобы обрабатывать его наравне со всеми остальными значениями. Получается, что в одном теле объединяются две совершенно разные функции: для обычного аргумента и для значения по умолчанию. С таким же успехом функцию можно разбить на две функции и предоставить выбор компилятору. В результате достигается небольшое (точнее, незаметное) повышение эффективности, поскольку программе не приходится передавать дополнительный аргумент и выполнять лишний код условной проверки. Важнее другое: мы разделяем разнородный код на две разные функции, вместо того чтобы объединять его в одной функции с аргументом по умолчанию. Такой подход упрощает сопровождение программы, особенно при большом объеме функций.

С другой стороны, рассмотрим класс Mem. Если проанализировать определения двух конструкторов и двух функций pointer(), становится ясно, что введение аргументов по умолчанию в обоих случаях совершенно не изменяет определений функций. Следовательно, класс легко преобразуется к виду:

```
//: C07:Mem2.h
#ifndef MEM2_H
#define MEM2_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem(int sz = 0);
  ~Mem();
  int msize();
  byte* pointer(int minSize = 0);
};
#endif // MEM2_H ///:~
```

Вызов ensureMinSize() всегда будет достаточно эффективным.

Хотя в обоих рассмотренных случаях решение принималось с учетом эффективности, было бы неправильно думать только об эффективности и ни о чем больше. Основным фактором при проектировании класса является интерфейс (совокупность открытых членов класса, доступных для прикладного программиста). Если с классом удобно работать, значит, он хорошо спроектирован; при необходимости его всегда можно доработать, чтобы повысить эффективность. Но если класс плохо спроектирован из-за того, что программист был «зациклен» на эффективности, это может привести к самым серьезным последствиям. Главное, к чему нужно стремиться, — чтобы интерфейс был удобным и логичным для пользователей класса, а также для тех, кто будет читать ваш код. Стоит заметить, что в MemTest.cpp использование класса MyString не зависит от того, применяется ли конструктор по умолчанию или нет, а также от эффективности класса.

Итоги

Не используйте аргумент по умолчанию как условие, на основании которого выбирается одна из ветвей программы. Вместо этого постарайтесь разделить функцию на две и более перегруженные функции. Аргумент по умолчанию должен содержать обычное значение, которое передается в этой позиции при вызове. Это значение должно требоваться чаще остальных значений, чтобы прикладной программист мог проигнорировать его (в большинстве случаев) или задать другое значение, отличное от значения по умолчанию.

Аргументы по умолчанию должны упрощать вызов функций, особенно если эти функции вызываются с большим количеством аргументов, принимающих типичные значения. Аргументы по умолчанию упрощают не только запись, но и чтение вызовов, особенно если создателю класса удается упорядочить аргументы так, чтобы аргументы, особенно часто сохраняющие свои значения, находились на последних позициях списка.

Существует и другая, столь же важная ситуация, в которой применяются аргументы по умолчанию. Предположим, вы определили функцию с неким набором аргументов, а через некоторое время выясняется, что в функцию нужно добавить новые аргументы. Объявление для всех новых аргументов значений по умолчанию

гарантирует, что работоспособность клиентского кода, использующего прежний интерфейс, не будет нарушена.

Упражнения

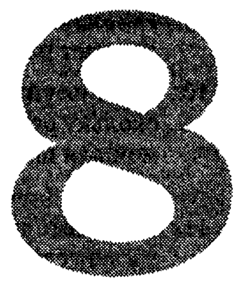
1. Создайте класс Text, содержащий объект string для хранения текста файла. Определите два конструктора: конструктор по умолчанию и конструктор, которому при вызове передается имя открываемого файла. Второй конструктор должен открывать файл и считывать его содержимое в переменную типа string. Включите в класс функцию contents(), которая бы возвращала string (например, для вывода). В функции main() откройте файл, используя объект Text, и выведите его содержимое.
2. Создайте класс Message с конструктором, получающим один аргумент типа string со значением по умолчанию. Определите закрытую переменную string; конструктор должен просто присваивать значение переданного аргумента внутренней переменной string. Создайте в классе Message две перегруженные функции print(): первая функция должна вызываться без аргументов и выводить сообщение, хранящееся в объекте, а другая функция должна получать аргумент string и выводить его содержимое вместе с сообщением из объекта. Насколько оправдан такой подход вместо того, в котором используются конструкторы?
3. Выясните, как генерировать ассемблерный код с помощью вашего компилятора, и, получив код, найдите в нем примеры украшения имен.
4. Создайте класс с четырьмя функциями, получающими соответственно 0, 1, 2 и 3 аргумента int. Напишите функцию main(), которая получает объект класса и вызывает каждую из его функций. Затем измените класс таким образом, чтобы он содержал одну функцию с четырьмя аргументами по умолчанию. Изменится ли при этом функция main()?
5. Напишите функцию с двумя аргументами и вызовите ее из main(). Затем преобразуйте один аргумент в «заполнитель» (то есть аргумент без идентификатора) и посмотрите, изменится ли при этом вызов в main().
6. Измените файлы Stash3.h и Stash3.cpp так, чтобы в конструкторе использовались аргументы по умолчанию. Протестируйте конструктор с двумя версиями объекта Stash.
7. Создайте новую версию класса Stack из главы 6 с конструктором по умолчанию и вторым конструктором, в аргументах которого передается массив указателей на объекты и размер этого массива. Конструктор должен перебирать элементы массива и заносить каждый указатель в стек. Протестируйте класс с массивом объектов string.
8. Измените класс SuperVar так, чтобы весь код vartype заключался в директивы #ifdef, как описано в соответствующем разделе. Преобразуйте тип vartype в обычное открытое перечисление (без экземпляра) и измените функцию print() так, чтобы ей передавался аргумент vartype, определяющий выполняемые действия.

9. Реализуйте пример Mem2.h и убедитесь в том, что измененный класс продолжает работать с примером MemTest.cpp.

10. Используйте класс Mem в реализации класса Stash. Обратите внимание: поскольку реализация объявлена закрытой, а следовательно, остается невидимой для прикладного программиста, изменять тестовую программу не придется.

11. Включите в класс Mem функцию bool moved(), которая получает результат вызова pointer() и сообщает, был ли перемещен указатель (вследствие перераспределения памяти). Напишите функцию main() для тестирования функции moved(). Что разумнее, использовать функции типа moved() или просто вызывать pointer() каждый раз, когда потребуется обратиться к памяти в Mem?

Константы



Концепция *констант* (определяемых с ключевым словом const) задумывалась для того, чтобы программист мог четко провести границу между переменными и неизменяемыми величинами. Такое разграничение повышает безопасность программирования на C++ и степень контроля операций, выполняемых в программе.

Изначально ключевое слово const предназначалось для нескольких целей, к тому же еще одно значение оно обрело после проникновения в язык С. На первый взгляд эти метаморфозы выглядят довольно запутанно, но в этой главе все прояснится: вы узнаете, когда, зачем и как должно применяться ключевое слово const. Глава завершается обсуждением ключевого слова volatile, близкого «родственника» const (оба ключевых слова связаны с возможностью изменения данных) с идентичным синтаксисом.

Вероятно, первым побудительным мотивом для введения ключевого слова const стало желание исключить подстановку значений препроцессорными директивами #define. С тех пор концепция константности была расширена для указателей, аргументов функций, возвращаемых значений, объектов и функций классов. Во всех перечисленных областях интерпретации константности слегка различаются, хотя и остаются совместимыми на концептуальном уровне. Они будут рассматриваться в разных разделах этой главы.

Подстановка значений

При программировании на языке С препроцессор широко используется для создания макросов и подстановки значений. Поскольку препроцессор выполняет простую замену текста, а для проверки типов в нем нет ни средств, ни концептуальной поддержки, подстановки часто порождают нетривиальные проблемы, которые предотвращаются в C++ при помощи ключевого слова const.

Типичный пример препроцессорной подстановки «имя/значение» в С выглядит так:

```
#define BUFSIZE 100
```

Имя BUFSIZE существует только во время препроцессорной обработки; следовательно, оно не занимает памяти и может храниться в заголовочном файле для определения единого значения во всех единицах трансляции, в которые этот файл включается. Использование подстановки вместо так называемых «волшебных чисел» играет очень важную роль при сопровождении программ. В случае «волшебных чисел» читатель программы не будет понимать, откуда взялось это число и что оно представляет, однако это еще не все. Если вы захотите изменить значение, вам придется редактировать программу вручную. Ничто не гарантирует, что вы случайно не забудете об одном из значений (или не измените то, которое изменять не следовало бы).

В большинстве случаев переменная BUFSIZE ведет себя, как обычная переменная, однако существуют и исключения. Кроме того, для нее не определен тип, поэтому препроцессорные подстановки порождают ошибки, которые очень трудно найти. В C++ эти проблемы устраняются благодаря ключевому слову const, которое передает подстановку значений в ведение компилятора. Теперь в программу можно включить строку:

```
const int bufsize = 100;
```

Имя bufsize может быть указано всюду, где значение должно быть известно компилятору на стадии компиляции. Компилятор может использовать bufsize для *подстановки констант*, то есть замены сложного константного выражения простым за счет выполнения необходимых вычислений во время компиляции. Это особенно важно при определении массивов:

```
char buf[bufsize];
```

Ключевое слово const может указываться со всеми встроенными типами (char, int, float и double) и их разновидностями (а также с объектами классов, как будет показано далее в этой главе). Как отмечалось ранее, препроцессорная подстановка может привести к появлению нетривиальных ошибок, поэтому в программах всегда следует использовать ключевое слово const вместо директивы подстановки #define.

Константы в заголовочных файлах

Чтобы использовать const вместо #define, необходимо иметь возможность включать определения const в заголовочные файлы, как определения #define. В этом случае определение константы оказывается в одном месте и распространяется по единицам трансляции посредством включения заголовочного файла. По умолчанию для констант C++ используется *внутреннее связывание*; иначе говоря, константы доступны только в том файле, в котором они определены, и во время компиляции не видны в других единицах трансляции. Значение константы всегда должно задаваться при определении, но это требование *не относится* к объявлению внешних констант с ключевым словом extern:

```
extern const int bufsize;
```

Обычно компилятор C++ не выделяет память для констант, а хранит определение в своей таблице имен. Однако ключевое слово extern в объявлении константы заставляет компилятор выделить память (впрочем, это происходит и в ряде других случаев, например при получении адреса константы в программе). Дело в том,

что ключевое слово extern фактически означает «задействовать внешнюю компоновку», то есть данные должны быть доступны в нескольких единицах трансляции, а для этого для них необходимо выделить память.

В рядовых ситуациях, когда определение не содержит ключевого слова extern, память не выделяется, а константные выражения просто вычисляются во время компиляции.

Политика отказа от выделения памяти также не срабатывает при определении сложных структур данных. Обязательное выделение памяти означает, что подстановка констант невозможна (потому что компилятор не может узнать, какие данные хранятся в памяти, — если бы он это знал, то выделять память не пришлось бы).

Так как в отдельных случаях компилятору приходится выделять память для констант, по умолчанию для определений констант *должно* использоваться внутреннее связывание (то есть связывание только *внутри* текущей единицы трансляции). В противном случае определение сложных констант приводило бы к ошибкам компоновки, поскольку память для них выделялась бы в нескольких cpp-файлах. Компоновщик находит одинаковые определения в нескольких объектных файлах и выдает сообщение об ошибке. Но так как для констант связывание по умолчанию является внутренним, компоновщик не пытается связывать эти определения между единицами трансляции, поэтому конфликтов не возникает. Для встроенных типов, которые чаще используются в константных выражениях, компилятор всегда может выполнить подстановку.

Константы и защита данных

Область применения ключевого слова const не ограничивается заменой директивы #define. Если переменная инициализируется значением, полученным во время выполнения программы, и вы точно знаете, что ее значение не изменится на протяжении всего жизненного цикла, хороший стиль программирования требует объявить переменную с ключевым словом const, чтобы компилятор выдавал сообщение об ошибке при непреднамеренных изменениях этой переменной. Пример:

```
//: C08:Safecons.cpp
// Использование const для защиты данных
#include <iostream>
using namespace std;

const int i = 100;  // Типичная константа
const int j = i + 10; // Значение константного выражения
long address = (long)&j; // Требует выделения памяти
char buf[j + 10]; // Также константное выражение

int main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // Изменение невозможно
  const char c2 = c + 'a';
  cout << c2;
  // ...
} ///:~
```

Вы видите, что i является константой времени компиляции, но значение j вычисляется на основании i. Вследствие константности i вычисленное значение j определяется константным выражением и само по себе является константой времени

компиляции. Следующая строка, в которой запрашивается адрес j, заставляет компилятор выделить память для j. Однако это не мешает задействовать j для определения размера buf — компилятор знает, что j является константой, поэтому значение может использоваться, несмотря даже на то, что где-то в программе была выделена память для хранения этой величины.

В функции main() идентификатор c представляет другую разновидность констант, значение которых в принципе не может быть определено на стадии компиляции. Это означает, что компилятору неизбежно придется выделить память, поэтому компилятор не пытается хранить что-либо в своей таблице имен (то же самое происходит в C). Инициализация должна выполняться в точке определения, и после инициализации значение остается неизменным. Из листинга видно, что c используется для вычисления c2, и механизм видимости работает для констант точно так же, как для всех остальных типов, — еще одно усовершенствование по сравнению с #define.

На практике действует простое правило: если вы считаете, что значение не должно изменяться, объявите его константным. Тем самым вы не только застрахуетесь от случайных изменений, но и позволите компилятору сгенерировать более эффективный код за счет отказа от выделения памяти и чтения из нее.

Агрегаты

Ключевое слово const может применяться к агрегатным структурам данных, однако компилятору почти наверняка не хватит интеллекта для сохранения агрегата в таблице имен, поэтому компилятор выделит для него память. В таких ситуациях const означает «область памяти с неизменным содержимым». Тем не менее это значение не может использоваться во время компиляции, поскольку компилятор может не знать содержимого этой памяти. В следующем фрагменте недопустимые команды выделены:

```
//: C08:Constag.cpp
// Константы и агрегаты
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Недопустимо
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Недопустимо
int main() {} ///:~
```

При определении массива компилятор должен сгенерировать код, перемещающий указатель стека для хранения массива. В обоих недопустимых определениях из предыдущего фрагмента компилятор не может найти константное выражение в определении массива.

Отличия от языка C

Константы появились в ранних версиях C++, в то время когда работа над спецификацией стандарта C еще продолжалась. Хотя комитет по стандартизации C решил включить поддержку ключевого слова const в язык C, под этим ключевым словом почему-то понималась «обычная переменная, значение которой не может изменяться». В языке C константа всегда занимает память, а ее имя является глобальным. Компилятор C не позволяет интерпретировать const как константу

времени компиляции. Например, в C следующий фрагмент выдает ошибку компиляции, хотя на первый взгляд он выглядит вполне разумно:

```
const int bufsize = 100;
char buf[bufsize];
```

Поскольку значение bufsize хранится где-то в памяти, компилятор C считает, что оно неизвестно на стадии компиляции. Кроме того, в C (но не в C++) разрешены конструкции вида

```
const int bufsize;
```

Компилятор C воспринимает эту строку как объявление, которое указывает, что память выделяется где-то в другом месте. Поскольку в C для констант по умолчанию используется внешнее связывание, это вполне логично. В C++ для констант по умолчанию используется внутреннее связывание, поэтому, чтобы добиться той же цели в C++, необходимо явно переключиться на внешнее связывание ключевым словом extern:

```
extern const int bufsize; // Только объявление
```

Эта строка работает также и в C.

В C++ ключевое слово const не всегда приводит к выделению памяти, тогда как в C память выделяется автоматически. Выделение памяти в C++ зависит от того, как задействована данная константа. В общем случае, если const используется просто для замены имени значением (как с директивами #define), то выделять память не обязательно. А если память не выделяется (это зависит от сложности типа данных и степени совершенства компилятора), становится возможной подстановка значений в код для повышения эффективности, но это происходит *после* проверки типов, а не *до* нее, как в случае с #define. Однако если программа получает адрес константы (даже неявно, посредством передачи ее функции со ссылочным аргументом) или константа определяется как внешняя (с ключевым словом extern), то для константы выделяется память.

В C++ константы, не принадлежащие ни к одной из функций, имеют файловую видимость (то есть остаются невидимыми за пределами файла). Следовательно, по умолчанию для них используется внутреннее связывание. В этом отношении константы принципиально отличаются от других идентификаторов C++ (и констант языка C!), для которых по умолчанию используется внешнее связывание. Итак, если объявить одноименные константы в двух разных файлах, не получать их адреса и не определять их имена с ключевым словом extern, идеальный компилятор C++ не будет выделять память для констант, а просто подставит их в программу. Поскольку для констант подразумевается файловая видимость, их включение в заголовочные файлы C++ не создает конфликтов на стадии компоновки.

Так как для констант C++ по умолчанию используется внутреннее связывание, вы не сможете определить константу в одном файле и сослаться на нее с ключевым словом extern из другого файла. Если потребуется назначить константе внешнее связывание, чтобы на нее можно было ссылаться из другого файла, ее необходимо явно определить с ключевым словом extern:

```
extern const int x = 1;
```

Обратите внимание: наличие инициализирующего значения и объявление константы extern заставляет компилятор выделить для нее память (хотя в данном случае компилятор может выполнить подстановку константы). Вследствие инициа-

лизации эта строка представляет собой определение, а не объявление. Следующее объявление в C++ означает, что определение существует где-то в другом месте (стоит напомнить, что в C это не обязательно):

```
extern const int x;
```

Теперь вы понимаете, почему C++ требует включения инициализирующего значения в определение константы, — это значение отличает объявление от определения (в C это всегда определение, поэтому инициализирующее значение не нужно). Встретив объявление extern const, компилятор не может выполнить подстановку константы, потому что не знает ее значения.

Интерпретация констант, принятая в C, особой пользы не приносит. Если именованное значение должно использоваться в константном выражении (то есть должно быть известно во время компиляции), программист C практически *вынужден* применять директиву #define с препроцессором.

Указатели

Указатели тоже могут объявляться с ключевым словом const. В этом случае компилятор также старается обойтись без выделения памяти и ограничиться подстановкой, хотя для указателей подстановка особой пользы не приносит. Важнее другое: компилятор сообщает о попытках модификации таких указателей, что заметно повышает надежность программы.

Существуют два варианта использования констант с указателями: ключевое слово const может применяться к объекту, на который ссылается указатель, или же к адресу, хранящемуся в самом указателе. Двойной синтаксис поначалу кажется излишне запутанным, но постепенно к нему привыкаешь.

Указатель на константу

Анализ определений указателей, как и любых сложных определений, лучше начинать с идентификатора и постепенно двигаться наружу. Спецификатор const ассоциируется с *ближайшим* элементом. Следовательно, если вы хотите запретить модификацию того элемента, на который ссылается указатель, запишите определение в следующем виде:

```
const int* u;
```

Начиная с идентификатора, мы читаем: «u — указатель, который ссылается на const int». Инициализация в данном случае не обязательна; u может указывать на что угодно (то есть сам указатель не является константой), но то, *на что* он указывает, изменяться не может.

А вот разобраться в дальнейшем будет посложнее. Напрашивается предположение, что для того, чтобы сделать неизменным сам указатель (то есть запретить любые изменения адреса, содержащегося в u), достаточно переместить const на другую сторону int:

```
int const* v;
```

Логично предположить, что это определение должно читаться как «v является константным указателем на int». Но *на самом деле* следует читать «v является обычным указателем на объект int, который является константным». Другими словами,

const снова привязывается к int с теми же последствиями, что и в предыдущем определении. Идентичность этих двух определений способна вызвать недоразумения, и, чтобы не сбивать с толку читателя программы, лучше ограничиться первой формой.

Константный указатель

Чтобы запретить модификацию самого указателя, необходимо разместить спецификатор const справа от знака *:

```
int d = 1;
int* const w = &d;
```

Теперь определение читается так: «w является константным указателем, который ссылается на int». Так как const теперь относится к указателю, компилятор требует, чтобы ему было присвоено начальное значение, неизменное на протяжении жизненного цикла указателя. С другой стороны, ничто не мешает изменять данные, на которые указатель ссылается:

```
*w = 2;
```

Также можно создать константный указатель на константный объект, для чего существуют две синтаксические формы:

```
int d = 1;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
```

Эти определения запрещают изменение как самого указателя, так и объекта.

Некоторые программисты считают вторую форму более последовательной, потому что ключевое слово const всегда находится справа от того, что оно модифицирует. Сами решите, какой из вариантов лучше подходит для вашего стиля программирования.

Пример компилируемого файла, содержащего предыдущий фрагмент:

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;
int d = 1;
int* const w = &d;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
int main() {} ///:~
```

Формат определений

В примерах этой книги определения указателей размещаются по одному на строку, причем указатели по возможности инициализируются в точке определения. Это позволяет использовать стиль форматирования, при котором знак * «прикрепляется» к типу данных:

```
int* u = &i;
```

Определение выглядит так, словно int* является отдельным типом. Такая запись упрощает понимание программы, но, к сожалению, это всего лишь иллюзия. На самом деле знак * связывается с идентификатором, а не с типом; просто он может находиться где угодно между именем типа и идентификатором. То есть вполне допустимы такие конструкции:

```
int *u = &i, v = 0;
```

Эта строка создает указатель int* u и обычную переменную (не указатель) int v. При чтении программы часто возникает путаница, поэтому лучше придерживаться формы, представленной в книге.

Присваивание и проверка типов

В языке C++ тщательно проверяются типы, и эта проверка распространяется на присваивание указателей. Адрес неконстантного объекта можно присвоить указателю на константу, потому что в этом случае вы просто обещаете не изменять величину, которая в принципе может изменяться. С другой стороны, адрес константного объекта нельзя присвоить указателю на «не-константу», потому что объект может быть изменен через указатель. Конечно, такое присваивание всегда можно выполнить принудительно путем приведения типа, но это считается проявлением плохого стиля программирования, потому что вы сознательно нарушаете константность объекта и отказываетесь от мер безопасности, предоставляемых ключевым словом const. Пример:

```
//: C08:PointerAssignment.cpp
int d = 1;
const int e = 2;
int* u = &d; // Можно -- d не является константой
//! int* v = &e; // Нельзя -- е является константой
int* w = (int*)&e; // Можно, но это плохой стиль
int main() {} ///:~
```

Язык C++ помогает предотвращать ошибки, но не защитит вас от самого себя, если вы сознательно хотите разрушить механизмы безопасности.

Константность символьных массивов

Жесткие правила константности несколько смягчаются при работе с литералами, которые представляют собой символьные массивы. Например, компилятор без возражений принимает следующее определение:

```
char* cp = "howdy";
```

С формальных позиций такое определение ошибочно, потому что литерал ("howdy" в данном случае) создается компилятором как константный символьный массив, а «результатом» символьного массива, заключенного в кавычки, является его начальный адрес в памяти. Модификация символов массива считается ошибкой времени выполнения, хотя не все компиляторы должным образом следят за выполнением этого правила.

Итак, символьные массивы-литералы в действительности являются константными символьными массивами. Конечно, компилятор допускает их неконстантную интерпретацию, потому что от этого зависит работа многих существующих программ С. Тем не менее попытка изменения значений в символьном массиве-литерале приводит к непредсказуемым последствиям, хотя, скорее всего, на многих компьютерах она успешно сработает.

Если вы хотите сохранить возможность модификации строки, поместите ее в массив:

```
char cp[] = "howdy";
```

Поскольку компиляторы часто не различают эти два определения, они не напоминают о необходимости использовать вторую форму, поэтому в программах могут появиться весьма нетривиальные ошибки.

Аргументы функций и возвращаемые значения

Применение ключевого слова const к аргументам функций и возвращаемым значениям — еще одна область, в которой концепция константности часто понимается неправильно. Если объекты передаются *по значению*, то ключевое слово const не имеет смысла с точки зрения клиента (оно означает, что переданный аргумент не может быть модифицирован внутри функции). Если функция возвращает объект пользовательского типа по значению, const указывает на то, что возвращаемое значение не может быть модифицировано. При передаче и возвращении *адресов* const обещает, что объект, находящийся по указанному адресу, не будет изменяться.

Передача констант по значению

Аргументы функции, передаваемые по значению, могут быть объявлены константными:

```
void f1(const int i) {
  i++; // Недопустимо -- ошибка компиляции
}
```

Однако что это означает на практике? Вы обещаете, что исходное значение переменной не будет изменяться функцией f1(). Но при передаче аргумента по значению исходная переменная копируется, поэтому данное клиенту обещание выполняется автоматически.

Внутри функции const означает запрет на изменение аргумента, но эта информация предназначена скорее для создателя функции, а не для вызывающей стороны.

Чтобы не путать прикладных программистов, аргумент можно объявить константным не в списке аргументов, а *внутри* функции. Задача решается при помощи указателя, хотя существует более изящный синтаксис, основанный на использовании *ссылок* (эта тема будет подробно рассматриваться в главе 11). Пока достаточно сказать, что ссылка напоминает константный указатель, который автоматически разыменовывается; таким образом, ссылка фактически определяет псевдоним для работы с объектом. Чтобы определить ссылку для объекта, следует включить в определение символ &. То есть более понятное определение функции выглядит так:

```
void f2(int ic) {
  const int& i = ic;
  i++;  // Недопустимо -- ошибка компиляции
}
```

В этом случае тоже будет выдано сообщение об ошибке, но на этот раз константность локального объекта не является частью сигнатуры функции; она имеет смысл только в реализации функции и полностью скрывается от клиента.

Константное возвращаемое значение

Все сказанное выше относится и к возвращаемым значениям. Объявляя возвращаемое значение функции константным, как показано ниже, вы гарантируете, что исходная переменная (находящаяся в теле функции) не будет изменяться:

```
const int g();
```

Как и прежде, возвращаемая по значению величина копируется, поэтому неизменность оригинала возвращаемого значения достигается автоматически.

На первый взгляд это делает объявление const бессмысленным. Следующий пример демонстрирует внешнюю бесполезность возвращения констант:

```
//: C08:Constval.cpp
// Возвращение констант по значению
// бессмысленно для встроенных типов

int f3() { return 1; }
const int f4() { return 1; }

int main() {
  const int j = f3(); // Работает нормально
  int k = f4(); // Но и это тоже работает!
} ///:~
```

Для встроенных типов константность возвращаемого значения ни на что не влияет, поэтому лучше не путать прикладных программистов и не указывать ключевое слово const при возвращении встроенного типа по значению.

Однако возвращение констант по значению становится существенным при работе с пользовательскими типами. Если функция возвращает по значению объект класса с модификатором const, то возвращаемое значение функции не может использоваться в качестве l-значения (то есть ему нельзя присвоить что-либо или изменить иным образом). Пример:

```
//: C08:ConstReturnValues.cpp
// Возвращение константы по значению
// Результат не может использоваться в качестве l-значения

class X {
  int i;
public:
  X(int ii = 0);
  void modify();
};

X::X(int ii) { i = ii; }

void X::modify() { i++; }

X f5() {
  return X();
}

const X f6() {
  return X();
}

void f7(X& x) { // Передача по неконстантной ссылке
```

```
    x.modify();
}

int main() {
  f5() = X(1); // Можно -- неконстантное возвращаемое значение
  f5().modify(); // Можно
//!  f7(f5()); // Предупреждение или ошибка
// Ошибки компиляции:
//!  f6() = X(1);
//!  f6().modify();
//!  f7(f6());
} ///:~
```

Вызов f5() возвращает неконстантный объект X, тогда как вызов f6() возвращает константный объект X. Только неконстантные возвращаемые значения могут использоваться в качестве l-значений. Следовательно, при возвращении объекта по значению следует указывать ключевое слово const, если вы хотите предотвратить использование этого объекта в качестве l-значения.

Бессмысленность ключевого слова const для встроенных типов, возвращаемых по значению, объясняется тем, что компилятор автоматически предотвращает их использование в качестве l-значений (потому что возвращается всегда значение, а не переменная). Различия возникают только при возвращении по значению объекта пользовательского типа.

Функция f7() получает свой аргумент в виде *ссылки* (этому дополнительному механизму работы с адресами в C++ посвящена глава 11). Фактически происходит то же, что при передаче указателя на неконстантную величину; отличается только синтаксис. Ошибка компиляции в C++ происходит из-за создания временного объекта.

Временные объекты

Иногда в процессе вычисления выражения компилятору приходится создавать *временные объекты*. Эти объекты принципиально не отличаются от других объектов: они также требуют выделения памяти, конструируются и уничтожаются. Отличие состоит в том, что эти объекты остаются невидимыми для программиста, — компилятор сам решает, когда они нужны, и определяет все аспекты их существования. Но у временных объектов есть одна важная особенность: они автоматически являются константными. Поскольку обычно вы не можете получить доступ к временному объекту, выполнение каких-либо операций, изменяющих состояние временного объекта, почти наверняка является ошибкой, потому что эту информацию все равно не удастся использовать. Автоматически объявляя все временные объекты константными, компилятор сможет сообщить вам об этой ошибке.

В предыдущем примере вызов f5() возвращает неконстантный объект X. Однако в следующем выражении компилятор должен создать временный объект для хранения возвращаемого значения f5(), чтобы передать его f7():

```
f7(f5());
```

Если бы функция f7() получала свой аргумент по значению, все было бы хорошо, поскольку временный объект копируется в f7() и при этом совершенно не важно, что произойдет с временным объектом X. Однако f7() получает аргумент *по ссылке*, а это в данном случае означает получение адреса временного объекта X. Поскольку f7() не получает объекта по ссылке на константу, получается, что функ-

ция сможет изменять временный объект. Но компилятор знает, что временный объект прекратит существование сразу же после того, как вычисление выражения будет завершено, поэтому все изменения во временном объекте X окажутся потерянными. Автоматическое объявление всех временных объектов константными приводит к выдаче сообщения на стадии компиляции, чтобы вы не пропустили ошибку, которую будет очень трудно найти.

Тем не менее учтите, что следующие выражения допустимы:

```
f5() = X(1);
f5().modify();
```

Хотя эти выражения проходят проверку компилятора, на самом деле они создают проблемы. Функция f5() возвращает объект X; чтобы компилятор мог выполнить заданные команды, он должен создать временный объект для хранения возвращаемого значения. В обоих выражениях временный объект модифицируется, и сразу же после завершения выражения он будет уничтожен. В результате все изменения теряются, а этот код, вероятно, является ошибочным, но компилятор вам об этом ничего не скажет. Приведенные выражения достаточно просты, чтобы программист мог обнаружить проблему самостоятельно, но в более сложных случаях в программу может закрасться ошибка.

Передача и возвращение адресов

Если функция передает или возвращает адрес (указатель или ссылку), прикладной программист может использовать этот адрес для модификации исходного значения. Если объявить указатель или ссылку с ключевым словом const, такая модификация станет невозможной, а вы избавитесь от лишних хлопот. При передаче адресов всегда следует объявлять их константными, если это возможно.

Выбор между возвращением указателя и возвращением ссылки на const зависит от того, какие операции с возвращаемым значением вы собираетесь разрешить прикладному программисту. Следующий пример демонстрирует использование указателей на константы в качестве аргументов и возвращаемых значений:

```
//: C08:ConstPointer.cpp
// Передача указателей на константы
// в аргументах и возвращаемых значениях

void t(int*) {}

void u(const int* cip) {
//!  *cip = 2; // Нельзя -- модификация значения
  int i = *cip; // Можно -- копирование значения
//!  int* ip2 = cip; // Нельзя -- не константа
}

const char* v() {
  // Возвращение адреса статического символьного массива:
  return "result of function v()";
}

const int* const w() {
  static int i;
  return &i;
}
```

```
int main() {
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip);  // Можно
//! t(cip); // Нельзя
  u(ip);  // Можно
  u(cip); // Тоже можно
//! char* cp = v(); // Нельзя
  const char* ccp = v(); // Можно
//! int* ip2 = w(); // Нельзя
  const int* const ccip = w(); // Можно
  const int* cip2 = w(); // Можно
//! *w() = 1; // Нельзя
} ///:~
```

Функции t() в аргументе передается обычный указатель, а функция u() получает указатель на константу. Внутри функции u() попытки изменить данные, на которые ссылается указатель, недопустимы, но эту информацию, разумеется, можно скопировать в неконстантную переменную. Компилятор также не позволяет создать обычный указатель по адресу, хранящемуся в указателе на константу.

Функции v() и w() проверяют семантику возвращаемого значения. Функция v() возвращает объект const char*, созданный на основе символьного массива-литерала. В действительности эта функция возвращает адрес символьного массива-литерала после того, как компилятор создаст его и сохранит в статической области памяти. Как упоминалось ранее, с технической точки зрения этот символьный массив является константой, на что указывает возвращаемое значение v().

Возвращаемое значение w() требует, чтобы константным был как сам указатель, так и те данные, на которые он указывает. Как и в предыдущем случае, значение, возвращаемое функцией w(), остается действительным после выхода из функции только потому, что оно объявлено статическим. Никогда не возвращайте указатели на локальные стековые переменные, потому что они становятся недействительными после возвращения из функции и очистки стека (на практике также часто возвращается адрес области памяти, выделенной из кучи, который остается действительным после вызова).

В main() функции тестируются с разными аргументами. Как видно из листинга, функция t() принимает обычный указатель, но если попытаться передать ей указатель на константу, ничто не гарантирует, что вызов t() оставит в неприкосновенности данные, на которые ссылается указатель, поэтому компилятор выдает сообщение об ошибке. Функция u() получает указатель на константу, поэтому она принимает оба типа аргументов. Таким образом, функция, получающая указатель на константу, оказывается более универсальной.

Как следовало ожидать, возвращаемое значение v() может присваиваться только указателю на const. Также можно было бы ожидать, что компилятор откажется присвоить обычному указателю возвращаемое значение w() и примет const int* const, но, как ни странно, компилятор также принимает значение const int*, не полностью соответствующее возвращаемому типу. Как и прежде, значение (адрес, содержащийся в указателе) копируется, поэтому гарантии неизменности исходной переменной предоставляются автоматически. То есть второе ключевое слово const в const int* const имеет смысл только в том случае, если вы попытаетесь использовать переменную в качестве l-значения, но это не позволит компилятор.

Стандартная передача аргументов

В языке С передача аргументов по значению применяется очень часто, а если вдруг потребуется передать адрес объекта, приходится использовать указатели[1]. В С++ ни один из этих вариантов не является основным. На первом месте стоит передача аргументов по ссылке, в том числе и ссылке на const. С точки зрения прикладного программиста, синтаксис такой же, как при передаче по значению, причем указатели не усложняют его — прикладному программисту вообще не приходится думать об указателях. Для создателя функции передача адреса почти всегда более эффективна, чем передача всего объекта класса, а передача по ссылке на const означает, что функция не сможет изменить данные, ассоциированные с указанным адресом. Для прикладного программиста все происходит точно так же, как при передаче по значению (только более эффективно).

Ссылочный синтаксис (который для вызывающей стороны практически неотличим от передачи по значению) позволяет передать временный объект функции, получающей ссылку на const, тогда как функции, получающей указатель, передать временный объект невозможно — указатель требует явного получения адреса. Таким образом, передача по ссылке открывает новую возможность, недоступную в С: функции можно передать *адрес* временного объекта, который всегда является константным. Вот почему для передачи временных объектов по ссылке аргумент должен быть ссылкой на const. Следующий пример поясняет сказанное:

```
//: C08:ConstTemporary.cpp
// Временные объекты являются константными

class X {};

X f() { return X(); } // Возвращение по значению

void g1(X&) {} // Передача по обычной ссылке
void g2(const X&) {} // Передача по ссылке на константу

int main() {
  // Ошибка: f() создает константный временный объект:
//!  g1(f());
  // Можно: g2 получает ссылку на константу:
  g2(f());
} ///:~
```

Функция f() возвращает объект класса X *по значению*. Следовательно, если мы берем возвращаемое значение f() и передаем его другой функции (как при вызовах g1() и g2()), это приводит к созданию временного объекта, который является константным. Значит, вызов g1() является ошибкой, потому что g1() не получает ссылки на константу, но с вызовом g2() все нормально.

Классы

В этом разделе представлены различные варианты использования ключевого слова const с классами. Возможно, вы хотите создать в классе локальную константу,

[1] Некоторые программисты полагают, что в С *всегда* используется передача по значению, поскольку при передаче указателя тоже происходит копирование (то есть указатель передается по значению). Возможно, формально они правы, но подобные рассуждения только запутывают вопрос.

которая должна задействоваться в константных выражениях, вычисляемых на стадии компиляции. Тем не менее в классах смысл const меняется, поэтому для создания константных членов необходимо хорошо понимать все аспекты константности.

Константным также можно объявить весь объект (как вы уже видели, компилятор всегда объявляет временные объекты константными). Тем не менее сохранение константности объекта является достаточно сложной задачей. Компилятор способен обеспечить константность встроенных типов, но не может отслеживать все тонкости работы класса. Для поддержания константности объектов класса вводятся константные функции классов: для константных объектов могут вызываться только константные функции.

Ключевое слово const в классах

Использование констант в константных выражениях было бы уместно и внутри классов. Типичный пример: допустим, вы определяете массив внутри класса и хотите использовать ключевое слово const вместо директивы #define для установления размеров массива и при вычислениях с массивом. Размер массива относится к числу тех данных, которые должны скрываться в классе, чтобы связанное с ним имя (например, size) могло использоваться в других классах без конфликтов. Препроцессор интерпретирует все директивы #define как глобальные, начиная с точки определения, поэтому такое решение не дает желаемого эффекта.

Напрашивается предположение, что для этого нужно включить ключевое слово const в класс, однако и это не приводит к нужному результату. Внутри класса ключевое слово const частично возвращается к своей интерпретации в языке C: оно выделяет память в каждом объекте и представляет значение, которое инициализируется один раз и в дальнейшем не может изменяться. Присутствие const внутри класса означает «константность на протяжении срока жизни объекта». Тем не менее в разных объектах могут содержаться разные значения этой константы.

Таким образом, при создании обычной (не статической) константы внутри класса ей нельзя присвоить начальное значение. Конечно, инициализация выполняется в конструкторе, но не в любом месте, а в особой точке. Поскольку константа должна инициализироваться в момент создания, при выполнении тела конструктора она должна быть *уже* инициализирована. В противном случае осталась бы возможность отложить инициализацию до более поздней стадии выполнения конструктора, а это означало бы, что константа остается неинициализированной в течение некоторого времени. Кроме того, ничто не помешало бы изменять значение константы в теле конструктора.

Списки инициализирующих значений конструктора

Особая точка инициализации, о которой упоминалось в предыдущем разделе, называется *списком инициализирующих значений*. Первоначально списки инициализирующих значений задумывались для поддержания наследования (глава 14). Список инициализирующих значений конструктора (который, как подсказывает само название, присутствует только в определении конструктора) представляет собой перечень «вызовов конструкторов», находящийся после списка аргументов и двоеточия, но перед открывающей фигурной скобкой тела конструктора. Такое расположение напоминает, что инициализация выполняется перед выполнением основного кода конструктора. Именно в этом месте инициализируются все

константы. Следующий пример показывает, как объявляются константы внутри класса:

```
//: C08:ConstInitialization.cpp
// Инициализация констант в классах
#include <iostream>
using namespace std;

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
} ///:~
```

Приведенная выше форма списка инициализирующих значений в конструкторе выглядит несколько странно – непривычно видеть, как встроенный тип в ней интерпретируется так, словно у него есть конструктор.

«Конструкторы» встроенных типов

По мере развития языка значительные усилия направлялись на то, чтобы пользовательские типы вели себя как встроенные. Но потом стало очевидно, что в некоторых ситуациях было бы полезно обратное: чтобы встроенные типы вели себя как пользовательские. В частности, в списке инициализирующих значений конструктора встроенный тип может интерпретироваться так, как будто у него есть конструктор:

```
//: C08:BuiltInTypeConstructors.cpp
#include <iostream>
using namespace std;

class B {
  int i;
public:
  B(int ii);
  void print();
};

B::B(int ii) : i(ii) {}
void B::print() { cout << i << endl; }

int main() {
  B a(1), b(2);
  float pi(3.14159);
  a.print(); b.print();
  cout << pi << endl;
} ///:~
```

Данная возможность особенно важна при инициализации константных переменных, потому что они должны быть инициализированы перед входом в тело функции.

Возникла логичная идея — обобщить концепцию «конструктора» встроенных типов (для которых конструирование сводится к обычному присваиванию). Именно благодаря этому обобщению в предыдущем фрагменте работает определение

```
float pi(3.14159);
```

Часто бывает удобно инкапсулировать встроенный тип в классе, чтобы гарантировать его инициализацию в конструкторе. Для примера рассмотрим класс **Integer**:

```
//: C08:EncapsulatingTypes.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii = 0);
  void print();
};

Integer::Integer(int ii) : i(ii) {}
void Integer::print() { cout << i << ' '; }

int main() {
  Integer i[100];
  for(int j = 0; j < 100; j++)
    i[j].print();
} ///:~
```

Массив объектов **Integer** в **main()** автоматически инициализируется нулями, причем такая инициализация не обязательно обходится дороже цикла **for** или вызова **memset()**. Многие компиляторы легко оптимизируют ее, поэтому инициализация выполняется очень быстро.

Константы времени компиляции в классах

Каким бы интересным (а иногда и полезным) ни было применение ключевого слова **const**, о котором рассказывается в предыдущем разделе, оно не дает ответа на исходный вопрос: «Как создать константу времени компиляции в классе?» Для решения этой задачи необходимо дополнительное ключевое слово **static**, полное описание которого откладывается до главы 10. В данной ситуации ключевое слово **static** означает «существующий только в одном экземпляре, независимо от количества созданных объектов класса», то есть именно то, что нам требуется: константная переменная класса, значение которой остается постоянным для всех объектов класса. Таким образом, статическая константа (то есть переменная встроенного типа, объявленная как **static const**) может рассматриваться как константа времени компиляции.

У статических констант в классах есть одна отличительная особенность: инициализация должна выполняться в точке определения. Это относится только статическим константам; как бы вам ни хотелось использовать инициализацию в точке определения в других ситуациях, она не будет работать, потому что остальные переменные должны инициализироваться конструктором или другими функциями класса.

Следующий пример демонстрирует создание и использование статической константы с именем size в классе, представляющем стек указателей на строки[1]:

```
//: C08:StringStack.cpp
// Использование static const для создания
// констант времени компиляции в классе
#include <string>
#include <iostream>
using namespace std;

class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
  StringStack();
  void push(const string* s);
  const string* pop();
};

StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(string*));
}

void StringStack::push(const string* s) {
  if(index < size)
    stack[index++] = s;
}

const string* StringStack::pop() {
  if(index > 0) {
    const string* rv = stack[--index];
    stack[index] = 0;
    return rv;
  }
  return 0;
}

string iceCream[] = {
  "pralines & cream",
  "fudge ripple",
  "jamocha almond fudge",
  "wild mountain blackberry",
  "raspberry sorbet",
  "lemon swirl",
  "rocky road",
  "deep chocolate fudge"
};

const int iCsz =
  sizeof iceCream / sizeof *iceCream;

int main() {
  StringStack ss;
  for(int i = 0; i < iCsz; i++)
    ss.push(&iceCream[i]);
  const string* cp;
```

[1] На момент написания книги такие стеки не поддерживались некоторыми компиляторами.

```
  while((cp = ss.pop()) != 0)
    cout << *cp << endl;
} ///:~
```

Поскольку size используется для определения размера массива stack, эта величина действительно является константой времени компиляции, но константой, определенной внутри класса.

Обратите внимание: функция push() получает const string*, функция pop() возвращает const string*, а в классе StringStack хранится массив const string*. Если бы эти условия не выполнялись, класс StringStack не мог бы использоваться для хранения указателей на элементы iceCream. С другой стороны, тем самым предотвращаются любые действия, которые могли бы привести к изменению объектов, хранящихся в StringStack. Конечно, не все контейнеры проектируются с этим ограничением.

«Финт с перечислением»

В первых версиях C++ конструкция static const в классах не поддерживалась. То есть ключевое слово const не могло использоваться в константных выражениях внутри классов. Однако такая потребность все же существовала, поэтому появилось типовое решение, основанное на применении анонимного перечисляемого типа без экземпляров. Для этого решения обычно использовался жаргонный термин «финт с перечислением» (enum hack). Значения элементов перечисления известны на стадии компиляции; перечисление является локальным для класса, а его элементы могут использоваться в константных выражениях. Так что на практике часто встречались решения вроде следующего:

```
//: C08:EnumHack.cpp
#include <iostream>
using namespace std;

class Bunch {
  enum { size = 1000 };
  int i[size];
};

int main() {
  cout << "sizeof(Bunch) = " << sizeof(Bunch)
       << ", sizeof(i[1000]) = "
       << sizeof(int[1000]) << endl;
} ///:~
```

Использование перечисления заведомо не увеличивает объем памяти на хранение объекта, а все значения элементов автоматически вычисляются во время компиляции или задаются явно:

```
enum { one = 1, two = 2, three };
```

Компилятор продолжает отсчет целых чисел после последней заданной величины, поэтому элементу three будет присвоено значение 3.

Обратите внимание на следующую строку из файла StringStack.cpp:

```
static const int size = 100;
```

В приведенном примере эта строка заменяется строкой

```
enum { size = 100 };
```

Прием, основанный на использовании перечисления, часто встречается в старых программах. Конструкция static const была включена в язык именно для реше-

ния проблемы. Тем не менее не существует никаких веских причин, *обязывающих* вас задействовать конструкцию static const вместо «финта с перечислением». Далее в книге будет использоваться именно этот «финт», потому что на момент ее написания он поддерживался большим числом компиляторов.

Константные объекты и функции классов

Функции классов тоже могут объявляться с ключевым словом const. Что это значит? Чтобы ответить на этот вопрос, необходимо понять концепцию константных объектов.

Константные объекты пользовательских типов определяются так же, как константные объекты встроенных типов. Пример:

```
const int i = 1;
const blob b(2);
```

Здесь b является константным объектом типа blob. При вызове его конструктора передается аргумент 2. Чтобы компилятор мог обеспечить соблюдение константности, он должен быть уверен в том, что переменные объекта не изменяются на протяжении жизненного цикла объекта. Конечно, проверить неизменность открытых данных несложно, но как узнать, какие функции класса изменяют данные, а какие «безопасны» для константного объекта?

Объявляя функцию класса с модификатором const, вы сообщаете компилятору, что эта функция может вызываться для константных объектов. Если функция класса не была явно объявлена константной, компилятор считает, что она изменяет данные объекта, и не позволит вызывать ее для константного объекта.

Впрочем, это еще не все. Простое *объявление* константной функции класса еще не гарантирует, что функция будет вести себя положенным образом, поэтому компилятор требует обязательного повторения модификатора const при определении функции (const становится частью сигнатуры функции, поэтому о константности функции известно как компилятору, так и компоновщику). Затем компилятор следит за соблюдением константности в определении функции и выдает сообщения об ошибке при любых попытках изменения переменных объекта *или* вызова неконстантной функции. Следовательно, любая функция, объявленная константной, заведомо должна соблюдать правила константности в своем определении.

Чтобы понять синтаксис объявления константной функции класса, для начала стоит вспомнить, что ключевое слово const перед объявлением функции означает *константность возвращаемого значения*, поэтому этот вариант не приносит желаемых результатов. Вместо этого модификатор const размещается *после* списка аргументов. Пример:

```
//: C08:ConstMember.cpp
class X {
  int i;
public:
  X(int ii);
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
```

```
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

Обратите внимание: ключевое слово const должно повторяться в определении функции, иначе компилятор будет считать, что речь идет о другой функции. Поскольку функция f() является константной, при любых попытках изменения переменной i *или* вызова другой неконстантной функции класса компилятор выдает сообщение об ошибке.

Константные функции могут вызываться как для константных, так и для неконстантных объектов. Таким образом, они составляют наиболее универсальную форму функций класса. Любая функция, не изменяющая данные класса, должна объявляться константной, чтобы ее можно было использовать с константными объектами.

В следующем примере сравниваются константные и неконстантные функции:

```
//: C08:Quoter.cpp
// Случайный выбор цитаты
#include <iostream>
#include <cstdlib> // Генератор случайных чисел
#include <ctime> // Раскрутка генератора случайных чисел
using namespace std;

class Quoter {
  int lastquote;
public:
  Quoter();
  int lastQuote() const;
  const char* quote();
};

Quoter::Quoter(){
  lastquote = -1;
  srand(time(0)); // Раскрутка генератора случайных чисел
}

int Quoter::lastQuote() const {
  return lastquote;
}

const char* Quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
    "Things that make us happy, make us wise",
  };
  const int qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
```

```
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

int main() {
  Quoter q;
  const Quoter cq;
  cq.lastQuote(); // Можно
//!  cq.quote(); // Нельзя: неконстантная функция
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
} ///:~
```

Ни конструкторы, ни деструкторы не могут объявляться константными, потому что они практически всегда модифицируют состояние объекта во время инициализации и зачистки. Функция quote() тоже не может быть константной, потому что она изменяет переменную lastquote (см. команду return). С другой стороны, функция lastQuote() изменений не вносит, поэтому ее можно объявить константной и затем вызвать для константного объекта cq.

Константность логическая и константность физическая

А что, если вы хотите создать константную функцию, но при этом сохранить возможность модификации некоторых данных объекта? В этом контексте иногда различают два вида константности: *логическую* и *физическую*. Физическая константность означает, что ни один бит объекта не изменился, поэтому его двоичное представление полностью сохраняется. Логическая константность означает, что на концептуальном уровне объект сохраняет константность, но при этом возможны изменения в отдельных членах. Но если контейнер видит, что объект объявлен с ключевым словом const, он ревностно защищает этот объект и обеспечивает его физическую константность. Для реализации логической константности существуют два механизма модификации переменных из константных функций. Первый, традиционный, механизм основан на отмене константности посредством приведения типа. Внешне он выглядит довольно странно: this (ключевое слово, определяющее адрес текущего объекта) преобразуется в указатель на объект текущего типа. На первый взгляд кажется, что указатель this *уже* им является. Тем не менее в константных функциях классов он в действительности соответствует указателю на константу, поэтому преобразование в обычный указатель отменяет свойство константности для выполняемой операции. Пример:

```
//: C08:Castaway.cpp
// Отмена константности преобразованием типа

class Y {
  int i;
public:
  Y();
  void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
```

```
//!  i++; // Ошибка -- константная функция класса
  ((Y*)this)->i++; // Можно: отмена константности
  // Рекомендуется: синтаксис явного приведения типов C++:
  (const_cast<Y*>(this))->i++;
}

int main() {
  const Y yy;
  yy.f(); // Функция изменяет константный объект!
} ///:~
```

Такое решение работает и встречается в унаследованном коде, но пользоваться им не рекомендуется. Проблема заключается в том, что нарушение константности скрывается в определении функции класса, тогда как в интерфейсе класса ничто не указывает на модификацию объекта; чтобы узнать об этом, необходимо иметь доступ к исходным текстам программы (кроме того, программист должен заподозрить нарушение константности и найти, где в программе выполняется преобразование типа). Чтобы четко выразить свои намерения, следует включить в объявление класса ключевое слово mutable, которое указывает, что некоторая переменная класса может изменяться в константных объектах:

```
//: C08:Mutable.cpp
// Ключевое слово "mutable"

class Z {
  int i;
  mutable int j;
public:
  Z();
  void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
//! i++; // Ошибка -- константная функция
    j++; // Можно: mutable
}

int main() {
  const Z zz;
  zz.f(); // Функция изменяет константный объект!
} ///:~
```

На этот раз пользователь по объявлению класса сразу видит, какие члены класса могут изменяться константной функцией.

Хранение данных в постоянной памяти

Объекты, объявленные константными, могут рассматриваться как кандидаты на размещение в постоянной памяти; данный фактор нередко играет важную роль в программировании встроенных систем. Однако простого объявления объекта с ключевым словом const недостаточно — к объектам, хранимым в постоянной памяти, предъявляются гораздо более жесткие требования. Естественно, объект должен обладать физической, а не логической константностью. Это условие легко проверяется, если логическая константность обеспечивается только ключевым

словом mutable; с другой стороны, если константность отменяется внутри константной функции класса, компилятор вряд ли сможет обнаружить ее нарушение. Кроме того:

- класс или структура не могут иметь пользовательских конструкторов или деструкторов;
- не допускается существование базовых классов (глава 14) или объектов-членов с пользовательскими конструкторами или деструкторами.

Операции записи в любую часть константного объекта типа, подходящего для хранения в постоянной памяти, приводят к неопределенным последствиям. Хотя объект, удовлетворяющий всем условиям, может быть размещен в постоянной памяти, ни для одного объекта нельзя *потребовать* соблюдения этого условия.

Ключевое слово volatile

Синтаксис ключевых слов volatile и const схож, но их смысл совершенно иной. Ключевое слово volatile означает, что данные могут изменяться без ведома компилятора. Иногда рабочая среда модифицирует данные (например, через механизмы многопоточности, многозадачности или обработки прерываний); ключевое слово volatile сообщает компилятору, что он не должен делать каких-либо предположений о значениях данных, особенно в целях оптимизации.

Если компилятор знает, что данные ранее были загружены в регистр и значение этого регистра не изменялось, обычно при следующем обращении он не станет читать данные заново. Но если данные объявлены с ключевым словом volatile, компилятор уже не вправе делать такие допущения, поскольку данные могли быть изменены другим процессом. Следовательно, компилятор читает данные заново и не пытается оптимизировать программу, исключив из нее операцию чтения, лишнюю в обычной ситуации.

Синтаксис создания изменчивых объектов почти не отличается от синтаксиса создания константных объектов. Также с помощью конструкции const volatile допускается создание объектов, которые не могут изменяться прикладным программистом, но изменяются под воздействием внешнего фактора. Для примера рассмотрим класс, связанный с неким коммуникационным устройством:

```
//: C08:Volatile.cpp
// Ключевое слово volatile

class Comm {
  const volatile unsigned char byte;
  volatile unsigned char flag;
  enum { bufsize = 100 };
  unsigned char buf[bufsize];
  int index;
public:
  Comm();
  void isr() volatile;
  char read(int index) const;
};
```

```
Comm::Comm() : index(0), byte(0), flag(0) {}

// Демонстрация: нельзя использовать
// для обработки прерывания:
void Comm::isr() volatile {
  flag = 0;
  buf[index++] = byte;
  // Возврат в начало буфера:
  if(index >= bufsize) index = 0;
}

char Comm::read(int index) const {
  if(index < 0 || index >= bufsize)
    return 0;
  return buf[index];
}

int main() {
  volatile Comm Port;
  Port.isr(); // OK
//! Port.read(0); // Ошибка, функция read() не объявлена volatile
} ///:~
```

Ключевое слово volatile, как и const, может применяться к переменным классов, функциям и объектам. Для объектов, объявленных с ключевым словом volatile, можно вызывать только функции классов, также объявленные как volatile.

Функция isr() не может использоваться как обработчик прерывания, поскольку функциям классов в первом аргументе скрыто передается адрес текущего объекта (this), а обработчик прерывания вызывается без аргументов. Проблема решается объявлением isr() как статической функции класса; эта тема рассматривается в главе 10.

Синтаксис volatile идентичен синтаксису const, поэтому эти два ключевых слова часто рассматриваются вместе. Иногда для них используется обобщенное название *квалификатор c-v*.

Итоги

Ключевое слово const позволяет определять константные объекты, аргументы функций, возвращаемые значения и функции классов. Оно заменяет собой препроцессорный механизм подстановки значений, сохраняя все его преимущества. Концепция константности открывает в программах новые возможности проверки типов и защиты. Обеспечение *строгой константности* (то есть использование ключевого слова const всюду, где можно) иногда бывает жизненно важным для проекта.

В принципе, вы можете не обращать внимания на ключевое слово const и продолжать применять старые приемы программирования на С, но это ключевое слово призвано помочь вам. Начиная с главы 11 мы начнем интенсивно использовать ссылки, и вы убедитесь, насколько важную роль иногда играет ключевое слово const применительно к аргументам функций.

Упражнения

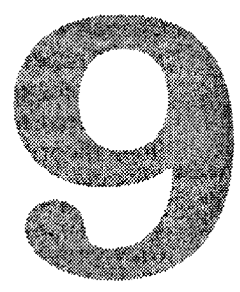
1. Создайте три значения const int и просуммируйте их; результат должен задавать размер массива в его определении. Попробуйте откомпилировать программу на С и посмотрите, что произойдет (обычно компилятор С++ переключается в режим компиляции С при помощи флага командной строки).
2. Убедитесь в том, что компиляторы С и С++ действительно по-разному работают с константами. Создайте глобальную константу и используйте ее в глобальном константном выражении; откомпилируйте программу в режимах С и С++.
3. Создайте примеры определений констант всех встроенных типов и их разновидностей. Используйте их с другими константами в выражениях, определяющих новые константы. Убедитесь в том, что программа успешно компилируется.
4. Создайте определение константы в заголовочном файле, включите этот заголовочный файл в два cpp-файла, откомпилируйте и скомпонуйте эти файлы. Ошибок быть не должно. Повторите эксперимент на языке С.
5. Создайте константу, значение которой определяется на стадии выполнения; для этого программа должна выяснять текущее время на момент запуска (воспользуйтесь стандартным заголовочным файлом <ctime>). Позднее в программе попытайтесь снова считать текущее время в константу и посмотрите, что произойдет.
6. Создайте константный массив char и попробуйте изменить один из элементов.
7. Создайте в файле объявление внешней константы (extern const); включите в файл функцию main(), которая выводит значение extern const. Создайте определение внешней константы в другом файле, откомпилируйте и скомпонуйте два файла.
8. Создайте два указателя на const long, используя обе формы объявления. Свяжите один из указателей с массивом long. Убедитесь в том, что значение указателя можно увеличивать и уменьшать, но нельзя изменить те данные, на которые он ссылается.
9. Создайте константный указатель на double, свяжите его с массивом double. Убедитесь в том, что программа может изменять данные, на которые ссылается указатель, но не может изменять значение указателя.
10. Создайте константный указатель на константный объект. Убедитесь в том, что вы можете читать значение, на которое ссылается указатель, но не можете изменить ни сам указатель, ни тот объект, на который он ссылается.
11. Уберите комментарий в ошибочной строке примера PointerAssignment.cpp и посмотрите, какую ошибку выдаст ваш компилятор.
12. Создайте символьный массив-литерал и указатель, установленный в начало массива. Затем попробуйте модифицировать элементы массива через указа-

тель. Выдает ли ваш компилятор сообщение об ошибке? А должен ли он это делать? Если не должен, то почему?

13. Напишите функцию, которая получает по значению константный аргумент; попробуйте изменить этот аргумент в теле функции.
14. Напишите функцию, которая получает аргумент типа float по значению. Внутри функции свяжите аргумент со ссылкой const float& и в дальнейшем, используя ссылку, убедитесь в том, что аргумент остается неизменным.
15. В программе ConstReturnValues.cpp последовательно удаляйте комментарии в строках, содержащих ошибки. Посмотрите, какие сообщения выдает компилятор.
16. В программе ConstPointer.cpp последовательно удаляйте комментарии в строках, содержащих ошибки. Посмотрите, какие сообщения выдает компилятор.
17. Создайте новую версию программы ConstPointer.cpp (назовите ее ConstReference.cpp) со ссылками вместо указателей. Возможно, для этого вам придется познакомиться с материалом главы 11.
18. В программе ConstTemporary.cpp удалите комментарий в строке, содержащей ошибку. Посмотрите, какое сообщение выдаст компилятор.
19. Создайте класс с константными и неконстантными переменными типа float. Задайте их значения с использованием списка инициализирующих значений конструктора.
20. Создайте класс MyString, который содержит переменную string, конструктор, инициализирующий эту переменную, и функцию print(). Измените пример StringStack.cpp так, чтобы в контейнере хранились объекты MyString, и организуйте их вывод в функции main().
21. Создайте класс с константной переменной, значение которой задается в списке инициализирующих значений конструктора, и безымянным перечислением, используемым для определения размера массива.
22. В примере ConstMember.cpp удалите модификатор const в определении функции класса, но оставьте его в объявлении. Посмотрите, какое сообщение выдаст компилятор.
23. Создайте класс, содержащий константные и неконстантные функции. Создайте несколько константных и неконстантных объектов класса, попробуйте вызывать разные функции для разных типов объектов.
24. Создайте класс, содержащий константные и неконстантные функции. Попробуйте вызвать неконстантную функцию из константной функции и посмотрите, какое сообщение об ошибке будет выдано компилятором.
25. В примере Mutable.cpp удалите комментарий в строке, содержащей ошибку. Посмотрите, какое сообщение выдаст компилятор.
26. Измените программу Quoter.cpp так, чтобы функция quote() была константной, а переменная lastquote объявлялась как mutable.

27. Создайте класс с изменчивой (volatile) переменной. Попробуйте обратиться к этой переменной из функций, объявленных с модификатором volatile и без него, и посмотрите, как на это отреагирует компилятор. Создайте изменчивые и неизменчивые объекты класса, попробуйте вызывать для них обе разновидности функций. Посмотрите, какие вызовы будут выполнены успешно и какие сообщения об ошибках выдаст компилятор.
28. Создайте класс bird с функцией fly() и класс rock, у которого такая функция отсутствует. Создайте объект класса rock, получите его адрес и присвойте его переменной типа void*. Затем возьмите значение void*, присвойте его bird* (для этого потребуется приведение типа) и вызовите fly() через указатель. Теперь понятно, почему свободное присваивание через void* (без приведения типа) в языке C является недостатком, который нельзя было оставлять в C++?

Подставляемые функции



Одной из важных особенностей языка C++, унаследованных им от C, является эффективность. Если бы C++ по эффективности заметно уступал C, то для довольно широкого круга задач он оказался бы неприемлемым.

В частности, для повышения эффективности в C использовались *макросы*, которые внешне походили на функции, но не требовали дополнительных накладных расходов, связанных с вызовом. Макроподстановка реализуется на уровне препроцессора, а не на уровне компилятора. Препроцессор замещает все вызовы макроса его кодом, поэтому программа избавляется от занесения аргументов в стек, выполнения ассемблерной команды CALL, возвращения аргументов и выполнения ассемблерной команды RETURN. Вся работа выполняется препроцессором; программист получает в свое распоряжение удобный и наглядный синтаксис вызова функции, причем это ему ничего не стоит.

Использование препроцессорных макросов в C++ сопряжено с двумя проблемами. Первая проблема характерна также для C: макрос внешне похож на вызов функции, но не всегда работает, как функция. Из-за этого в программах появляются ошибки, которые очень тяжело обнаружить. Вторая проблема характерна только для C++: препроцессор не имеет прав доступа к данным классов. Следовательно, препроцессорные макросы не могут использоваться в качестве функций классов.

В C++ существует механизм *подставляемых функций*, который, сохраняя эффективность препроцессорных макросов, добавляет к ним надежность и классовый уровень доступа полноценных функций. В этой главе мы рассмотрим проблемы, связанные с применением препроцессорных макросов в C++, и увидим, как эти проблемы решаются при помощи подставляемых функций. Также будет подробно описан внутренний механизм работы подставляемых функций.

Недостатки препроцессорных макросов

Все проблемы, связанные с использованием препроцессорных макросов, имеют общую причину: у программиста возникает иллюзия того, что препроцессор ра-

ботает точно так же, как компилятор. Конечно, изначально предполагалось, что макросы *должны* выглядеть и работать, как вызовы функций, поэтому такая иллюзия вполне объяснима. Трудности начинаются там, где имеют место нетривиальные различия.

Рассмотрим следующий простой пример:

```
#define F (x) (x+1)
```

Теперь сделаем макроподстановку F вида

```
F(1)
```

Несколько неожиданно эта макроподстановка расширяется препроцессором в следующее выражение:

```
(x) (x + 1) (1)
```

Проблемы возникают из-за лишнего пробела между символом F и открывающей круглой скобкой в определении макроса. Если удалить этот пробел, то макрос *действительно* можно будет вызывать с пробелом:

```
F (1)
```

Тогда он будет правильно расширен до следующего выражения:

```
(1 + 1)
```

Приведенный пример тривиален, а сама проблема очевидна с первого взгляда. Настоящие трудности начинаются при вызовах в макросах выражений.

Существуют две основные категории ошибок. Во-первых, расширение выражений в макросах может привести к изменению очередности обработки операндов в выражениях. Пример:

```
#define FLOOR(x,b) x>=b?0:1
```

Попробуем теперь передать в аргументах следующие выражения:

```
if(FLOOR(a&0x0f,0x07)) // ...
```

Тогда макрос будет расширен до вида

```
if(a&0x0f>=0x07?0:1)
```

Приоритет оператора & ниже, чем оператора >=, поэтому результат подстановки оказывается весьма неожиданным. Как только проблема обнаружена, она легко решается — для этого достаточно заключить все компоненты в определении макроса в круглые скобки (кстати, это вообще рекомендуется делать при создании процессорных макросов):

```
#define FLOOR(x,b) ((x)>=(b)?0:1)
```

С другой стороны, обнаружить проблему тоже бывает непросто. Иногда это происходит уже после того, как программист убеждается в работоспособности макроса. В версии предыдущего макроса без круглых скобок *большинство* выражений будет работать правильно, потому что приоритет оператора >= ниже, чем приоритет операторов +, /, -- и даже оператора поразрядного сдвига. Возникает обманчивое впечатление, что макрос работает со *всеми* выражениями, в том числе и с использующими поразрядные операторы.

Проблемы первого типа решаются тщательным соблюдением правила программирования, в соответствии с которым все компоненты макросов заключаются в круглые скобки. Со второй категорией ошибок дело обстоит сложнее. В отличие от обычных функций при каждом вызове макроса происходит вычисление его

аргумента. Пока макросы вызываются с обычными переменными, все идет нормально, но если вычисление аргумента приводит к побочным эффектам, то результат часто оказывается неожиданным и определенно не соответствует обычным правилам вызова функций.

Например, следующий макрос проверяет, принадлежит ли его аргумент некоторому интервалу:

```
#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)
```

Пока макрос вызывается с «нормальными» аргументами, он ничем не отличается от обычных функций. Но стоит вам расслабиться и посчитать, что он *является* обычной функцией, проблем не избежать. Пример:

```
//: C09:MacroSideEffects.cpp
#include "../require.h"
#include <fstream>
using namespace std;

#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)

int main() {
  ofstream out("macro.out");
  assure(out, "macro.out");
  for(int i = 4; i < 11; i++) {
    int a = i;
    out << "a = " << a << endl << '\t';
    out << "BAND(++a)=" << BAND(++a) << endl;
    out << "\t a = " << a << endl;
  }
} ///:~
```

Обратите внимание: имя макроса записано символами верхнего регистра. Благодаря этому полезному обозначению читатель сразу видит, что речь идет о макросе, а не о функции, что несколько упрощает диагностику проблем.

Программа выводит совсем не тот результат, который был бы получен в случае обычной функции:

```
a = 4
  BAND(++a)=0
   a = 5
a = 5
  BAND(++a)=8
   a = 8
a = 6
  BAND(++a)=9
   a = 9
a = 7
  BAND(++a)=10
   a = 10
a = 8
  BAND(++a)=0
   a = 10
a = 9
  BAND(++a)=0
   a = 11
a = 10
  BAND(++a)=0
   a = 12
```

Когда значение a равно 4, обрабатывается только первая часть условия, поэтому выражение вычисляется только один раз, а от побочного эффекта вызова макроса значение a увеличивается до 5, как и при обычном вызове функции в аналогичной ситуации. Но когда число входит в интервал, проверяются *обе* составляющие условия, что приводит к двукратному выполнению инкремента. Результат определяется повторным вычислением аргумента, что приводит к третьему инкременту. А когда число снова выходит из интервала, также проверяются обе составляющие условия, то есть инкремент выполняется дважды. Получается, что побочные эффекты от применения макроса различаются в зависимости от аргумента.

Естественно, макрос, который имитирует вызов функции, не должен вести себя подобным образом. В данном случае наиболее очевидное решение — заменить макрос обычной функцией; конечно, это связано с дополнительными затратами и снижением эффективности при многократном вызове. К сожалению, проблема не всегда столь очевидна. Иногда приходится использовать чужие библиотеки, в которых перемешаны функции и макросы, что приводит к возникновению крайне изощренных ошибок. Например, макрос putc() в библиотеке stdio может вычислять свой второй аргумент дважды, о чем упоминается в стандарте C. Кроме того, небрежные реализации функции toupper() в виде макроса тоже могут вычислять свой аргумент больше одного раза, что приведет к неожиданным результатам при вызове toupper(*p++).

Макросы и доступ

Аккуратное программирование и препроцессорные макросы играют важную роль в программировании на C. В принципе, можно было бы перенести эту методику в C++, если бы не одно препятствие: макросы не поддерживают концепцию видимости, обязательную для функций классов. Препроцессор просто выполняет текстовую замену, поэтому вам не удастся использовать представленную ниже или подобные конструкции:

```
class X {
  int i;
public:
#define VAL(X::i) // Ошибка
```

Кроме того, вам не удастся указать, на какой объект ссылается макрос. Проще говоря, макросы не обладают средствами для представления классовой видимости. Если у программиста не будет достойной альтернативы макросам, у него появится искушение объявить все данные класса со спецификатором public, чтобы повысить эффективность доступа. Но в этом случае он раскроет базовую реализацию и не сможет вносить в нее изменения, а также лишится защиты, предоставляемой механизмом ограничения доступа.

Подставляемые функции

Механизм решения проблемы доступа к закрытым членам классов в C++ позволяет преодолеть *все* недостатки препроцессорных макросов. Для этого концепция макросов переводится под контроль компилятора, как и должно быть. В C++ макросы реализуются в форме *подставляемых* (inline) *функций*, которые являются

полноценными функциями во всех отношениях. Подставляемые функции обладают всеми свойствами обычных функций. Единственное различие заключается в том, что подставляемые функции расширяются «на месте», как и препроцессорные макросы, избавляя программу от издержек, связанных с вызовами функций. Следовательно, вместо макросов (почти) всегда лучше использовать подставляемые функции.

Любая функция, определяемая в теле класса, автоматически становится подставляемой, но обычные функции тоже можно сделать подставляемыми — для этого они объявляются с ключевым словом inline. Но чтобы ключевое слово было учтено компилятором, объявление должно содержать тело функции, иначе компилятор рассматривает его как обычное объявление. Следовательно, следующая строка просто объявляет функцию, и ничего больше:

```
inline int plusOne(int x);
```

В правильном объявлении подставляемой функции должно содержаться ее тело:

```
inline int plusOne(int x) {return ++x; }
```

Обратите внимание: компилятор, как обычно, проверяет правильность списка аргументов и возвращаемого значения с выполнением всех необходимых преобразований; препроцессор на это не способен. Кроме того, попытка реализовать эту функцию в виде макроса приводит к нежелательному побочному эффекту.

Определения подставляемых функций почти всегда размещаются в заголовочных файлах. Встречая такое определение, компилятор заносит тип функции (сигнатуру в сочетании с типом возвращаемого значения) *и* тело функции в свою таблицу символических имен. Встретив вызов функции, компилятор проверяет правильность вызова и использования возвращаемого значения, а затем подставляет тело функции на место вызова, сокращая лишние затраты. Конечно, подставляемый код занимает определенное место, но при небольшом размере функции этот код будет даже меньше кода, сгенерированного для обычного вызова функции (с занесением аргументов в стек и передачей управления командой CALL).

Подставляемая функция в заголовочном файле обладает особым статусом: заголовочный файл, содержащий функцию *и* ее определение, должен включаться во все файлы, где эта функция используется, однако это не приводит к ошибкам повторного определения (тем не менее эти определения везде должны быть идентичными).

Подставляемые функции внутри классов

Определения подставляемых функций обычно должны начинаться с ключевого слова inline. Впрочем, внутри определения класса это не обязательно. Любая функция, определяемая внутри класса, автоматически становится подставляемой. Пример:

```
//: C09:Inline.cpp
// Подставляемые функции внутри классов
#include <iostream>
#include <string>
using namespace std;

class Point {
```

```
  int i, j, k;
public:
  Point(): i(0), j(0), k(0) {}
  Point(int ii, int jj, int kk)
    : i(ii), j(jj), k(kk) {}
  void print(const string& msg = "") const {
    if(msg.size() != 0) cout << msg << endl;
    cout << "i = " << i << ", "
         << "j = " << j << ", "
         << "k = " << k << endl;
  }
};

int main() {
  Point p, q(1,2,3);
  p.print("value of p");
  q.print("value of q");
} ///:~
```

Здесь оба конструктора и функция print() становятся подставляемыми по умолчанию. В функции main() факт использования подставляемых функций абсолютно незаметен, как и должно быть. Логически поведение функции остается постоянным независимо от того, была она подставлена в месте вызова или нет (если нет, вероятно, у вас какие-то проблемы с компилятором). Различия проявляются только в быстродействии.

Конечно, у программиста возникает искушение повсеместно использовать подставляемые функции внутри объявлений класса, потому что это избавляет его от необходимости создавать внешние определения функций. Однако следует помнить, что подставляемые функции предназначены для расширения возможностей оптимизации со стороны компилятора. Если объявить большую функцию подставляемой, то код этой функции будет дублироваться при каждом ее вызове и увеличение объема программы перевесит выигрыш в быстродействии. Существует только один надежный способ: поэкспериментируйте и выясните, к каким последствиям приводит подстановка тех или иных функций.

Функции доступа

Одну из важнейших областей применения подставляемых функций в классах составляют так называемые *функции доступа* — небольшие функции, предназначенные для чтения и записи внутренних переменных объекта. Следующий пример наглядно показывает, почему подстановка так важна для функций доступа:

```
//: C09:Access.cpp
// Подставляемые функции доступа

class Access {
  int i;
public:
  int read() const { return i; }
  void set(int ii) { i = ii; }
};

int main() {
```

```
  Access A;
  A.set(100);
  int x = A.read();
} ///:~
```

Пользователь класса никогда не работает с переменными состояния напрямую, поэтому эти переменные объявляются закрытыми и находятся под контролем разработчика класса. Весь доступ к закрытым данным класса организуется через интерфейс функций класса. Благодаря подстановке обращения к переменным класса выполняются чрезвычайно эффективно. Без нее для вызова read() будет сгенерирован код, включающий занесение this в стек и передачу управления ассемблерной командой CALL. На большинстве компьютеров этот код будет занимать больше места, чем подставляемый код, и наверняка будет медленнее работать.

А если бы подставляемых функций не было? Разработчик класса, стремясь улучшить эффективность доступа, просто объявил бы переменную i открытой и разрешил прямой доступ к ней, избавившись от всех издержек, связанных с вызовом функции. С точки зрения архитектуры такое решение считается крайне неудачным, поскольку разработчик класса уже никогда не сможет изменить его. Интерфейс намертво привязывается к переменной i типа int. Позднее может оказаться, что информация состояния гораздо лучше представляется в формате float, а не int, однако переменная int i уже стала частью открытого интерфейса и изменить ее не удастся. Или вдруг в процессе чтения или присваивания i потребуется выполнять дополнительные вычисления, но если переменная является открытой, это сделать не удастся. С другой стороны, если для чтения и изменения состояния объекта всегда использовались функции доступа, вы сможете изменять базовое представление объекта по своему усмотрению.

Кроме того, работа с данными класса через функции доступа позволяет включить в функцию код отслеживания изменений данных. Это бывает чрезвычайно полезно в процессе отладки. Если переменная объявлена открытой, любой желающий сможет в любой момент изменить ее, и вы ничего не будете знать об этих изменениях.

Функции чтения и записи

Некоторые авторы дополнительно разделяют функции доступа на *функции чтения* (предназначенные для получения информации о состоянии объекта) и *функции записи*, изменяющие состояние объекта. Более того, механизм перегрузки функций позволяет использовать общее имя для функций чтения и записи. То, какая операция выполняется в каждом конкретном случае, компилятор определяет по контексту вызова функции. Пример:

```
//: C09:Rectangle.cpp
// Функции чтения и записи

class Rectangle {
  int wide, high;
public:
  Rectangle(int w = 0, int h = 0)
    : wide(w), high(h) {}
  int width() const { return wide; } // Чтение
```

```
  void width(int w) { wide = w; } // Запись
  int height() const { return high; } // Чтение
  void height(int h) { high = h; } // Запись
};
int main() {
  Rectangle r(19, 47);
  // Изменение ширины и высоты:
  r.height(2 * r.width());
  r.width(2 * r.height());
} ///:~
```

В конструкторе значения wide и high задаются при помощи списка инициализирующих значений (кратко представленных в главе 8, подробно — в главе 14) с использованием псевдоконструкторов для встроенных типов.

Поскольку имена функций классов не могут совпадать с именами переменных, некоторые программисты включают в имена переменных начальный символ подчеркивания. Однако этот префикс обычно является признаком зарезервированных идентификаторов, поэтому использовать его нежелательно.

Функции чтения и записи часто обозначаются префиксами get и set:

```
//: C09:Rectangle2.cpp
// Функции чтения и записи с префиксами "get" и "set"
class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0)
    : width(w), height(h) {}
  int getWidth() const { return width; }
  void setWidth(int w) { width = w; }
  int getHeight() const { return height; }
  void setHeight(int h) { height = h; }
};
int main() {
  Rectangle r(19, 47);
  // Изменение width и height:
  r.setHeight(2 * r.getWidth());
  r.setWidth(2 * r.getHeight());
} ///:~
```

Конечно, работа функций чтения и записи не сводится к простой выборке внутренних переменных. Иногда они используются для выполнения более сложных операций. Так, в следующем примере представлен простой класс Time, в работе которого задействованы функции стандартной библиотеки C:

```
//: C09:Cpptime.h
// Простой класс для работы со временем
#ifndef CPPTIME_H
#define CPPTIME_H
#include <ctime>
#include <cstring>
class Time {
  std::time_t t;
  std::tm local;
  char asciiRep[26];
  unsigned char lflag, aflag;
  void updateLocal() {
    if(!lflag) {
      local = *std::localtime(&t);
```

```
        lflag++;
      }
    }
    void updateAscii() {
      if(!aflag) {
        updateLocal();
        std::strcpy(asciiRep,std::asctime(&local));
        aflag++;
      }
    }
  public:
    Time() { mark(); }
    void mark() {
      lflag = aflag = 0;
      std::time(&t);
    }
    const char* ascii() {
      updateAscii();
      return asciiRep;
    }
    // Разность в секундах:
    int delta(Time* dt) const {
      return int(std::difftime(t, dt->t));
    }
    int daylightSavings() {
      updateLocal();
      return local.tm_isdst;
    }
    int dayOfYear() { // День года, начиная с 1 января
      updateLocal();
      return local.tm_yday;
    }
    int dayOfWeek() { // День недели, начиная с воскресенья
      updateLocal();
      return local.tm_wday;
    }
    int since1900() { // Количество лет с 1900 года
      updateLocal();
      return local.tm_year;
    }
    int month() { // Месяц, начиная с января
      updateLocal();
      return local.tm_mon;
    }
    int dayOfMonth() {
      updateLocal();
      return local.tm_mday;
    }
    int hour() { // Час от полуночи, по 24-часовой шкале
      updateLocal();
      return local.tm_hour;
    }
    int minute() {
      updateLocal();
      return local.tm_min;
    }
    int second() {
      updateLocal();
```

```
    return local.tm_sec;
  }
};
#endif // CPPTIME_H ///:~
```

В класс Time включены аналоги нескольких функций стандартной библиотеки С, обеспечивающих разное представление времени. Впрочем, обновлять все представления не нужно, поэтому класс использует базовое представление time_t t, а для переменной tm local и символьного представления asciiRep определены флаги, указывающие, была ли переменная обновлена до текущего времени. Две закрытые функции updateLocal() и updateAscii() проверяют состояние флагов и производят обновление в зависимости от результата проверки.

Конструктор вызывает функцию mark(), которая также может вызываться пользователем для перевода объекта на текущее время. Функция сбрасывает флаги, указывая тем самым, что местное время и символьное представление стали недействительными. Функция ascii() вызывает функцию updateAscii(), которая копирует результат вызова стандартной библиотечной функции asctime() в локальный буфер (это необходимо, потому что функция asctime() использует статическую область данных, содержимое которой стирается при вызове функции в другом месте). Функция ascii() возвращает адрес этого локального буфера.

Все функции начиная с daylightSavings() используют функцию updateLocal(), в результате чего составные фрагменты подставляемого кода получаются относительно большими. Возможно, итоговые затраты окажутся неоправданно высокими, особенно если учесть, что функции будут вызываться не слишком часто. Впрочем, из этого не следует, что все функции нужно преобразовать в неподставляемые. А если вы все же решите объявить остальные функции неподставляемыми, по крайней мере, оставьте подставляемой функцию updateLocal(). Ее код окажется продублированным в неподставляемых функциях, а из программы будут исключены лишние вызовы.

Небольшая тестовая программа для класса Time:

```
//: C09:Cpptime.cpp
// Тестирование простого класса для работы со временем
#include "Cpptime.h"
#include <iostream>
using namespace std;
int main() {
  Time start;
  for(int i = 1; i < 1000; i++) {
    cout << i << ' ';
    if(i%10 == 0) cout << endl;
  }
  Time end;
  cout << endl;
  cout << "start = " << start.ascii();
  cout << "end = " << end.ascii();
  cout << "delta = " << end.delta(&start);
} ///:~
```

Программа создает объект Time, в течение некоторого времени работает с ним, а затем создает второй объект Time в момент окончания операции. Оба объекта Time используются для отображения начального и конечного времени, а также продолжительности операции.

Классы Stash и Stack с подставляемыми функциями

Взяв на вооружение подставляемые функции, мы можем повысить эффективность классов Stash и Stack:

```
//: C09:Stash4.h
// Подставляемые функции
#ifndef STASH4_H
#define STASH4_H
#include "../require.h"
class Stash {
  int size;      // Размер каждого элемента
  int quantity;  // Количество элементов
  int next;      // Следующий пустой элемент
  // Динамически выделяемый байтовый массив:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int sz) : size(sz), quantity(0),
    next(0), storage(0) {}
  Stash(int sz, int initQuantity) : size(sz),
    quantity(0), next(0), storage(0) {
    inflate(initQuantity);
  }
  Stash::~Stash() {
    if(storage != 0)
      delete []storage;
  }
  int add(void* element);
  void* fetch(int index) const {
    require(0 <= index, "Stash::fetch (-)index");
    if(index >= next)
      return 0; // Признак конца
    // Указатель на запрашиваемый элемент:
    return &(storage[index * size]);
  }
  int count() const { return next; }
};
#endif // STASH4_H ///:~
```

Естественно, мелкие функции лучше оформить как подставляемые, но две большие функции остались в прежнем виде, поскольку для них подстановка не дает сколько-нибудь заметного выигрыша:

```
//: C09:Stash4.cpp {O}
#include "Stash4.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;
int Stash::add(void* element) {
  if(next >= quantity) // В буфере есть свободное место?
    inflate(increment);
  // Скопировать элемент в следующую свободную позицию буфера:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
```

```
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Индекс
}
void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Копирование старого буфера в новый
  delete [](storage); // Освобождение старого буфера
  storage = b; // Перевод указателя на новый буфер
  quantity = newQuantity; // Изменение размера
} ///:~
```

И вновь тестовая программа поможет убедиться в работоспособности класса:

```
//: C09:Stash4Test.cpp
//{L} Stash4
#include "Stash4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash4Test.cpp");
  assure(in, "Stash4Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

Тестовая программа почти не изменилась, поэтому результаты останутся практически теми же.

В классе Stack подставляемые функции можно использовать еще шире:

```
//: C09:Stack4.h
// Реализация с подставляемыми функциями
#ifndef STACK4_H
#define STACK4_H
#include "../require.h"
class Stack {
```

```
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack() {
    require(head == 0, "Stack not empty");
  }
  void push(void* dat) {
    head = new Link(dat, head);
  }
  void* peek() const {
    return head ? head->data : 0;
  }
  void* pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // STACK4_H ///:~
```

Обратите внимание: из класса исчез пустой деструктор Link, присутствовавший в предыдущей версии Stack. В функции pop() выражение delete oldHead просто освобождает память, занятую этим объектом Link (хотя оно не уничтожает объект data, связанный с Link).

Большинство функций просто и естественно преобразуется в подставляемую форму, особенно в классе Link. Даже для pop() такое преобразование кажется приемлемым, хотя везде, где используются условные конструкции или локальные переменные, выигрыш от применения подставляемых функций неочевиден. Но в данном случае функция настолько мала, что вреда от замены почти наверняка не будет.

Объявление *всех* функций подставляемыми упрощает работу с библиотекой, поскольку в этом случае исчезает необходимость в дополнительной компоновке, как показывает следующий тестовый пример (обратите внимание на отсутствие файла Stack4.cpp):

```
//: C09:Stack4Test.cpp
//{T} Stack4Test.cpp
#include "Stack4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Аргумент - имя файла
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Чтение файла и сохранение строк в стеке:
```

```
    while(getline(in, line))
      textlines.push(new string(line));
    // Извлечение строк из стека и вывод:
    string* s;
    while((s = (string*)textlines.pop()) != 0) {
      cout << *s << endl;
      delete s;
    }
  } ///:~
```

Программисты иногда создают классы, содержащие только подставляемые функции, чтобы весь класс находился в заголовочном файле (кстати, в книге тоже иногда встречаются аналогичные классы). Вероятно, на стадии разработки программы такое решение безвредно, хотя иногда оно приводит к более продолжительной компиляции. Но когда программа станет более или менее стабильной, стоит вернуться и там, где это уместно, преобразовать функции в неподставляемые.

Подставляемые функции и компилятор

Итак, в каких же случаях оправданно применение подставляемых функций? Чтобы ответить на этот вопрос, необходимо знать, как действует компилятор, встретив подставляемую функцию. Как и для любой другой функции, компилятор сохраняет *тип* функции (то есть ее прототип, включающий имя и типы аргументов, в сочетании с возвращаемым значением функции) в своей таблице имен. А если при этом компилятор видит, что тип подставляемой функции *и* ее тело обрабатываются без ошибок, то в таблицу имен также заносится код тела функции. Выбор представления кода (исходный текст, откомпилированные ассемблерные команды и т. д.) зависит от компилятора.

При вызове подставляемой функции компилятор сначала проверяет правильность вызова. Иначе говоря, типы аргументов должны либо точно соответствовать типам, передаваемым в списке аргументов функции, либо приводиться к ним; аналогичная проверка выполняется для возвращаемого значения. Конечно, компилятор точно так же поступает со всеми функциями, и этим он принципиально отличается от препроцессора, который не способен проверять или преобразовывать типы.

Если вся информация о типах в функции соответствует контексту вызова, хранящийся в таблице код подставляется непосредственно на место вызова. В результате программа освобождается от издержек, связанных с вызовом, и появляется возможность дополнительной оптимизации со стороны компилятора. Кроме того, если подставляемая функция является функцией класса, компилятор сохраняет адрес текущего объекта (this) там, где он должен находиться. Естественно, эта операция тоже не может выполняться препроцессором.

Ограничения

Существуют две ситуации, в которых компилятор не может выполнить подстановку. В этом случае он просто возвращается к обычной форме функции, для чего берет определение подставляемой функции и выделяет для нее память, как для обычных (неподставляемых) функций. Если замену приходится выполнять

в нескольких единицах трансляции (что в обычной ситуации привело бы к ошибке повторного определения), компилятор приказывает компоновщику игнорировать лишние определения.

Итак, компилятор не сможет выполнить подстановку, если подставляемая функция слишком сложна. Оценка сложности зависит от конкретного компилятора. Тем не менее, если «пасует» большинство компиляторов, подстановка все равно особой пользы не принесла бы. Как правило, любые циклические конструкции считаются слишком сложными для подстановки. Впрочем, это вполне логично — скорее всего, время выполнения цикла внутри функции заметно превышает издержки вызова. Если функция представляет собой последовательность простых команд, компилятор обычно легко справляется с подстановкой, но если таких команд много, издержки вызова оказываются существенно меньше издержек, связанных с выполнением тела функции. И помните: каждый раз, когда компилятор встречает в программе вызов подставляемой функции, он полностью подставляет ее тело на место вызова, что может привести к разрастанию программы без сколько-нибудь заметного повышения эффективности (в некоторых примерах книги размер подставляемых функций превышает разумные пределы, но это делается ради сокращения объема листинга).

Кроме того, компилятор не может выполнить подстановку, если в программе выполняется явное или косвенное получение адреса функции. Чтобы получить адрес, компилятор должен выделить память под функцию и использовать адрес выделенного блока. Впрочем, если адрес реально не задействован, вероятно, компилятор оставит функцию подставляемой.

Необходимо четко понимать, что объявление подставляемой функции — не более чем рекомендация для компилятора, которая его ни к чему не обязывает. Хороший компилятор подставляет компактные и простые функции, игнорируя слишком большие и сложные. В результате мы получаем именно то, чего добивались, — полноценную семантику вызова функции в сочетании с эффективностью макроса.

Опережающие ссылки

Поверхностное понимание того, как компилятор реализует подставляемые функции, иногда создает обманчивое впечатление, будто применение подставляемых функций связано с дополнительными ограничениями. Например, если подставляемая функция содержит опережающую ссылку на функцию (подставляемую или обычную), которая еще не была объявлена в классе, может показаться, что компилятор не справится с этой ситуацией:

```
//: C09:EvaluationOrder.cpp
// Порядок обработки подставляемых функций
class Forward {
  int i;
public:
  Forward() : i(0) {}
  // Вызов необъявленной функции:
  int f() const { return g() + 1; }
  int g() const { return i; }
};
int main() {
```

```
  Forward frwd;
  frwd.f();
} ///:~
```

Функция f() содержит вызов функции g(), которая еще не была объявлена. Однако такое определение работает, поскольку в соответствии со спецификацией языка подставляемые функции класса не обрабатываются до закрывающей фигурной скобки объявления класса.

Конечно, если функция g() в свою очередь вызовет f(), возникнет цепочка рекурсивных вызовов, слишком сложная для подстановки (кроме того, в f() и g() придется предусмотреть некоторое условие выхода, или рекурсия получится бесконечной).

Скрытые операции в конструкторах и деструкторах

Многие программисты склонны полагать, будто подстановка особенно эффективна для конструкторов и деструкторов. На самом деле это не всегда так. Конструкторы и деструкторы могут выполнять скрытые операции, поскольку класс может содержать связанные объекты, для которых также должны вызываться конструкторы и деструкторы. Эти связанные объекты могут быть членами класса или же входить в иерархию наследования (глава 14). Пример класса с вложенными объектами:

```
//: C09:Hidden.cpp
// Скрытые операции при подстановке
#include <iostream>
using namespace std;
class Member {
  int i, j, k;
public:
  Member(int x = 0) : i(x), j(x), k(x) {}
  ~Member() { cout << "~Member" << endl; }
};
class WithMembers {
  Member q, r, s; // Имеют конструкторы
  int i;
public:
  WithMembers(int ii) : i(ii) {} // Тривиально?
  ~WithMembers() {
    cout << "~WithMembers" << endl;
  }
};
int main() {
  WithMembers wm(1);
} ///:~
```

Конструктор Member достаточно прост для оформления в виде подставляемой функции, поскольку в нем не происходит ничего особенного, — скрытые операции, обусловленные наследованием или присутствием вложенных объектов, отсутствуют. С другой стороны, в конструкторах и деструкторах класса WithMembers происходит нечто большее, чем кажется на первый взгляд. В них автоматически вызываются конструкторы и деструкторы объектов q, r и s, причем *эти* конструкторы и деструкторы тоже являются подставляемыми, что приводит к существенному увеличению объема программы. Впрочем, это не означает, что конструкторы и деструкторы всегда должны определяться неподставляемыми; в некоторых

случаях такая подстановка имеет смысл. Кроме того, в процессе ускоренного построения прототипа программы бывает удобно использовать подставляемые функции. Однако если вас действительно беспокоят проблемы эффективности, к конструкторам и деструкторам стоит присмотреться повнимательнее.

Вынесение определений из класса

Простые и компактные подставляемые определения чрезвычайно удобны в учебных целях, поскольку они позволяют разместить больше информации на странице (или на экране). Тем не менее Дэн Сакс (Dan Saks)[1] считает, что в реальных проектах определение функций внутри класса (в его терминологии это называется латинским термином «in situ», то есть «на месте») приводит к излишнему загромождению интерфейса класса, а следовательно, усложняет работу с ним. По мнению Дэна, для сохранения четкости и наглядности интерфейса все определения следует размещать за пределами класса, а оптимизация — совсем другое дело. Если вы хотите оптимизировать программу, используйте ключевое слово inline. В этом варианте приведенная выше программа Rectangle.cpp принимает вид:

```
//: C09:Noinsitu.cpp
// Исключение функций "in situ"
class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0);
  int getWidth() const;
  void setWidth(int w);
  int getHeight() const;
  void setHeight(int h);
};
inline Rectangle::Rectangle(int w, int h)
  : width(w), height(h) {}
inline int Rectangle::getWidth() const {
  return width;
}
inline void Rectangle::setWidth(int w) {
  width = w;
}
inline int Rectangle::getHeight() const {
  return height;
}
inline void Rectangle::setHeight(int h) {
  height = h;
}
int main() {
  Rectangle r(19, 47);
  // Поменять местами width и height:
  int iHeight = r.getHeight();
  r.setHeight(r.getWidth());
  r.setWidth(iHeight);
} ///:~
```

Если потребуется сравнить последствия от замены подставляемых функций неподставляемыми, достаточно удалить ключевое слово inline (впрочем, подстав-

[1] Соавтор книги «C++ Programming Guidelines», Plum Hall, 1991.

ляемые функции обычно находятся в заголовочных файлах, тогда как неподставляемые должны находиться в собственной единице трансляции). Включение функций в документацию сводится к простому копированию текста. Функции «in situ» увеличивают объем работы и с большей вероятностью могут стать источником ошибок. Другой аргумент в пользу такого подхода — возможность унифицированного форматирования определений функций, не всегда доступная для функций «in situ».

Другие возможности препроцессора

Ранее упоминалось, что подставляемые функции *почти* всегда следует применять вместо макросов. Исключения встречаются при использовании трех специфических возможностей препроцессора C (который также является препроцессором C++): при преобразовании к строковому виду, при конкатенации и при вставке лексем. Преобразование к строковому виду (см. главу 3) выполняется директивой # и позволяет превратить идентификатор в символьный массив. Конкатенацией называется автоматическое объединение смежных символьных массивов. Эти две возможности особенно часто применяются для написания отладочного кода. Таким образом, следующая директива выводит значение произвольной переменной:

```
#define DEBUG(x) cout << #x " = " << x << endl
```

Аналогично реализуется трассировка программы с выводом команд по мере их выполнения:

```
#define TRACE(s) cerr << #s << endl; s
```

Конструкция #s преобразует команду в строку для вывода, а второй экземпляр s заново вставляет команду в программу для выполнения. Конечно, подобные фокусы иногда порождают проблемы, особенно в однострочных циклах for:

```
for(int i = 0; i < 100; i++)
  TRACE(f(i));
```

Так как макрос TRACE() в действительности содержит две команды, в однострочном цикле for будет выполняться только первая команда. Проблема решается заменой символа точки с запятой в макросе на запятую.

Вставка лексем

Вставка лексем, выполняемая директивой ##, чрезвычайно полезна для автоматизации программирования. Программист берет два идентификатора и объединяет их, создавая новый идентификатор. Пример:

```
#define FIELD(a) char* a##_string; int a##_size
class Record {
  FIELD(one);
  FIELD(two);
  FIElD(three);
  // ...
};
```

Каждый вызов макроса FIELD() создает два идентификатора: в первом хранится символьный массив, а во втором — длина массива. Такое решение не только делает

программу более понятной, но и снижает риск потенциальных ошибок и упрощает сопровождение программы.

Средства диагностики

До настоящего момента мы неоднократно использовали в примерах функции из файла require.h, но не рассматривали их определения (впрочем, там, где это было уместно, также использовался макрос assert()). Настало время познакомиться с определением этого заголовочного файла. В данном случае подставляемые функции особенно удобны — они позволяют разместить весь код функции в заголовочном файле, а это упрощает работу с пакетом — достаточно включить в программу заголовочный файл, не беспокоясь о компоновке файла реализации.

Учтите, что исключения, подробно рассмотренные во втором томе книги, предоставляют гораздо более эффективные средства обработки ошибок, особенно если вместо аварийного завершения вы собираетесь восстановить нормальный режим работы программы. Впрочем, условия, проверяемые в require.h, например недостаточное количество аргументов командной строки или ошибка при открытии файла, все равно не позволяют продолжить работу программы, поэтому в этих ситуациях вызов стандартной библиотечной функции C exit() можно считать оправданным.

Следующий заголовочный файл находится в корневом каталоге прилагаемого к книге архива, поэтому он легко доступен для примеров всех глав:

```
//: :require.h
// Проверка ошибок в программах
// Локальные "using namespace std" для старых компиляторов
#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <string>
inline void require(bool requirement.
  const std::string& msg = "Requirement failed"){
  using namespace std:
  if (!requirement) {
    fputs(msg.c_str(). stderr):
    fputs("\n". stderr):
    exit(1):
  }
}
inline void requireArgs(int argc. int args.
  const std::string& msg =
    "Must use %d arguments") {
  using namespace std:
   if (argc != args + 1) {
     fprintf(stderr. msg.c_str(). args):
     fputs("\n". stderr):
     exit(1):
   }
}
inline void requireMinArgs(int argc. int minArgs.
  const std::string& msg =
    "Must use at least %d arguments") {
```

```
  using namespace std;
  if(argc < minArgs + 1) {
    fprintf(stderr, msg.c_str(), minArgs);
    fputs("\n", stderr);
    exit(1);
  }
}

inline void assure(std::ifstream& in,
  const std::string& filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr, "Could not open file %s\n",
      filename.c_str());
    exit(1);
  }
}
inline void assure(std::ofstream& out,
  const std::string& filename = "") {
  using namespace std;
  if(!out) {
    fprintf(stderr, "Could not open file %s\n",
      filename.c_str());
    exit(1);
  }
}
#endif // REQUIRE_H ///:~
```

В аргументах по умолчанию содержатся базовые сообщения, содержимое которых при желании можно изменить.

Обратите внимание: вместо аргументов char* используются аргументы типа const string&. Это позволяет передавать функциям как char*, так и string, поэтому в общем случае такие объявления более универсальны (вероятно, данная форма объявления пригодится и в ваших программах).

В определениях requireArgs() и requireMinArgs() количество обязательных аргументов в командной строке увеличивается на 1, поскольку значение argc всегда включает имя выполняемой программы (нулевой аргумент). Поэтому оно всегда на 1 больше реального количества аргументов командной строки.

Также обратите внимание на локальные объявления using namespace std в каждой функции. Дело в том, что на момент написания книги некоторые компиляторы по ошибке не включали функции стандартной библиотеки C в пространство имен std, поэтому явное уточнение принадлежности функции приводило к ошибкам компиляции. Локальные объявления позволяют файлу require.h работать как с правильными, так и с неправильными библиотеками, причем без автоматического открытия пространства std при включении заголовочного файла.

Ниже приведена простая программа для тестирования файла require.h:

```
//: C09:ErrTest.cpp
//{T} ErrTest.cpp
// Тестирование require.h
#include "../require.h"
#include <fstream>
using namespace std;
int main(int argc, char* argv[]) {
  int i = 1;
```

```
  require(i, "value must be nonzero");
  requireArgs(argc, 1);
  requireMinArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]); // Имя файла
  ifstream nofile("nofile.xxx");
  // Ошибка:
//! assure(nofile); // Аргумент по умолчанию
  ofstream out("tmp.txt");
  assure(out);
} ///:~
```

Возможно, вам захочется пойти еще дальше и совместить открытие файла с проверкой, включив в require.h макрос вида

```
#define IFOPEN(VAR,NAME) \
  ifstream VAR(NAME); \
  assure(VAR,NAME);
```

Пример использования:

```
IFOPEN(in,argv[1]);
```

На первый взгляд макрос выглядит заманчиво, поскольку он сокращает объем вводимого текста. Тем не менее, хотя его никак нельзя назвать крайне опасным, таких решений лучше избегать. Перед вами знакомая ситуация: макрос выглядит, как функция, но ведет себя совершенно иначе — он создает объект in, который продолжает существовать за пределами макроса. Возможно, вам как автору программы это понятно, но для неопытных программистов, занимающихся сопровождением вашей программы, такое решение обернется очередной головоломкой. Не стоит усложнять C++, этот язык и так достаточно сложен. Постарайтесь воздержаться от использования препроцессорных макросов там, где это возможно.

Итоги

Базовая реализация класса должна быть скрыта от пользователей, чтобы позднее разработчик мог внести в нее изменения. Например, такие изменения вносятся для повышения эффективности, потому что разработчик нашел другой подход к решению проблемы или у него появился новый класс, который должен использоваться в реализации. Любые решения, нарушающие закрытость базовой реализации, лишь делают язык менее гибким. Следовательно, подставляемые функции играют очень важную роль, поскольку они практически устраняют необходимость в препроцессорных макросах и избавляют программу от связанных с этим проблем. Благодаря механизму подстановки функции классов по своей эффективности не уступают препроцессорным макросам.

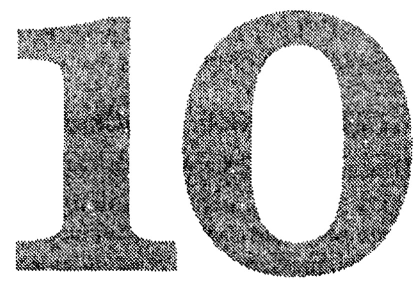
Некоторые программисты злоупотребляют подставляемыми функциями в определениях классов, потому что определять функции в классах проще и удобнее. Впрочем, проблема не столь серьезна, поскольку позднее, на стадии оптимизации кода, вы всегда сможете преобразовать подставляемые функции в неподставляемые, и такая замена никак не отразится на их функциональности. Во время разработки следует руководствоваться правилом: «Сначала заставь программу работать, а потом оптимизируй».

Упражнения

1. Напишите программу с использованием макроса F(), приведенного в начале главы. Программа должна демонстрировать ошибки расширения макроса, описанные в тексте. Исправьте макрос и докажите, что в новой версии он работает правильно.
2. Напишите программу с использованием макроса FLOOR(), приведенного в начале главы. Продемонстрируйте условия, при которых макрос работает неправильно.
3. Измените программу MacroSideEffects.cpp так, чтобы макрос BAND() работал правильно.
4. Создайте две одинаковые функции с именами f1() и f2(). Объявите функцию f1() подставляемой, а функцию f2() — неподставляемой. При помощи стандартной функции clock() из библиотеки <ctime> отметьте время начала и завершения функций и выясните, какая из двух функций работает быстрее. Вероятно, для получения осмысленных данных каждую функцию придется многократно вызывать в цикле.
5. Поэкспериментируйте с размерами и сложностью кода функций из упражнения 4. Удастся ли вам найти граничную точку, в которой подставляемая и неподставляемая функции выполняются за одинаковое время? По возможности опробуйте программу с разными компиляторами и проанализируйте различия.
6. Докажите, что для подставляемых функций по умолчанию используется внутреннее связывание.
7. Создайте класс, содержащий массив char. Включите в него подставляемый конструктор, который использует стандартную библиотечную функцию C memset() для инициализации массива значением, переданным в аргументе («по умолчанию»). Также включите в класс подставляемую функцию print() для вывода всех символов массива.
8. Возьмите пример NestFriend.cpp из главы 5 и преобразуйте все функции класса в подставляемые. Определения функций должны находиться за пределами класса. Также замените функции initialize() конструкторами.
9. В примере StringStack.cpp из главы 8 преобразуйте все функции класса в подставляемые.
10. Создайте перечисляемый тип Hue с элементами red, blue и yellow. Затем создайте класс Color, содержащий переменную типа Hue и конструктор, присваивающий Hue значение переданного аргумента. Определите для Hue функции доступа get и set. Все функции должны быть подставляемыми.
11. Измените упражнение 10 так, чтобы использовать одноименные функции чтения и записи.
12. Измените пример Cpptime.cpp так, чтобы он измерял время от запуска программы до нажатия пользователем клавиши Enter.

13. Создайте класс с двумя подставляемыми функциями. Первая функция должна вызывать вторую функцию без опережающего объявления. Напишите функцию main(), которая создает объект класса и вызывает первую функцию.
14. Создайте класс A с подставляемым конструктором по умолчанию, который выводит сообщение о вызове. Создайте новый класс B, включите в него вложенный объект класса A и определите для B подставляемый конструктор. Создайте массив объектов B и посмотрите, что произойдет.
15. Создайте большое количество объектов из предыдущего упражнения. При помощи класса Time измерьте различия во времени работы подставляемых и неподставляемых конструкторов (если у вас имеется профайлер, попробуйте также воспользоваться им).
16. Напишите программу, получающую объект string в аргументе командной строки. Включите в нее цикл for, который при каждой итерации должен удалять из string один символ и выводить новое содержимое string с помощью макроса DEBUG().
17. Исправьте макрос TRACE() так, как описано в тексте главы, и докажите, что в новой версии он работает правильно.
18. Измените макрос FIELD() так, чтобы в нем присутствовал числовой аргумент index. Создайте класс, члены которого вызывают макрос FIELD(). Включите в класс функцию для выборки поля по индексу. Напишите функцию main() для тестирования класса.
19. Измените макрос FIELD() так, чтобы он автоматически генерировал функции доступа для каждого поля (при этом данные должны оставаться закрытыми). Создайте класс, члены которого вызывают макрос FIELD(). Напишите функцию main() для тестирования класса.
20. Напишите программу, получающую два аргумента командной строки: число int и имя файла. При помощи функций require.h убедитесь в том, что программа получает правильное количество аргументов, значение int находится в интервале от 5 до 10, а файл успешно открывается.
21. Напишите программу, которая при помощи макроса IFOPEN() открывает файл как входной поток. Обратите внимание на создание объекта ifstream и его видимость.
22. (Задача повышенной трудности) Выясните, как генерируется ассемблерный код в вашем компиляторе. Создайте файл, содержащий очень маленькую функцию, и функцию main(), в которой эта функция вызывается. Сгенерируйте ассемблерный код для двух вариантов оформления функции (подставляемого и неподставляемого) и убедитесь в том, что в подставляемой версии отсутствуют команды вызова функции.

Механизм контроля имен



Именование относится к числу важнейших задач программирования; в больших проектах количество имен нередко бывает просто колоссальным.

C++ предоставляет программисту развитые средства контроля над созданием, видимостью, выделением памяти и компоновкой имен.

Ключевое слово static перегружалось в C еще до того, как появился термин «перегрузка», а в C++ у него появилось еще одно значение. Однако во всех интерпретациях ключевое слово static означало «нечто, сохраняющее свое положение», будь то физическое местонахождение в памяти или видимость на уровне файла.

В этой главе вы узнаете, как при помощи ключевого слова static управлять выделением памяти и видимостью имен. В ней вы также познакомитесь с механизмом *пространств имен*, представляющим собой новый усовершенствованный механизм контроля имен в C++. Кроме того, вы научитесь использовать функции, написанные и откомпилированные на C.

Статические элементы в языке C

В C и C++ понятие статичности (ключевое слово static) имеет два основных значения, которые, к сожалению, часто путают.

- Созданный один раз с фиксированным адресом. Объект создается в специальной *статической области данных*, а не в стеке при каждом вызове функции. Эта интерпретация определяет концепцию *статического хранения*.
- Локальный по отношению к некоторой единице трансляции (а также, как будет показано далее, локальный по отношению к видимости на уровне класса C++). В этом случае ключевое слово static управляет *видимостью* имени, то есть имя является невидимым за пределами единицы трансляции или класса. Эта интерпретация относится к концепции *связывания*, которая определяет, какие имена видны компоновщику.

В этом разделе мы рассмотрим эти два значения ключевого слова static, унаследованные из языка С.

Статические переменные в функциях

При создании локальной переменной внутри функции компилятор выделяет память для этой переменной при каждом вызове функции, для чего указатель стека смещается на соответствующую величину. Если у переменной имеется инициализирующее значение, то при каждом прохождении этой точки инициализация выполняется заново.

Но в некоторых ситуациях требуется сохранять значение между вызовами функций. Вообще говоря, задача может быть решена объявлением глобальной переменной, но такая переменная не находится под исключительным контролем функции. С и C++ позволяют создавать в функциях статические объекты. Память для таких объектов выделяется не в стеке, а в статической области данных программы. Статический объект инициализируется только один раз при первом вызове функции, а затем сохраняет свое значение между вызовами. Например, следующая функция при каждом вызове возвращает очередной символ из массива:

```
//: C10:StaticVariablesInfunctions.cpp
#include "../require.h"
#include <iostream>
using namespace std;

char oneChar(const char* charArray = 0) {
  static const char* s;
  if(charArray) {
    s = charArray;
    return *s;
  }
  else
    require(s, "un-initialized s");
  if(*s == '\0')
    return 0;
  return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
  // oneChar(); // Ошибка при вызове require()
  oneChar(a); // Инициализирует s значением a
  char c;
  while((c = oneChar()) != 0)
    cout << c << endl;
} ///:~
```

Переменная static char* s сохраняет свое значение между вызовами oneChar(), потому что ее память находится не в кадре стека данной функции, а в статической области данных программы. При вызове oneChar() с аргументом char* значение аргумента присваивается s, и функция возвращает первый символ массива. При каждом последующем вызове oneChar() *без аргумента* по умолчанию charArray присваивается 0; это означает, что функция должна продолжить выборку символов из ранее инициализированного значения s. Функция продолжает выдавать символы до тех

пор, пока не достигнет завершающего нулевого символа в символьном массиве. На этой стадии она перестает увеличивать указатель, чтобы не выйти за пределы массива.

Но что произойдет, если вызвать oneChar() без аргументов и без предварительной инициализации s? В определение s можно включить инициализирующее значение:

```
static char* s = 0;
```

Но при отсутствии инициализирующего значения для статической переменной встроенного типа компилятор гарантирует, что переменная будет инициализирована нулем (преобразованным к соответствующему типу) при запуске программы. Следовательно, при первом вызове oneChar() переменная s равна нулю. В представленном примере этот факт обнаруживается при проверке условия if(!s).

В данном случае инициализация s очень проста, но статические объекты (как и любые другие объекты) могут инициализироваться произвольными выражениями, включающими константы, а также объявленные ранее переменные или функции.

Учтите, что функция oneChar() крайне ненадежна в отношении многопоточности. При разработке функций, содержащих статические переменные, следует постоянно помнить о многопоточных аспектах доступа.

Статические объекты в функциях

Статические объекты пользовательских типов подчиняются тем же правилам, включая необходимость инициализации объекта. Тем не менее обнуление имеет смысл только для встроенных типов; пользовательские типы должны инициализироваться вызовом конструктора. Следовательно, если при определении статического объекта не указаны аргументы конструктора, то класс должен иметь конструктор по умолчанию. Пример:

```
//: C10:StaticObjectsInFunctions.cpp
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {} // По умолчанию
  ~X() { cout << "X::~X()" << endl; }
};

void f() {
  static X x1(47);
  static X x2; // Нужен конструктор по умолчанию
}

int main() {
  f();
} ///:~
```

Статические объекты типа X внутри функции f() могут инициализироваться либо списком аргументов конструктора, либо конструктором по умолчанию. Конструирование происходит при первом, и *только* при первом проходе определения.

Деструкторы статических объектов

Деструкторы статических объектов (то есть всех объектов со статическим хранением, а не только локальных статических объектов из предыдущего примера) вызываются при выходе из main() или при явном вызове стандартной библиотечной функции С exit(). В большинстве реализаций main() при завершении просто вызывает exit(). Это означает, что вызывать exit() в деструкторе рискованно — может возникнуть бесконечная рекурсия. Деструкторы статических объектов *не вызываются* при завершении программы библиотечной функцией abort().

Функция стандартной библиотеки С atexit() определяет действия, которые должны выполняться при выходе из main() (или при вызове exit()). Функции, зарегистрированные функцией atexit(), могут вызываться до вызова деструкторов всех сконструированных объектов перед выходом из main() (или вызовом exit()).

Статические объекты, как и обычные, уничтожаются в порядке, обратном порядку инициализации. Уничтожаются только объекты, которые ранее были сконструированы. К счастью, средства разработки С++ следят за порядком инициализации и за сконструированными объектами. Глобальные объекты всегда конструируются перед входом в main() и уничтожаются при выходе из main(), но с локальными статическими объектами дело обстоит иначе. Если функция с локальным статическим объектом не вызывалась в программе, то конструктор этого объекта не выполнялся, а следовательно, деструктор тоже выполняться не будет. Пример:

```
//: C10:StaticDestructors.cpp
// Деструкторы статических объектов
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // Трассировочный файл

class Obj {
  char c;
public:
  Obj(char cc) : c(cc) {
    out << "Obj::Obj() for " << c << endl;
  }
  ~Obj() {
    out << "Obj::~Obj() for " << c << endl;
  }
};

Obj a('a'); // Глобальный объект (статическое хранение)
// Конструктор и деструктор вызываются всегда

void f() {
  static Obj b('b');
}

void g() {
  static Obj c('c');
}

int main() {
  out << "inside main()" << endl;
  f(); // Вызов конструктора для статического объекта b
  // g() не вызывается
  out << "leaving main()" << endl;
} ///:~
```

В объекте Obj переменная char c используется для идентификации, чтобы конструктор и деструктор могли выводить информацию об объекте, с которым они работают. Объект Obj a является глобальным, поэтому его конструктор всегда вызывается перед входом в main(), но конструкторы статического объекта Obj b внутри функции f() и статического объекта Obj c внутри функции g() вызываются только при вызове этих функций.

Чтобы показать, какие конструкторы и деструкторы вызываются в программе, мы вызываем только функцию f(). Результат выполнения программы:

```
Obj::Obj() for a
inside main()
Obj::Obj() for b
leaving main()
Obj::~Obj() for b
Obj::~Obj() for a
```

Конструктор a вызывается перед входом в main(), а конструктор b вызывается вследствие вызова функции f(). При выходе из main() деструкторы сконструированных объектов вызываются в порядке, обратном порядку конструирования. Следовательно, если в программе *была* вызвана функция g(), то порядок вызова деструкторов b и c зависит от того, какая функция вызывалась первой — f() или g().

Также обратите внимание на то, что объект out трассировочного файлового потока ofstream тоже является статическим объектом. Поскольку этот объект определяется вне всех функций, он находится в области статического хранения. Очень важно, что его определение (в отличие от объявления extern) находится в самом начале файла перед первым обращением к out. В противном случае в программе использовался бы неинициализированный объект.

В C++ конструкторы глобальных статических объектов вызываются перед входом в main(), поэтому в вашем распоряжении оказывается простой и переносимый способ выполнения кода перед входом в main() или после выхода из main(). В языке С для этого приходилось подолгу разбираться в ассемблерном коде запуска, используемом конкретным разработчиком компилятора.

Управление связыванием

Как правило, любые имена, имеющие *файловую видимость* (то есть не определенные внутри класса или функции), видны всем единицам трансляции в программе. Также часто встречается термин *внешнее связывание*, потому что в момент компоновки имя доступно для компоновки из всех единиц, внешних по отношению к данной. Глобальные переменные и обычные функции обладают внешним связыванием.

Но в некоторых ситуациях видимость имени приходится ограничивать. Допустим, вам понадобилась переменная с файловой видимостью, которая может использоваться всеми функциями файла, но при этом переменная должна оставаться невидимой для внешних функций, или вы хотите предотвратить конфликты с именами внешних идентификаторов.

Объекты или функции с файловой видимостью, объявленные с ключевым словом static, являются локальными по отношению к своей единице трансляции (в контексте книги — к cpp-файлу, в котором находится объявление). Их имена обладают *внутренним связыванием*; это означает, что их использование в других единицах трансляции не вызывает конфликтов.

Одно из преимуществ внутреннего связывания состоит в том, что имена можно размещать в заголовочном файле, не беспокоясь о возможных конфликтах компоновки. Для имен, которые обычно размещаются в заголовочных файлах, например определений констант и подставляемых функций, по умолчанию используется внутреннее связывание (впрочем, для констант это справедливо только в C++; в C для них по умолчанию имеет место внешнее связывание). Учтите, что связывание определено только для элементов, обладающих адресами на стадии компоновки/загрузки; следовательно, объявления классов и локальные переменные не обладают связыванием.

Путаница

Следующий пример показывает, как два смысла ключевого слова static могут пересекаться. По умолчанию все глобальные объекты имеют статический класс хранения, поэтому если в программе встречается показанное ниже определение, то память для a выделяется в статической области данных программы, а инициализация a выполняется один раз перед входом в main():

```
int a = 0;
```

Кроме того, переменная a является глобальной по отношению ко всем единицам трансляции. В отношении видимости противоположностью static (видимость только в данной единице трансляции) является ключевое слово extern, которое означает, что имя видимо во всех единицах трансляции. Следовательно, предыдущее определение эквивалентно такому:

```
extern int a = 0;
```

Если заменить его следующим определением, вы всего лишь измените видимость переменной, то есть a будет обладать внутренним связыванием:

```
static int a = 0;
```

Класс хранения при этом не меняется, то есть объект хранится в статической области данных независимо от того, определялся он с ключевым словом static или extern.

Однако с переходом к локальным переменным ключевое слово static изменяет уже не видимость, а класс хранения объекта.

Если переменная, на первый взгляд похожая на локальную, объявляется с ключевым словом extern, это означает, что память была выделена в другом месте (то есть переменная в действительности является глобальной для данной функции). Пример:

```
//: C10:LocalExtern.cpp
//{L} LocalExtern2
#include <iostream>

int main() {
  extern int i;
  std::cout << i;
} ///:~

//: C10:LocalExtern2.cpp {O}
int i = 5;
///:~
```

С именами функций (не являющихся членами классов) ключевые слова static и extern могут лишь изменять видимость, поэтому следующие объявления эквивалентны:

```
extern void f():
void f():
```

С другой стороны, показанное ниже объявление означает, что функция f() видима только внутри данной единицы трансляции:

```
static void f():
```

Иногда такую функцию называют «функцией, статической в границах файла».

Другие определители класса хранения

Ключевые слова static и extern используются очень часто. Наряду с ними существуют еще два спецификатора класса хранения, которые встречаются реже. Так, спецификатор auto, который на практике почти не используется, сообщает компилятору, что переменная является локальной. Auto является сокращением от слова «автоматический» (речь идет об автоматическом выделении компилятором памяти под переменную). Но компилятор всегда может выяснить это обстоятельство по контексту определения переменной, поэтому ключевое слово auto является избыточным.

Регистровые (register) переменные представляют собой локальные (auto) переменные, которые так интенсивно используются в программе, что компилятору рекомендуется по возможности хранить их в регистре (то есть регистровые переменные являются средством оптимизации). Разные компиляторы по-разному реагируют на эту рекомендацию и могут проигнорировать ее. Если в программе используется адрес переменной, спецификатор register почти наверняка будет проигнорирован. Постарайтесь обходиться без ключевого слова register, потому что компилятор обычно справляется с оптимизацией лучше программиста.

Пространства имен

Хотя видимость имен может ограничиваться при помощи классов, имена глобальных функций, глобальных переменных и классов по-прежнему входят в единую глобальную совокупность имен. Ключевое слово static позволяет до определенной степени контролировать имена за счет назначения переменным и функциям внутреннего связывания (то есть их объявления статическими в границах файла). Однако в больших проектах недостаточный контроль над глобальным пространством имен порождает немало проблем. Стремясь решить эти проблемы для своих классов, разработчики библиотек часто используют длинные сложные имена, которые вряд ли станут причиной конфликтов, но тогда вам придется возиться с вводом этих имен (хотя задача часто упрощается при помощи ключевого слова typedef). Это не слишком элегантное решение, к тому же не поддерживаемое на уровне языка.

C++ позволяет разбивать глобальную совокупность имен на части, с которыми удобнее работать, — *пространства имен*. По аналогии с ключевыми словами class, struct, enum и union, ключевое слово namespace выделяет имена своих членов

в отдельное пространство. Но если другие ключевые слова имеют дополнительный смысл, то namespace используется только для определения нового пространства имен.

Создание пространств имен

Синтаксис создания пространства имен имеет много общего с синтаксисом создания класса:

```
//: C10:MyLib.cpp
namespace MyLib {
  // Объявления
}
int main() {} ///:~
```

Определение создает новое пространство имен, содержащее объявления из вложенного блока. Однако эти пространства имен принципиально отличаются от пространств, создаваемых ключевыми словами class, struct, union и enum.

- Определения пространств имен находятся либо в глобальной области видимости, либо внутри другого пространства имен.
- После закрывающей фигурной скобки определения пространства имен не обязательно присутствие символа точки с запятой (;).
- Определение пространства имен может быть «продолжено» в других заголовочных файлах. При этом используется синтаксис, который бы для класса воспринимался как переопределение:

```
//: C10:Header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
  extern int x:
  void f():
  // ...
}

#endif // HEADER1_H ///:~
//: C10:Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Добавление новых имен в MyLib
namespace MyLib { // НЕ ЯВЛЯЕТСЯ переопределением!
  extern int y:
  void g():
  // ...
}

#endif // HEADER2_H ///:~
//: C10:Continuation.cpp
#include "Header2.h"
int main() {} ///:~
```

- Пространствам имен можно назначать *псевдонимы*, чтобы вам не приходилось вручную вводить неудобные имена, придуманные разработчиками библиотек:

```
//: C10:BobsSuperDuperLibrary.cpp
namespace BobsSuperDuperLibrary {
  class Widget { /* ... */ };
  class Poppit { /* ... */ };
  // ...
}
// Слишком длинное имя! Назначаем псевдоним:
namespace Bob = BobsSuperDuperLibrary;
int main() {} ///:~
```

- Пространства имен, в отличие от классов, не позволяют создавать экземпляры.

Анонимные пространства имен

Каждая единица трансляции содержит анонимное пространство имен, для добавления которого используется ключевое слово namespace без идентификатора:

```
//: C10:UnnamedNamespaces.cpp
namespace {
  class Arm  { /* ... */ };
  class Leg  { /* ... */ };
  class Head { /* ... */ };
  class Robot {
    Arm arm[4];
    Leg leg[16];
    Head head[3];
    // ...
  } xanthan;
  int i, j, k;
}
int main() {} ///:~
```

Элементы этого пространства имен автоматически доступны в текущей единице трансляции и не требуют уточнения. Компилятор гарантирует уникальность анонимного пространства имен для каждой единицы трансляции. Для локальных имен, находящихся в анонимном пространстве имен, не нужно назначать внутреннее связывание, объявляя их с ключевым словом static.

В C++ концепция файловой статичности считается устаревшей, вместо нее рекомендуется использовать анонимные пространства имен.

Друзья

В пространство имен можно *ввести* дружественный элемент, объявив его с ключевым словом friend во внутреннем классе:

```
//: C10:FriendInjection.cpp
namespace Me {
  class Us {
    // ...
    friend void you();
  };
}
int main() {} ///:~
```

Теперь функция you() входит в пространство имен Me.

Если функция объявляется дружественной внутри класса, находящегося в глобальном пространстве имен, то она вводится в глобальное пространство имен.

Использование пространств имен

На идентификатор, принадлежащий пространству имен, можно сослаться тремя способами:

- явно задать пространство имен оператором уточнения области видимости (::);
- импортировать все имена *директивой* using;
- импортировать отдельные имена *объявлением* using.

Явное задание пространства имен

На любой идентификатор, принадлежащий некоторому пространству имен, можно сослаться при помощи оператора :: (по аналогии со ссылками на члены класса):

```
//: C10:ScopeResolution.cpp
namespace X {
  class Y {
    static int i;
  public:
    void f();
  };
  class Z;
  void func();
}
int X::Y::i = 9;
class X::Z {
  int u, v, w;
public:
  Z(int i);
  int g();
};
X::Z::Z(int i) { u = v = w = i; }
int X::Z::g() { return u = v = w = 0; }
void X::func() {
  X::Z a(1);
  a.g();
}
int main(){} ///:~
```

Обратите внимание: определение X::Y::i в равной степени может относиться к переменной i класса Y, вложенного в класс X, а не принадлежащего пространству имен X.

В этом отношении пространства имен очень похожи на классы.

Директива using

Вводить полные (с уточнениями) идентификаторы из пространства имен быстро надоедает, поэтому в языке появилось ключевое слово using, позволяющее импортировать сразу все пространство имен. В сочетании с ключевым словом namespace оно образует *директиву* using. После выполнения директивы using доступ к именам осуществляется так, словно они принадлежат ближайшему вмещающему пространству имен, то есть без уточнения. Рассмотрим простое пространство имен:

```
//: C10:NamespaceInt.h
#ifndef NAMESPACEINT_H
#define NAMESPACEINT_H
namespace Int {
```

```
  enum sign { positive, negative };
  class Integer {
    int i;
    sign s;
  public:
    Integer(int ii = 0)
      : i(ii),
        s(i >= 0 ? positive : negative)
    {}
    sign getSign() const { return s; }
    void setSign(sign sgn) { s = sgn; }
    // ...
  };
}
#endif // NAMESPACEINT_H ///:~
```

При помощи директивы using можно, например, перенести все имена из Int в другое пространство имен:

```
//: C10:NamespaceMath.h
#ifndef NAMESPACEMATH_H
#define NAMESPACEMATH_H
#include "NamespaceInt.h"
namespace Math {
  using namespace Int;
  Integer a, b;
  Integer divide(Integer, Integer);
  // ...
}
#endif // NAMESPACEMATH_H ///:~
```

В следующем примере все имена Int импортируются в функцию, а доступ к ним производится только внутри этой функции:

```
//: C10:Arithmetic.cpp
#include "NamespaceInt.h"
void arithmetic() {
  using namespace Int;
  Integer x;
  x.setSign(positive);
}
int main(){} ///:~
```

Без директивы using все имена из пространства Int пришлось бы уточнять оператором уточнения области видимости (::).

У директивы using есть одна особенность, которая на первый взгляд выглядит противоестественно. Видимость имен, импортированных директивой using, ограничивается тем блоком, в котором находится директива. Однако вы можете переопределять импортированные имена так, словно они были объявлены глобально по отношению к этому блоку:

```
//: C10:NamespaceOverriding1.cpp
#include "NamespaceMath.h"
int main() {
  using namespace Math;
  Integer a; // Скрывает Math::a;
  a.setSign(negative);
  // Теперь для обращения к Math::a
  // необходимо уточнение:
  Math::a.setSign(positive);
} ///:~
```

Предположим, у нас имеется второе пространство имен, содержащее некоторые имена из namespace Math:

```
//: C10:NamespaceOverriding2.h
#ifndef NAMESPACEOVERRIDING2_H
#define NAMESPACEOVERRIDING2_H
#include "NamespaceInt.h"
namespace Calculation {
  using namespace Int;
  Integer divide(Integer, Integer);
  // ...
}
#endif // NAMESPACEOVERRIDING2_H ///:~
```

Поскольку это пространство также импортируется директивой using, существует опасность конфликта. Тем не менее неоднозначность возникает в точке *использования* имени, а не в точке выполнения директивы using:

```
//: C10:OverridingAmbiguity.cpp
#include "NamespaceMath.h"
#include "NamespaceOverriding2.h"
void s() {
  using namespace Math;
  using namespace Calculation;
  // Все хорошо, пока не встретится команда:
  //! divide(1, 2); // Неоднозначность
}
int main() {} ///:~
```

Таким образом, в программу можно импортировать несколько пространств с конфликтующими именами, используя директивы using, и это не обязательно автоматически приведет к конфликту.

Объявление using

Также существует возможность импортирования отдельных имен в текущую область видимости при помощи объявления using. Если директива using интерпретирует имена как глобальные, то объявление using позволяет объявлять имена внутри текущей области видимости. Следовательно, таким образом можно переопределять имена, импортированные директивой using:

```
//: C10:UsingDeclaration.h
#ifndef USINGDECLARATION_H
#define USINGDECLARATION_H
namespace U {
  inline void f() {}
  inline void g() {}
}
namespace V {
  inline void f() {}
  inline void g() {}
}
#endif // USINGDECLARATION_H ///:~

//: C10:UsingDeclaration1.cpp
#include "UsingDeclaration.h"
void h() {
  using namespace U; // Директива using
  using V::f; // Объявление using
```

```
  f(); // Вызов V::f();
  U::f(); // Для вызова необходимо полное уточнение
}
int main() {} ///:~
```

Объявление using всего лишь сообщает полностью уточненное имя идентификатора, но не содержит информации о типе. Значит, если пространство имен содержит набор перегруженных функций с одинаковыми именами, объявление using позволяет объявить сразу все функции из этого набора.

Объявление using может находиться везде, где может находиться обычное объявление. Объявление using похоже на обычное объявление во всех отношениях, кроме одного: так как в объявлении using отсутствует список аргументов, оно может привести к перегрузке функции с теми же типами аргументов (что не допускается при стандартной перегрузке). Тем не менее эта неоднозначность проявится лишь в точке использования, а не объявления.

Объявления using также могут находиться внутри пространств имен. Там они делают то же самое, что и в любом другом месте, то есть импортируют внешние имена:

```
//: C10:UsingDeclaration2.cpp
#include "UsingDeclaration.h"
namespace Q {
  using U::f;
  using V::g;
  // ...
}
void m() {
  using namespace Q;
  f(); // Вызывает U::f();
  g(); // Вызывает V::g();
}
int main() {} ///:~
```

Объявление using создает псевдоним, позволяющий объявлять одну функцию в нескольких пространствах имен. Даже если импортирование разных пространств имен приводит к многократному объявлению одной функции, программа работает нормально — в ней не возникает никаких неоднозначностей или конфликтов.

Управление пространствами имен

Некоторые из описанных выше правил поначалу обескураживают, особенно если у вас сложилось впечатление, будто вам придется постоянно работать с ними. Но на практике обычно бывает достаточно очень простых применений пространств имен; нужно лишь хорошо понимать, как они работают. Главное — помните, что при включении в программу глобальной директивы using (конструкция using namespace вне какой-либо области видимости) вы предоставляете доступ к пространству имен в рамках этого файла. В файлах реализации (cpp-файлах) это обычно допустимо, поскольку директива using продолжает действовать только до конца компиляции этого файла. На другие файлы она не распространяется, поэтому управление пространствами имен может осуществляться на уровне отдельных файлов. Например, если в некотором файле реализации возникнет конфликт имен из-за обилия директив using, достаточно изменить этот файл и устранить конфликт посредством полного уточнения имен или объявлениями using. Другие файлы реализации остаются неизменными.

С заголовочными файлами дело обстоит совершенно иначе. Глобальные директивы using почти никогда не должны размещаться в заголовочных файлах, потому что любой другой файл, включивший данный заголовок, тоже получит свободный доступ к пространству имен (а заголовочные файлы могут включать другие заголовочные файлы).

Итак, в заголовочных файлах следует использовать либо полные уточнения, либо директивы и объявления using с явным указанием пространства. Такой подход применяется в примерах, приведенных в книге. Соблюдение этого правила предотвращает «загрязнение» глобального пространства имен и возврат к прежним временам, когда в C++ еще не поддерживались пространства имен.

Статические члены в C++

В некоторых ситуациях требуется, чтобы все объекты класса совместно работали с одним экземпляром переменной. В C для этой цели использовались глобальные переменные, но такой подход недостаточно надежен. Глобальные данные могут изменяться всеми желающими, а в больших проектах они также порождают конфликты имен. В идеальном случае данные должны храниться как глобальные, но при этом быть скрытыми внутри класса и четко связываться с ним.

Задача решается объявлением статических переменных внутри класса. Статической переменной всегда выделяется только одна область памяти независимо от того, сколько объектов класса будет создано в программе. Все объекты используют общую статическую область данных при работе с переменной, что позволяет им «обмениваться информацией» друг с другом. Но при этом статическая переменная принадлежит классу; ее имя находится в области видимости этого класса, и она может быть объявлена открытой (public), закрытой (private) или защищенной (protected).

Определение статических переменных классов

Поскольку статические переменные всегда представляются одной областью памяти независимо от количества объектов, эта память должна выделяться в одной точке программы; компилятор не выделит ее за вас. Если статическая переменная объявлена, но не определена, компоновщик выдаст сообщение об ошибке.

Определение должно находиться за пределами класса (внутренние определения не разрешены), и при этом в единственном экземпляре. По этой причине определения часто размещают в файле реализации класса. Синтаксис вызывает проблемы у некоторых программистов, но в действительности он вполне логичен. Допустим, статическая переменная объявляется внутри класса следующим образом:

```
class A {
  static int i;
public:
  // ...
};
```

Выделение памяти для статической переменной класса должно выполняться в файле реализации примерно так:

```
int A::i = 1;
```

Определение обычной глобальной переменной выглядело бы так:

```
int i = 1;
```

Однако для статической переменной дополнительно используется оператор уточнения области видимости (::) и имя класса.

На первый взгляд происходит нечто странное: переменная A::i объявлена закрытой, но ее значение вроде бы свободно задается где-то в программе. Разве это не нарушает механизм защиты данных? Нет, такое присваивание абсолютно безопасно по двум причинам. Во-первых, такая инициализация разрешена только в одном месте — в определении. В самом деле, если бы статическая переменная была объектом с конструктором, то вы бы вызвали конструктор вместо присваивания оператором =. Во-вторых, пользователь не сможет выполнить определение повторно — компоновщик выдаст сообщение об ошибке. Наконец, создатель класса *должен* предоставить определение, или программа не пройдет компоновку. Следовательно, определение выполняется только один раз и только создателем класса.

Все выражение инициализации статической переменной принадлежит к области видимости класса. Пример:

```
//: C10:Statinit.cpp
// Видимость инициализирующего значения статической переменной
#include <iostream>
using namespace std;

int x = 100;

class WithStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "WithStatic::x = " << x << endl;
    cout << "WithStatic::y = " << y << endl;
  }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x, НЕ ::x

int main() {
  WithStatic ws;
  ws.print();
} ///:~
```

Уточнение WithStatic:: распространяет область видимости класса WithStatic на все определение.

Инициализация статических массивов

В главе 8 были представлены статические константные (static const) переменные, позволяющие определять константные значения в теле класса. Также имеется возможность создания массивов статических объектов (как константных, так и неконстантных). Синтаксис выглядит вполне логично:

```
//: C10:StaticArray.cpp
// Инициализация статических массивов в классах
```

```
class Values {
  // Статические константы инициализируются на месте:
  static const int scSize = 100;
  static const long scLong = 100;
  // Автоматический подсчет элементов работает и для статических массивов.
  // Массивы, нецелые и неконстантные статические переменные
  // требуют внешней инициализации:
  static const int scInts[];
  static const long scLongs[];
  static const float scTable[];
  static const char scLetters[];
  static int size;
  static const float scFloat;
  static float table[];
  static char letters[];
};

int Values::size = 100;
const float Values::scFloat = 1.1;

const int Values::scInts[] = {
  99, 47, 33, 11, 7
};

const long Values::scLongs[] = {
  99, 47, 33, 11, 7
};

const float Values::scTable[] = {
  1.1, 2.2, 3.3, 4.4
};

const char Values::scLetters[] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

float Values::table[4] = {
  1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

int main() { Values v; } ///:~
```

Для целочисленных статических констант допускается определение внутри класса, но во всех остальных случаях (включая массивы целочисленных типов, даже статические константные) для членов класса необходимо предоставить отдельное внешнее определение. Эти определения обладают внутренним связыванием, поэтому могут размещаться в заголовочных файлах. Для инициализации статических массивов используется тот же синтаксис, что и для любых других агрегатов, включая автоматический подсчет элементов по списку инициализирующих значений.

Также допускается создание статических константных объектов классов и массивов таких объектов. Впрочем, они не могут инициализироваться в синтаксисе

«подставляемых определений», разрешенном для статических констант целочисленных встроенных типов:

```
//: C10:StaticObjectArrays.cpp
// Статические массивы объектов классов
class X {
  int i;
public:
  X(int ii) : i(ii) {}
};

class Stat {
  // Не работает, как бы нам ни хотелось:
//!  static const X x(100);
  // Константные и неконстантные статические объекты классов
  // должны инициализироваться во внешних определениях:
  static X x2;
  static X xTable2[];
  static const X x3;
  static const X xTable3[];
};

X Stat::x2(100);

X Stat::xTable2[] = {
  X(1), X(2), X(3), X(4)
};

const X Stat::x3(100);

const X Stat::xTable3[] = {
  X(1), X(2), X(3), X(4)
};

int main() { Stat v; } ///:~
```

Константные и неконстантные статические массивы объектов классов инициализируются аналогичным образом с использованием обычного синтаксиса статических определений.

Вложенные и локальные классы

Статические переменные без труда определяются в классах, вложенных в другие классы. Определения таких переменных очевидны и интуитивно понятны — при указании имен просто задается дополнительный уровень уточнения. Тем не менее локальные классы (то есть классы, определяемые внутри функций) не могут содержать статических переменных:

```
//: C10:Local.cpp
// Статические переменные и локальные классы
#include <iostream>
using namespace std;

// Вложенные классы МОГУТ содержать статические переменные:
class Outer {
  class Inner {
    static int i; // OK
  };
```

```
};

int Outer::Inner::i = 47;

// Локальные классы не могут содержать статических переменных:
void f() {
  class Local {
  public:
//! static int i;  // Ошибка
    // (Как определить i?)
  } x;
}

int main() { Outer x; f(); } ///:~
```

Проблемы со статическими переменными в локальных классах проявляются немедленно: как описать переменную в файловой области видимости для ее определения? Впрочем, на практике локальные классы используются крайне редко.

Статические функции классов

В классе также могут определяться статические функции, которые, как и статические переменные, относятся к классу в целом, а не к отдельному объекту класса. Вместо того чтобы определять глобальную функцию, которая существует в глобальном или локальном пространстве имен и «загромождает» его, следует просто внести функцию в класс.

Статические функции могут вызываться с обычным синтаксисом (с применением операторов . или ->) для конкретных объектов. Но чаще статические функции вызываются не для конкретного объекта, а для класса в целом с применением оператора уточнения области видимости (::):

```
//: C10:SimpleStaticMemberFunction.cpp
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

Встречая статическую функцию в классе, помните, что разработчик класса задумывал эту функцию как концептуально связанную с классом в целом.

Статические функции класса не могут обращаться к обычным переменным класса, для них доступны только статические переменные. Кроме того, они могут вызывать только другие статические функции. Обычно при вызове любой функции незаметно передается адрес текущего объекта (this), но для статических функций указателя this не существует (именно поэтому статические функции не могут обращаться к обычным членам класса). Таким образом, статические функции работают чуть быстрее обычных функций классов, поскольку они обходятся без дополнительной передачи адреса через указатель this (что характерно для глобальных функций). В то же время сохраняются все преимущества от определения функций внутри класса.

Для переменной класса ключевое слово static означает, что все объекты класса работают с одним экземпляром этой переменной (то есть общей областью памя-

ти). Такая интерпретация соответствует определению статических объектов *внутри* функции, когда для всех вызовов функции существует только один экземпляр локальной переменной.

В следующем примере используются как статические переменные, так и статические функции класса:

```
//: C10:StaticMemberFunctions.cpp
class X {
  int i;
  static int j;
public:
  X(int ii = 0) : i(ii) {
     // Нестатические функции класса могут обращаться
     // к статическим функциям и данным:
    j = i;
  }
  int val() const { return i; }
  static int incr() {
    //! i++; // Ошибка: статическая функция класса
    // не может обратиться к нестатической переменной
    return ++j;
  }
  static int f() {
    //! val(); // Ошибка: статическая функция класса
    // не может вызвать нестатическую функцию
    return incr(); // OK -- вызывается статическая функция
  }
};

int X::j = 0;

int main() {
  X x;
  X* xp = &x;
  x.f();
  xp->f();
  X::f(); // Работает только со статическими членами
} ///:~
```

Для статических функций класса указатель this не определен, поэтому они не могут обращаться к нестатическим переменным класса или вызывать нестатические функции класса.

В функции main() при обращениях к статическим членам классов может использоваться обычный синтаксис (операторы . или ->), связывающий функцию с конкретным объектом, но также допускается обращение без указания объекта при помощи имени класса и оператора уточнения области видимости (::). Как отмечалось ранее, это связано с тем, что статическая переменная ассоциируется не с конкретным объектом, а с классом в целом.

А теперь одна интересная особенность: механизм инициализации статических объектов-членов класса позволяет размещать статические переменные класса *внутри* этого класса. Следующий пример допускает существование только одного объекта типа Egg, для чего конструктор объявляется закрытым. Вы можете обращаться к этому объекту, но не можете создавать новые объекты Egg:

```
//: C10:Singleton.cpp
// Статическая переменная того же типа
```

```
// гарантирует существование только одного экземпляра типа Egg.
// Эта идиома также называется "синглетом".
#include <iostream>
using namespace std;

class Egg {
  static Egg e;
  int i;
  Egg(int ii) : i(ii) {}
  Egg(const Egg&); // Предотвращает конструирование копий
public:
  static Egg* instance() { return &e; }
  int val() const { return i; }
};

Egg Egg::e(47);

int main() {
//! Egg x(1); // Ошибка -- создать экземпляр Egg нельзя
  // Возможно только обращение к единственному существующему экземпляру:
  cout << Egg::instance()->val() << endl;
} ///:~
```

Переменная e инициализируется после завершения объявления класса, поэтому компилятор располагает всей информацией, необходимой для выделения памяти и вызова конструктора.

Чтобы полностью предотвратить создание любых других объектов, в класс было включено нечто новое: второй закрытый конструктор, называемый *копирующим конструктором*. В данный момент вы еще не поймете, почему это необходимо, поскольку копирующие конструкторы будут представлены лишь в следующей главе. А пока достаточно сказать, что если убрать копирующий конструктор из предыдущего примера, то программа сможет создавать объекты Egg следующим образом:

```
Egg e = *Egg::instance();
Egg e2(*Egg::instance());
```

В обоих случаях используется копирующий конструктор. Чтобы полностью исключить эту возможность, мы объявляем копирующий конструктор закрытым (определение не требуется, потому что конструктор никогда не вызывается). Большая часть следующей главы посвящена копирующим конструкторам, и вскоре вам все станет ясно.

Порядок инициализации статических объектов

В конкретной единице трансляции статические объекты гарантированно инициализируются в том порядке, в котором они определяются в этой единице трансляции. Уничтожение объектов заведомо происходит в порядке, обратном порядку инициализации.

Однако не существует никаких гарантий относительно порядка инициализации статических объектов *в разных* единицах трансляции, а в языке не предусмотрена возможность задания этого порядка. Подобная ситуация может стать источником серьезных проблем. Ниже приведен катастрофический пример, который

«подвешивает» примитивные операционные системы и «убивает» процессы в более совершенных системах. Пусть один из файлов содержит такой фрагмент:

```
// Первый файл
#include <fstream>
std::ofstream out("out.txt");
```

В другом файле объект out используется с инициализирующим значением:

```
// Второй файл
#include <fstream>
extern std::ofstream out;
class Oof {
public:
  Oof() { std::out << "ouch"; }
} oof;
```

Иногда программа будет работать, иногда — нет. Если среда программирования построит программу так, что первый файл будет инициализироваться раньше второго, то проблем не будет. Но если сначала будет инициализироваться второй файл, то окажется, что работа конструктора Oof зависит от несконструированного объекта out. Последствия непредсказуемы.

Проблема возникает только с инициализирующими значениями статических объектов, которые зависят друг от друга. Статические объекты в единице трансляции инициализируются перед первым вызовом функций этой единицы, но это может произойти после main(). Если статические объекты находятся в разных файлах, нельзя делать никаких предположений относительно порядка их инициализации.

Более тонкий пример имеется в ARM[1]. Допустим, в глобальной области видимости одного файла содержится такой фрагмент:

```
extern int y;
int x = y + 1;
```

В глобальной области видимости второго файла находится следующий фрагмент:

```
extern int x;
int y = x + 1;
```

Для всех статических объектов механизм компоновки и загрузки гарантирует, что перед выполнением динамической инициализации, определяемой программистом, статические данные будут инициализированы нулями. В предыдущем примере обнуление памяти, занимаемой объектом fstream out, не имеет особого смысла, поэтому содержимое объекта остается неопределенным до вызова конструктора. Тем не менее для встроенных типов инициализация нулями *имеет* смысл, и если файлы будут инициализироваться в приведенном порядке, то переменная y сначала статически инициализируется нулем, переменная x становится равной 1, после чего переменная y динамически инициализируется значением 2. Но если инициализация пойдет в обратном порядке, то переменная x статически инициализируется нулем, переменная y динамически инициализируется единицей, так что переменная x становится равной 2.

Если вы забудете об этом, возможна крайне неприятная ситуация: программа, в которой имеют место зависимости при статической инициализации, нормально

[1] «The Annotated C++ Reference Manual», Bjarne Stroustrup and Margaret Ellis, Addison-Wesley, 1990.

работает на одной платформе, но при переходе в другую среду компиляции она вдруг загадочным образом перестает работать.

Существуют три основных подхода к решению проблемы.

- Не создавайте себе проблем. Лучшее решение — обойтись без зависимостей при инициализации статических объектов.
- Если зависимости абсолютно необходимы, разместите определения важнейших статических объектов в одном файле. Расставляя определения в нужном порядке, вы сможете управлять их инициализацией без нарушения переносимости.
- Если вы твердо убеждены, что распределение статических объектов по единицам трансляции неизбежно (как в случае с библиотекой, когда действия программиста, использующего библиотеку, не поддаются контролю), существуют два программных решения.

Первое решение

Этот прием впервые был использован Джерри Шварцем (Jerry Schwarz) в работе над библиотекой потоков ввода-вывода (так как потоки cin, cout и cerr являются статическими и находятся в отдельном файле). Вообще говоря, это решение хуже второго, но оно используется уже давно, а программы, в которых оно задействовано, встречаются довольно часто; следовательно, вы должны по крайней мере понимать, как оно работает.

Для работы этого решения в заголовочный файл библиотеки включается дополнительный класс, отвечающий за динамическую инициализацию статических объектов библиотеки. Рассмотрим простой пример:

```
//: C10:Initializer.h
// Статическая инициализация
#ifndef INITIALIZER_H
#define INITIALIZER_H
#include <iostream>
extern int x; // Объявления, но не определения
extern int y;

class Initializer {
  static int initCount;
public:
  Initializer() {
    std::cout << "Initializer()" << std::endl;
    // Initialize first time only
    if(initCount++ == 0) {
      std::cout << "performing initialization"
                << std::endl;
      x = 100;
      y = 200;
    }
  }
  ~Initializer() {
    std::cout << "~Initializer()" << std::endl;
    // Clean up last time only
    if(--initCount == 0) {
      std::cout << "performing cleanup"
```

```
            << std::endl;
        // Вся необходимая зачистка
      }
    }
  };

// Следующая строка создает один объект в каждом файле,
// включающем Initializer.h, причем этот объект виден
// только в границах файла:
static Initializer init;
#endif // INITIALIZER_H ///:~
```

Объявления x и y всего лишь заявляют о существовании объектов, но не выделяют для них память. С другой стороны, определение Initializer init выделяет память для объекта в каждом файле, в который включается заголовочный файл. Но поскольку имя является статическим (причем в контексте управления видимостью, а не выделения памяти; по умолчанию память выделяется на уровне файла), оно видимо только в границах этой единицы трансляции, и компоновщик не будет жаловаться на повторное определение.

Пример файла с определениями x, y и initCount:

```
//: C10:InitializerDefs.cpp {O}
// Определения Initializer.h
#include "Initializer.h"
// Следующие переменные принудительно обнуляются
// в процессе статической инициализации:
int x;
int y;
int Initializer::initCount;
///:~
```

(Разумеется, при включении заголовка в этот файл также включается экземпляр init, статический в границах файла.) Теперь допустим, что пользователь библиотеки создал еще два файла. Первый файл:

```
//: C10:Initializer.cpp {O}
// Статическая инициализация
#include "Initializer.h"
///:~
```

Второй файл:

```
//: C10:Initializer2.cpp
//{L} InitializerDefs Initializer
// Статическая инициализация
#include "Initializer.h"
using namespace std;

int main() {
  cout << "inside main()" << endl;
  cout << "leaving main()" << endl;
} ///:~
```

Теперь уже неважно, какая единица трансляции будет инициализироваться первой. При первой инициализации единицы трансляции, содержащей файл Initializer.h, значение initCount будет равно нулю, что приведет к выполнению инициализации (что в значительной степени зависит от обнуления статической области памяти перед проведением динамической инициализации). В остальных единицах трансляции переменная initCount будет отлична от нуля, поэтому инициализации не

произойдет. Зачистка производится в обратном порядке, и деструктор ~Initializer() гарантирует, что она будет выполнена только один раз.

В приведенном примере в качестве глобальных статических объектов использовались встроенные типы. Этот прием также подходит для объектов классов, но они должны быть динамически инициализированы классом Initializer. Одно из решений основано на создании классов без конструкторов и деструкторов, но содержащих функции инициализации и зачистки с другими именами. Впрочем, более распространенный подход заключается в создании указателей на объекты и вызове оператора new внутри конструктора Initializer().

Второе решение

Первое решение использовалось в течение довольно долгого времени, но потом кто-то (неизвестно, кто именно) предложил другую методику, более простую и наглядную. Вероятно, столь длительная задержка обусловлена сложностью C++.

Новая методика основана на том факте, что статические объекты внутри функций инициализируются в первый и единственный раз при вызове функции. Вспомните, что решаемая проблема связана не с тем, *когда* инициализируются статические объекты (это отдельная тема), а с определенным порядком выполнения инициализации.

Решение выглядит очень умно и изящно. Для любой зависимости статический объект помещается в функцию, возвращающую ссылку на этот объект. Обращение к статическому объекту возможно только через вызов функции, а если этому объекту потребуется обратиться к другим статическим объектам, от которых он зависит, он вызывает функции *этих* объектов. При первом вызове функции выполняется инициализация. Порядок статической инициализации заведомо будет правильным, поскольку он определяется архитектурой программы, а не выбирается произвольно на усмотрение компоновщика.

Для примера рассмотрим два класса, зависящих друг от друга. Первый класс содержит переменную bool, которая инициализируется только в конструкторе, поэтому мы можем определить, был ли конструктор вызван для статического экземпляра класса (статическая область данных обнуляется при запуске программы, и, если конструктор не вызывался, переменная bool будет равна false):

```
//: C10:Dependency1.h
#ifndef DEPENDENCY1_H
#define DEPENDENCY1_H
#include <iostream>

class Dependency1 {
  bool init;
public:
  Dependency1() : init(true) {
    std::cout << "Dependency1 construction"
              << std::endl;
  }
  void print() const {
    std::cout << "Dependency1 init: "
              << init << std::endl;
  }
};
#endif // DEPENDENCY1_H ///:~
```

Конструктор также анонсирует вызов. Функция print() выводит данные о состоянии объекта, чтобы программист мог убедиться в его инициализации.

Второй класс инициализируется на основании объекта первого класса, вследствие чего и возникает зависимость:

```
//: C10:Dependency2.h
#ifndef DEPENDENCY2_H
#define DEPENDENCY2_H
#include "Dependency1.h"

class Dependency2 {
  Dependency1 d1;
public:
  Dependency2(const Dependency1& dep1): d1(dep1){
    std::cout << "Dependency2 construction ";
    print();
  }
  void print() const { d1.print(); }
};
#endif // DEPENDENCY2_H ///:~
```

Конструктор сообщает о вызове и выводит данные о состоянии объекта d1. По результатам вывода можно проверить, был ли объект инициализирован на момент вызова конструктора.

Для демонстрации проблем, которые могут возникнуть в этой программе, в следующем файле определения статических объектов сначала задаются в обратном порядке. То же самое происходит тогда, когда компоновщик инициализирует объект Dependency2 раньше объекта Dependency1. Затем порядок определений изменяется, чтобы показать, что при «правильном» порядке программа работает нормально. Наконец, демонстрируется рассмотренное решение (второе).

Для получения удобочитаемых результатов в программе создается функция separator(). Но так как функция может вызываться глобально лишь в том случае, если она используется для инициализации переменной, separator() возвращает фиктивное значение, по которому инициализируется пара фиктивных глобальных переменных.

```
//: C10:Technique2.cpp
#include "Dependency2.h"
using namespace std;

// Функция возвращает значение, чтобы она могла вызываться
// как глобальное инициализирующее значение:
int separator() {
  cout << "---------------------" << endl;
  return 1;
}

// Имитация проблемы с зависимостью от порядка инициализации:
extern Dependency1 dep1;
Dependency2 dep2(dep1);
Dependency1 dep1;
int x1 = separator();

// Но если инициализация выполняется в обратном порядке,
// все работает нормально:
Dependency1 dep1b;
```

```
Dependency2 dep2b(dep1b);
int x2 = separator();

// Включение статических объектов в функции решает проблему:
Dependency1& d1() {
  static Dependency1 dep1;
  return dep1;
}

Dependency2& d2() {
  static Dependency2 dep2(d1());
  return dep2;
}

int main() {
  Dependency2& dep2 = d2();
} ///:~
```

Функции d1() и d2() представляют собой оболочки для статических экземпляров объектов Dependency1 и Dependency2. Теперь к статическим объектам можно обратиться лишь через вызов функций, что приводит к принудительному выполнению статической инициализации при первом вызове функции. Следовательно, инициализация гарантированно будет выполнена в правильном порядке. Чтобы убедиться в этом, достаточно запустить программу и ознакомиться с результатами.

А теперь рассмотрим типичную структуру программы, в которой используется эта методика. Поскольку статические объекты определяются в разных файлах (просто мы почему-либо вынуждены это делать; собственно, вся проблема возникает из-за определения статических объектов в разных файлах), оболочки также должны определяться в разных файлах. Но, кроме того, они должны быть объявлены в заголовочных файлах:

```
//: C10:Dependency1StatFun.h
#ifndef DEPENDENCY1STATFUN_H
#define DEPENDENCY1STATFUN_H
#include "Dependency1.h"
extern Dependency1& d1();
#endif // DEPENDENCY1STATFUN_H ///:~
```

Вообще говоря, ключевое слово extern является избыточным при объявлении функции. Второй заголовочный файл:

```
//: C10:Dependency2StatFun.h
#ifndef DEPENDENCY2STATFUN_H
#define DEPENDENCY2STATFUN_H
#include "Dependency2.h"
extern Dependency2& d2();
#endif // DEPENDENCY2STATFUN_H ///:~
```

Далее в файлах реализаций на место определений статических объектов подставляются определения оболочек:

```
//: C10:Dependency1StatFun.cpp {O}
#include "Dependency1StatFun.h"
Dependency1& d1() {
static Dependency1 dep1;
return dep1;
} ///:~
```

Вероятно, в файлах реализации также будет находиться другой код, непосредственно используемый в программе. Второй файл реализации выглядит так:

```
//: C10:Dependency2StatFun.cpp {O}
#include "Dependency1StatFun.h"
#include "Dependency2StatFun.h"
Dependency2& d2() {
static Dependency2 dep2(d1());
return dep2;
} ///:~
```

Итак, мы получаем два файла, которые могут компоноваться в любом порядке без нарушения очередности инициализации. Функции-оболочки предотвращают опасность неправильной инициализации:

```
//: C10:Technique2b.cpp
//{L} Dependency1StatFun Dependency2StatFun
#include "Dependency2StatFun.h"
int main() { d2(); } ///:~
```

Запустив эту программу, вы убедитесь, что статический объект Dependency1 всегда инициализируется перед статическим объектом Dependency2. Такое решение гораздо проще первого.

Возможно, у вас возникнет искушение оформить d1() и d2() в виде подставляемых функций в соответствующих заголовочных файлах, но этого делать определенно не следует. Подставляемые функции могут дублироваться во всех файлах, в которых они присутствуют, причем дублирование также *включает* определение статического объекта. А поскольку для подставляемых функций по умолчанию используется внутреннее связывание, это приведет к появлению нескольких статических объектов в разных единицах трансляции, что наверняка обернется новыми проблемами. Итак, для каждой функции-оболочки должно существовать только одно определение, а это означает, что оболочки не должны быть подставляемыми.

Альтернативные спецификации компоновки

Что делать, если вы пишете программу на C++ и хотите воспользоваться библиотекой C? Если объявить функцию C в следующем виде, компилятор C++ преобразует это имя в _f_int_char (или что-нибудь в этом роде), чтобы обеспечить возможность перегрузки функций и безопасной компоновки:

```
float f(int a, char b);
```

Однако компилятор C, которым была откомпилирована библиотека, наверняка *не украшал* это имя, поэтому в библиотеке будет использоваться внутреннее имя _f. А значит, в C++ компоновщик не сможет разрешить вызовы f().

В C++ эта проблема решается при помощи механизма *альтернативной спецификации компоновки*, основанного на перегрузке ключевого слова extern. После extern указывается строка с определением типа компоновки, затем следует само объявление:

```
extern "C" float(int a, char b);
```

Тем самым мы сообщаем компилятору, что функция f() должна компоноваться по правилам C, поэтому компилятор не должен украшать ее имя. Стандарт

поддерживает всего два типа спецификаций компоновки: "C" и "C++", но разработчики компиляторов могут предусмотреть аналогичную поддержку для других языков.

Если программа содержит группу объявлений с альтернативной компоновкой, заключите ее в фигурные скобки:

```
extern "C" {
float f(int a, char b);
double d(int a, char b);
}
```

Или для заголовочного файла:

```
extern "C" {
#include "MyHeader.h"
}
```

Многие разработчики компиляторов C++ включают в свои заголовочные файлы альтернативные спецификации компоновки как для C, так и для C++, поэтому вам не придется беспокоиться об этом.

Итоги

Ключевое слово static иногда вызывает недоразумения, потому что в одних ситуациях оно управляет выделением памяти, а в других — видимостью и компоновкой имен.

Пространства имен C++ предоставляют более совершенные и гибкие средства для управления именами в больших проектах.

Еще один способ управления именами в программе основан на использовании ключевого слова static в классах. Такие имена не конфликтуют с глобальными именами, а их видимость и доступ ограничиваются границами класса, что упрощает сопровождение кода.

Упражнения

1. Создайте функцию со статической переменной-указателем. Когда при вызове функции передается аргумент, он указывает на начало массива int. Если функция вызывается с нулевым аргументом (аргумент по умолчанию), она возвращает значение следующего элемента массива, пока не обнаружит элемент со значением –1 (признак конца массива). Протестируйте функцию в main().
2. Создайте функцию, которая при очередном вызове возвращает следующее число Фибоначчи. Добавьте аргумент типа bool, по умолчанию равный false. При передаче аргумента true функция возвращается к началу последовательности Фибоначчи. Протестируйте функцию в main().
3. Создайте класс, содержащий массив int. Размер массива должен задаваться переменной класса static const int. Добавьте переменную const int и выполните ее инициализацию в списке инициализирующих значений конструктора; сделайте конструктор подставляемым (inline). Добавьте переменную static int и выполните ее инициализацию конкретным значением. Добавьте

статическую функцию, которая выводит эту статическую переменную. Добавьте подставляемую функцию print(), которая выводит все содержимое массива и вызывает статическую функцию. Протестируйте класс в main().

4. Создайте класс Monitor, который отслеживает количество вызовов функции incident() этого класса. Добавьте функцию print(), выводящую количество вызовов. Теперь создайте глобальную функцию (не функцию класса), содержащую статический объект Monitor. При каждом вызове функция должна вызывать incident(), а затем выводить счетчик вызовов функцией print(). Протестируйте функцию в main().
5. Измените класс Monitor из упражнения 4 так, чтобы он позволял уменьшить значение счетчика вызовов функцией decrement(). Создайте класс Monitor2, который получает в аргументе конструктора указатель на Monitor1, сохраняет этот указатель и вызывает функции incident() и print(). Вызовите функции decrement() и print() в деструкторе Monitor2. Создайте статический объект Monitor2 внутри функции. Поэкспериментируйте с main(), попробуйте вызывать и не вызывать функцию и посмотрите, что произойдет с деструктором Monitor2.
6. Создайте глобальный объект Monitor2 и посмотрите, что произойдет.
7. Создайте класс с деструктором, который выводит сообщение и затем вызывает exit(). Создайте глобальный объект класса и посмотрите, что произойдет.
8. В примере StaticDesctructors.cpp поэкспериментируйте с очередностью вызова конструкторов и деструкторов, вызывая f() и g() внутри main() в разном порядке. Правильно ли это делает ваш компилятор?
9. Используя пример StaticDesctructors.cpp, проверьте, как организована стандартная обработка ошибок в вашей реализации. Для этого преобразуйте исходное определение out в объявление extern и разместите определение после определения переменной a (чей конструктор Obj передает информацию в out). Прежде чем запускать программу, убедитесь в том, что на компьютере не выполняются никакие важные задачи.
10. Докажите, что содержащиеся в заголовочных файлах переменные, статические в границах файлов, не конфликтуют при включении заголовка в разные cpp-файлы.
11. Создайте простой класс, который содержит переменную типа int, конструктор, инициализирующий int своим аргументом, и функцию print() для вывода int. Разместите класс в заголовочном файле и включите его в два cpp-файла. В одном cpp-файле создайте экземпляр своего класса, а во втором объявите этот идентификатор внешним (extern) и протестируйте его в main(). Не забудьте скомпоновать оба объектных файла, в противном случае компоновщик не найдет объект.
12. Сделайте экземпляр объекта из упражнения 11 статическим и убедитесь в том, что компоновщик не находит его.
13. Объявите функцию в заголовочном файле. Разместите определение функции в одном cpp-файле и вызовите его из main() во втором cpp-файле.

Откомпилируйте программу и убедитесь в том, что она работает. Измените определение так, чтобы функция была статической, и убедитесь в том, что компоновщик не находит ее.

14. Измените пример Volatile.cpp из главы 8 так, чтобы функция comm:isr() действительно могла работать как обработчик прерываний (обработчики прерываний должны вызываться без аргументов).
15. Напишите и откомпилируйте простую программу, в которой используются ключевые слова auto и register.
16. Создайте заголовочный файл, содержащий определение пространства имен. Объявите в пространстве имен несколько функций. Создайте второй заголовочный файл, который включает первый и дополняет пространство имен несколькими новыми объявлениями функций. Затем создайте cpp-файл, включающий второй заголовочный файл. Определите для пространства имен псевдоним, то есть другое, более короткое, имя. Внутри определения функции вызовите одну из функций с использованием оператора уточнения области видимости (::). Включите в отдельное определение функции директиву using, импортирующую пространство имен в область видимости этой функции, и убедитесь в том, что функции из пространства имен теперь могут вызываться без уточнения имен.
17. Создайте заголовочный файл с анонимным пространством имен. Включите этот заголовок в два разных cpp-файла и покажите, что с каждой единицей трансляции ассоциируется свое уникальное пространство имен.
18. Используя заголовочный файл из упражнения 17, покажите, что имена анонимного пространства имен автоматически доступны в единице трансляции без уточнения имен.
19. Измените пример FriendInjection.cpp, добавив в него определение дружественной функции и вызов этой функции в main().
20. В примере Arithmetic.cpp покажите, что действие директивы using не распространяется за пределы функций, в которых эта директива находится.
21. Исправьте ошибку в OverridingAmbiguity.cpp — сначала путем уточнения имен, затем с использованием объявления using, заставляющего компилятор выбрать одно из идентичных имен функций.
22. В двух заголовочных файлах создайте два пространства имен, в каждое из которых входит класс (содержащий только подставляемые определения); имена классов должны быть одинаковыми. Создайте cpp-файл, включающий оба заголовочных файла. Создайте функцию, внутри функции импортируйте оба пространства имен директивой using. Попробуйте создать объект класса и посмотрите, что произойдет. Сделайте директивы using глобальными (то есть выведите их из функций) и проверьте, изменит ли это что-нибудь. Исправьте ошибку путем уточнения имен и создайте объекты обоих классов.
23. Исправьте ошибку в упражнении 22 при помощи объявления using, которое бы заставляло компилятор выбрать одно из совпадающих имен классов.
24. Извлеките объявления пространства имен из файлов BobsSuperDuperLibrary.cpp и UnnamedNamespaces.cpp, поместите их в разные заголовочные файлы. При-

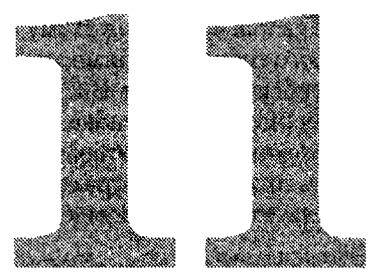
свойте анонимному пространству имя. В третьем заголовочном файле создайте новое пространство имен, которое объединяет элементы первых двух пространств с использованием объявлений using. В main() импортируйте новое пространство имен директивой using и попробуйте обратиться ко всем его элементам.

25. Создайте заголовочный файл с директивами <string> и <iostream>, но без директив или объявлений using. Добавьте препроцессорные директивы, защищающие от многократного включения, как это сделано в заголовочных файлах книги. Создайте класс, все функции которого являются подставляемыми; класс должен содержать переменную string, конструктор, инициализирующий string своим аргументом, и функцию print() для вывода string. Создайте cpp-файл и поэкспериментируйте с классом в main().
26. Создайте класс, содержащий статические переменные double и long. Напишите статическую функцию для вывода их значений.
27. Создайте класс, который содержит переменную типа int, конструктор, инициализирующий int своим аргументом, и функцию print() для вывода int. Затем создайте второй класс, содержащий статический объект первого класса. Добавьте статическую функцию для вызова функции print() статического объекта. Поэкспериментируйте с классом в main().
28. Создайте класс, содержащий два массива int: константный и неконстантный. Напишите статические методы для вывода содержимого массива. Поэкспериментируйте с классом в main().
29. Создайте класс, который содержит переменную типа string, конструктор, инициализирующий int своим аргументом, и функцию print() для вывода string. Создайте другой класс, содержащий два статических массива объектов первого класса (константный и неконстантный) и статические методы для вывода этих массивов. Поэкспериментируйте со вторым классом в main().
30. Создайте структуру, которая содержит int и конструктор по умолчанию, инициализирующий int нулем. Объявите структуру локальной по отношению к функции. Внутри функции создайте массив экземпляров структуры и покажите, что все переменные int в элементах массива автоматически инициализируются нулями.
31. Создайте класс, представляющий соединение с принтером. Класс должен поддерживать только один принтер.
32. В заголовочном файле создайте класс Mirror, содержащий указатель на объект Mirror и переменную bool. Определите два конструктора: конструктор по умолчанию инициализирует bool значением true, а указателю Mirror присваивает ноль. Второй конструктор получает указатель на объект Mirror, присваивает его внутренней переменной, а затем присваивает переменной bool значение false. Включите в класс функцию test(): если указатель на объект отличен от нуля, то функция должна возвращать значение, полученное при вызове test() через указатель. Если указатель равен нулю, функция должна возвращать bool. Создайте пять cpp-файлов, в каждый из которых включите заголовочный файл Mirror. Первый cpp-файл определяет глобальный объект Mirror с использованием конструктора по умолчанию. Второй файл объявляет

объект в первом файле внешним (extern) и определяет глобальный объект Mirror с использованием второго конструктора и передачей указателя на первый объект. Цепочка продолжается до последнего файла, который также содержит определение глобального объекта. В этом файле функция main() вызывает функцию test() и выводит результат. Если результат равен true, выясните, как изменить порядок компоновки в вашей системе программирования, и изменяйте его до тех пор, пока функция не вернет false.

33. Исправьте ошибку в упражнении 32 при помощи первого решения, описанного в этой главе.
34. Исправьте ошибку в упражнении 32 при помощи второго решения, описанного в этой главе.
35. Объявите функцию puts() стандартной библиотеки С, не включая заголовочный файл. Вызовите эту функцию в main().

Ссылки и копирующий конструктор



Ссылки можно рассматривать как константные указатели, автоматически разыменуемые компилятором.

Хотя ссылки также поддерживались в языке Pascal, в C++ они пришли из языка Algol. Ссылки абсолютно необходимы для перегрузки операторов (глава 12), но они также являются удобным средством управления передачей данных в аргументах и возвращаемых значениях функций.

В этой главе мы сначала рассмотрим различия между указателями C и C++, а затем перейдем к ссылкам. Однако основная часть материала будет посвящена теме *копирующих конструкторов*, которая нередко вызывает затруднения у новичков. Это особая разновидность конструкторов, создающих новые объекты на базе существующих объектов того же типа. Работа копирующего конструктора неразрывно связана со ссылками. В частности, копирующий конструктор используется компилятором при вызове функции для передачи и возврата объектов *по значению*.

В завершение главы рассматриваются несколько странные на первый взгляд *указатели на члены классов*.

Указатели в C++

Важнейшее различие между указателями C и C++ связано с тем, что C++ является языком с сильной типизацией. Это особенно наглядно проявляется в отношении указателей void*: язык C не разрешает свободно присваивать указатели одного типа переменным-указателям на другой тип, но он *позволяет* выполнить присваивание через void*:

```
bird* b;
rock* r;
void* v;
v = r;
b = v;
```

Так как благодаря этому «свойству» С можно легко интерпретировать данные любого типа как данные другого типа, оно «пробивает огромную брешь» в системе безопасности типов. В C++ такое присваивание невозможно — компилятор выдает сообщение об ошибке. А если вам действительно нужно интерпретировать данные в контексте другого типа, придется явно сообщить об этом как компилятору, так и читателю программы путем приведения типа (усовершенствованный синтаксис «явного» приведения типов в C++ рассматривается в главе 3).

Ссылки в C++

Ссылка (&) похожа на константный указатель, автоматически разыменуемый компилятором. Обычно ссылки используются в списках аргументов и возвращаемых значениях функций, однако вы также можете определять пользовательские ссылочные переменные. Пример:

```
//: C11:FreeStandingReferences.cpp
#include <iostream>
using namespace std;

// Пользовательская ссылочная переменная:
int y;
int& r = y;
// При создании ссылка должна инициализироваться
// существующим объектом.
// Следовательно, возможна запись вида:
const int& q = 12;  // (1)
// Ссылка связывается с памятью, занимаемой другой переменной:
int x = 0;          // (2)
int& a = x;         // (3)
int main() {
  cout << "x = " << x << ", a = " << a << endl;
  a++;
  cout << "x = " << x << ", a = " << a << endl;
} ///:~
```

В строке (1) компилятор выделяет область памяти, инициализирует ее значением 12 и связывает ссылку с выделенной памятью. Вся суть в том, что ссылка связывается с областью памяти, выделенной под некоторые данные. Обращаясь по ссылке, вы обращаетесь к этой памяти. Следовательно, в строках (2) и (3) при увеличении a в действительности увеличивается x, как показано в функции main(). Как упоминалось ранее, ссылку проще всего представить себе как усовершенствованный указатель. У такого «указателя» есть два основных достоинства: программисту не нужно беспокоиться ни о его инициализации (компилятор следит за тем, чтобы это было сделано), ни о его разыменовании (это тоже делает компилятор).

Использование ссылок должно подчиняться нескольким правилам.

- Ссылки должны инициализироваться в момент создания (указатели могут инициализироваться в любой момент).
- После инициализации ссылку нельзя перевести на другой объект (указатели могут переводиться на другие объекты в любой момент).
- Не существует NULL-ссылок. Предполагается, что ссылка всегда связана с действительной областью памяти.

Ссылки в функциях

Ссылки чаще всего используются в аргументах и возвращаемых значениях функций. Если ссылка передается в аргументе функции, то любые изменения по данной ссылке *внутри* функции приведут к изменению аргумента *вне* функции. Конечно, то же самое можно сделать с передачей указателя, но ссылочный синтаксис гораздо удобнее (при желании ссылки можно рассматривать просто как синтаксическое удобство).

Возвращая ссылку из функции, необходимо предпринимать такие же меры предосторожности, как при возврате указателя. Объект, с которым ассоциируется ссылка, не должен исчезать при возврате из функции, иначе ссылка будет ассоциироваться с недействительной памятью.

Пример:

```
//: C11:Reference.cpp
// Простые примеры использования ссылок в C++

int* f(int* x) {
  (*x)++;
  return x; // Безопасно, x существует вне функции
}

int& g(int& x) {
  x++; // Тот же эффект, что и в f()
  return x; // Безопасно, x существует вне функции
}

int& h() {
  int q;
//! return q;  // Ошибка
  static int x;
  return x; // Безопасно, x существует вне функции
}

int main() {
  int a = 0;
  f(&a); // Некрасиво (но предельно ясно)
  g(a);  // Красиво (но смысл остается скрытым)
} ///:~
```

Вызов f() не обладает удобством и четкостью ссылочного синтаксиса, но зато сразу понятно, что функции передается адрес. При вызове g() тоже передается адрес (через ссылку), однако из вызова функции это понять нельзя.

Ссылки на константные объекты

Передача аргумента по ссылке в примере Reference.cpp работает только в том случае, если аргумент является неконстантным объектом. В противном случае функция g() не примет аргумент, и это правильно, потому что функция *изменяет* внешний аргумент. Если вы знаете, что функция соблюдает константность объекта, объявление аргумента как ссылки на константу позволит использовать функцию в любом контексте. Другими словами, для встроенных типов функция не изменяет аргумент, а для пользовательских типов функция вызывает только константные функции класса и не изменяет открытые переменные класса.

Использование ссылок на константные объекты в аргументах функций особенно важно, потому что при вызове функции может передаваться временный объект. Например, этот объект может создаваться автоматически для возвращаемого значения другой функции или явно пользователем функции. Временные объекты всегда константны, поэтому без ссылки на константный объект аргумент не будет принят компилятором. Рассмотрим простейший пример:

```
//: C11:ConstReferenceArguments.cpp
// Передача ссылок на константы

void f(int&) {}
void g(const int&) {}

int main() {
//!  f(1); // Ошибка
  g(1);
} ///:~
```

Вызов f(1) приводит к ошибке компиляции, потому что компилятор должен сначала создать ссылку. Он выделяет память для int, инициализирует ее значением 1 и связывает адрес выделенной памяти со ссылкой. Ссылка *должна* быть константной, поскольку изменение содержимого памяти бессмысленно — временные данные все равно будут уничтожены и станут недоступными. Работая с временными объектами, следует исходить из того же предположения о невозможности их модификации. Информация о попытке изменения таких данных, полученная от компилятора, весьма ценна, потому что модификация ведет лишь к потере данных.

Ссылки на указатели

Если функция должна изменять содержимое *самого указателя*, а не те данные, на которые он ссылается, в языке С объявление функции выглядит примерно так:

```
void f(int**);
```

С другой стороны, при передаче указателя функции приходится дополнительно получать его адрес:

```
int i = 47;
int* ip = &i;
f(&ip_);
```

Ссылочный синтаксис С++ гораздо «чище». Аргумент функции заменяется ссылкой на указатель, и вам уже не приходится получать адрес этого указателя:

```
//: C11:ReferenceToPointer.cpp
#include <iostream>
using namespace std;

void increment(int*& i) { i++; }

int main() {
  int* i = 0;
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
} ///:~
```

Запуск этой программы доказывает, что увеличивается сам указатель, а не те данные, на которые он ссылается.

Рекомендации по передаче аргументов

При передаче аргументов функциям прежде всего подумайте, нельзя ли передать ссылку на константу. Хотя на первый взгляд кажется, что константность отражается только на эффективности программы (а обычно на стадии проектирования и разработки программ заниматься оптимизацией еще рано), в действительности ставка более высока. Как будет показано далее, для передачи объекта по значению необходим копирующий конструктор, который не всегда доступен.

Даже такой простой совет позволяет получить существенный выигрыш в эффективности: передача аргумента по значению требует вызова конструктора и деструктора, но если вы не собираетесь изменять аргумент, то передача по ссылке на константу сводится к занесению адреса в стек.

В сущности, передача адреса *не является* предпочтительным решением только в одной ситуации: если функция изменяет объект так, что у программиста не остается другого выбора, кроме передачи по значению (вместо модификации внешнего объекта, обычно неожиданной для вызывающей стороны). Этой теме посвящен следующий раздел.

Копирующий конструктор

Теперь, когда вы разбираетесь в основных принципах работы со ссылками в C++, можно переходить к более сложной языковой концепции — ***копирующим конструкторам***. Эта разновидность конструкторов имеет большое значение для управления передачей и возвращением пользовательских типов по значению при вызове функций. Более того, копирующие конструкторы настолько важны, что компилятор автоматически генерирует копирующий конструктор, если этого не делает программист.

Передача и возврат по значению

Чтобы вы лучше поняли, почему так важны копирующие конструкторы, рассмотрим механизм передачи и возврата переменных по значению при вызове функций. Предположим, вы объявляете функцию и вызываете ее в программе:

```
int f(int x, char c);
int g = f(a, b);
```

Откуда компилятор знает, как нужно передавать и возвращать эти переменные? Да знает, и все! Диапазон стандартных типов относительно невелик (char, int, float, double и их разновидности), поэтому вся необходимая информация просто встраивается в компилятор.

Если вы разберетесь, как ваш компилятор генерирует ассемблерный код, и найдете фрагмент, сгенерированный при вызове функции f(), то получите примерно следующее:

```
push  b
push  a
call  f()
add   sp,4
mov g, register a
```

В целях обобщения этот фрагмент основательно «подчищен»; так, оформление выражений b и a зависит от того, являются переменные глобальными (и тогда будут использоваться имена _b и _a) или локальными (в этом случае компилятор индексирует их от указателя стека). То же самое относится к g. Внешний вид вызова f() зависит от схемы украшения имен, используемой вашим компилятором, а вид выражения register a — от обозначений регистров процессора в вашем ассемблере. Тем не менее общая логика кода всегда остается постоянной.

В C и C++ аргументы заносятся в стек справа налево, после чего происходит передача управления. Код вызова отвечает за извлечение аргументов из стека (команда add sp,4). Но обратите внимание: чтобы передать аргументы по значению, компилятор просто заносит их копии в стек — ему известны размеры аргументов, и он знает, что при выполнении команды push будут созданы их точные копии.

Возвращаемое значение f() помещается в регистр. И снова компилятор знает все, что необходимо знать о типе возвращаемого значения, поскольку этот тип является встроенным. Для примитивных типов данных C копирование объекта выполняется простым копированием битов значения.

Передача и возврат больших объектов

А теперь займемся пользовательскими типами. Допустим, вы создали класс и хотите передать объект этого класса по значению; каким образом компилятор узнает, как это делается? У него нет встроенной информации об этом типе, поскольку тип был определен пользователем.

Чтобы узнать ответ на этот вопрос, можно начать с простой структуры, размеры которой явно не подходят для передачи в регистрах:

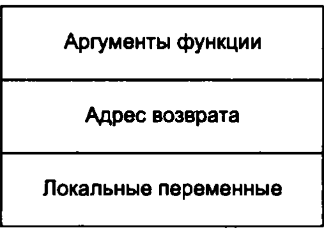
```
//: C11:PassingBigStructures.cpp
struct Big {
  char buf[100];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) {
  b.i = 100; // Делаем что-то с аргументом
  return b;
}

int main() {
  B2 = bigfun(B);
} ///:~
```

На этот раз расшифровать ассемблерный эквивалент оказывается несколько сложнее, поскольку большинство компиляторов использует вспомогательные функции вместо выполнения всех операций «на месте». В функции main() вызов bigfun() начинается именно так, как можно было бы предположить, — все содержимое B заносится в стек (некоторые компиляторы загружают в регистры адрес структуры Big и ее размер, а затем вызывают вспомогательную функцию для занесения Big в стек).

В предыдущем фрагменте программы вся подготовка к вызову ограничивалась занесением аргументов в стек. Но в примере PassingBigStructures.cpp происходит нечто большее: перед вызовом в стек заносится адрес B2, хотя эта переменная явно не является аргументом. Чтобы разобраться в происходящем, необходимо понять, какие ограничения устанавливаются для вызова функций компиляторами.

Кадр стека при вызове функции

Когда компилятор генерирует код вызова функции, он сначала заносит все аргументы в стек, а затем передает управление. Внутри функции указатель стека смещается еще ниже, чтобы зарезервировать память для хранения локальных переменных функции (термин «ниже» в данном случае относителен; на вашем компьютере указатель стека может как увеличиваться, так и уменьшаться). При выполнении ассемблерной команды CALL процессор заносит в стек тот адрес программного кода, *из которого* была вызвана функция. Далее ассемблерная команда RETURN использует этот адрес для возврата к точке вызова. Конечно, этот адрес жизненно важен — без него программа не будет знать, куда вернуть управление. Вот как выглядит кадр стека после выполнения команды CALL и выделения в стеке памяти под локальные переменные функции:



В остальном коде функции предполагается, что стек структурирован именно таким образом, и поэтому компилятор может получить аргументы функции и локальных переменных, не трогая адрес возврата. Мы будем называть полный блок данных, используемых функцией в процессе вызова, *кадром функции*.

На первый взгляд кажется, что возвращаемые значения было бы разумно передавать в стеке. Компилятор просто заносит значение в стек, а функция возвращает смещение значения в стеке.

Реентерабельность

Проблемы начинаются из-за того, что функции C и C++ поддерживают обработку прерываний; иначе говоря, эти языки являются *реентерабельными*. Кроме того, в них поддерживается рекурсивный вызов функций. А это означает, что в любой момент во время выполнения программы может произойти прерывание. Конечно, программист, который пишет обработчик прерывания, должен сохранить и восстановить все регистры процессора, задействованные в обработке. А если обработчику прерывания вдруг придется использовать дополнительную память ниже в стеке, это не должно создавать проблем. (Обработчик прерывания можно рассматривать как обычную функцию без аргументов и с возвращаемым значением void, которая сохраняет и полностью восстанавливает состояние процессора на момент вызова. Обработка прерывания инициируется не программным вызовом, а наступлением некоторого аппаратного события.)

А теперь представьте, что произойдет, если обычная функция попытается возвращать значения в стеке. Трогать данные, находящиеся выше адреса возврата, нельзя, поэтому функции придется занести значения под адресом возврата. Но при выполнении ассемблерной команды RETURN указатель стека должен указывать на адрес возврата (или на позицию сразу же за ним, в зависимости от компьютера),

поэтому перед выполнением RETURN функция должна сместить указатель стека вверх, стирая все свои локальные переменные. Если попытаться занести возвращаемое значение в стек под адресом возврата, возникает опасность того, что в этот момент может произойти прерывание. Чтобы сохранить свой адрес возврата и локальные переменные, обработчик сместит указатель стека вниз и сотрет возвращаемое значение.

Как быть? В принципе *можно* заставить вызывающую сторону перед вызовом функции выделить в стеке дополнительную память для возвращаемых данных. Тем не менее в языке С такое поведение не было предусмотрено, а язык С++ должен быть совместим с С. Как вы вскоре убедитесь, компилятор С++ использует более эффективную схему.

Следующая мысль — вернуть значение в какой-нибудь глобальной области данных, но и этот вариант не работает. Реентерабельность означает, что любая функция может получить управление из любой другой функции, *включая текущую*. Следовательно, если мы поместим возвращаемое значение в глобальную память, управление может быть возвращено этой же функции, которая перезапишет возвращаемое значение. Аналогичная логика применима к рекурсии.

Итак, единственное безопасное место для передачи возвращаемого значения — регистры процессора, и мы возвращаемся к знакомой проблеме: что делать, если возвращаемое значение не помещается в регистрах? Ответ: нужно сохранить адрес возвращаемого значения в стеке как один из аргументов функции и позволить функции скопировать возвращаемые данные прямо в нужную область памяти. Этот вариант не только решает все проблемы, но и оказывается наиболее эффективным. Кстати, именно по этой причине в примере PassingBigStructures.cpp компилятор занес в стек адрес B2 перед вызовом bigfun() в main(). Просмотрев ассемблерный код bigfun(), вы убедитесь, что функция знает о существовании скрытого аргумента и копирует в него свой результат.

Поразрядное копирование и инициализация

Пока все идет хорошо. У нас имеется работоспособный механизм передачи и возврата больших простых структур. Но если задуматься, этот механизм способен лишь копировать двоичное представление данных из одного места в другое. Для упрощенного подхода к переменным в языке С он работает нормально, но объекты С++ бывают гораздо сложнее простого набора битов. Они обладают смысловой интерпретацией, которая может плохо реагировать на простое копирование.

Рассмотрим простой пример — класс, который знает, сколько объектов «своего» типа существует в произвольный момент времени. Из главы 10 вы уже знаете, как решить эту задачу с использованием статической переменной:

```
//: C11:HowMany.cpp
// Класс с подсчетом экземпляров
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany.out");

class HowMany {
  static int objectCount;
public:
  HowMany() { objectCount++; }
```

```
  static void print(const string& msg = "") {
    if(msg.size() != 0) out << msg << ": ";
    out << "objectCount = "
         << objectCount << endl;
  }
  ~HowMany() {
    objectCount--;
    print("~HowMany()");
  }
};

int HowMany::objectCount = 0;

// Передача и возврат ПО ЗНАЧЕНИЮ:
HowMany f(HowMany x) {
  x.print("x argument inside f()");
  return x;
}

int main() {
  HowMany h;
  HowMany::print("after construction of h");
  HowMany h2 = f(h);
  HowMany::print("after call to f()");
} ///:~
```

Класс HowMany содержит счетчик static int objectCount и статическую функцию print() для вывода значения objectCount с необязательным строковым аргументом. Конструктор увеличивает счетчик при каждом создании объекта, а деструктор уменьшает его.

Однако результат выполнения программы отличается от ожидаемого:

```
after construction of h: objectCount = 1
x argument inside f(): objectCount = 1
~HowMany(): objectCount = 0
after call to f(): objectCount = 0
~HowMany(): objectCount = -1
~HowMany(): objectCount = -2
```

После создания h счетчик объектов равен 1, и это нормально. Но после вызова f() он вроде был должен стать равным 2 из-за появления объекта h2. Однако счетчик равен нулю, значит, в программе произошла какая-то серьезная ошибка. Это подтверждается тем фактом, что вызовы деструкторов в конце программы уменьшают счетчик до отрицательной величины, чего быть вообще не должно.

Стоит повнимательнее присмотреться к функции f() после передачи аргумента по значению. Передача по значению говорит о том, что исходный объект h существует где-то за пределами кадра функции, а *внутри* кадра функции появился дополнительный объект — копия, переданная по значению. Но при передаче аргумента использовалась примитивная концепция поразрядного копирования C, а для поддержания целостности класса C++ HowMany необходима инициализация. То есть применяемое по умолчанию поразрядное копирование не обеспечивает желаемого эффекта.

Когда локальный объект прекращает свое существование в конце вызова f(), для него вызывается деструктор, уменьшающий счетчик objectCount, поэтому за пределами функции счетчик objectCount равен нулю. Экземпляр h2 тоже создается

поразрядным копированием, при этом конструктор не вызывается, а при выходе h и h2 из области видимости их деструкторы уменьшают счетчик objectCount до отрицательной величины.

Конструирование копий

Проблема возникает из-за того, что компилятор делает некие предположения относительно *создания нового объекта на базе существующего объекта*. При передаче объекта по значению вы создаете новый объект (переданный объект в кадре функции) на базе существующего объекта (исходный объект, находящийся вне кадра функции). То же самое часто происходит и при возвращении объекта из функции. В следующем выражении несконструированный объект h2 создается по возвращаемому значению f(), то есть новый объект снова создается на базе существующего объекта:

```
HowMany h2 = f(h);
```

По умолчанию компилятор предполагает, что объект должен создаваться поразрядным копированием. Такое предположение подходит для многих случаев, но не для HowMany, поскольку смысл инициализации выходит за пределы простого копирования. Другая типичная проблема возникает при наличии указателей в классе. На что должны ссылаться соответствующие переменные копии? Нужно ли копировать их значения или выделить для них новый блок памяти?

К счастью, вы можете вмешаться в этот процесс и предотвратить поразрядное копирование. Для этого необходимо определить собственную функцию, которая будет вызываться каждый раз, когда компилятору понадобится создать новый объект на базе существующего объекта. Речь идет о создании новых объектов, поэтому вполне логично, что новая функция является конструктором. Также логично, что единственный аргумент этого конструктора как-то связан с прототипом (объектом, на базе которого конструируется новый объект). Но объект нельзя передавать конструктору по значению, потому что вы *определяете* функцию, которая должна обеспечивать передачу по значению. С другой стороны, передавать указатель тоже бессмысленно, поскольку новый объект все же создается на базе существующего объекта. Здесь на помощь приходят ссылки — вы просто получаете ссылку на исходный объект. Полученная функция и называется *копирующим конструктором*.

При наличии копирующего конструктора компилятор при создании новых объектов на базе существующих не будет выполнять поразрядное копирование — он всегда будет вызывать копирующий конструктор. При отсутствии копирующего конструктора компилятор все равно постарается сделать что-нибудь разумное, но у программиста есть возможность полностью взять на себя контроль над процессом.

Теперь мы можем исправить ошибку в HowMany.cpp:

```
//: C11:HowMany2.cpp
// Копирующий конструктор
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
```

```
    string name: // Идентификатор объекта
    static int objectCount:
  public:
    HowMany2(const string& id = "") : name(id) {
      ++objectCount:
      print("HowMany2()"):
    }
    ~HowMany2() {
      --objectCount:
      print("~HowMany2()"):
    }
    // Копирующий конструктор:
    HowMany2(const HowMany2& h) : name(h.name) {
      name += " copy":
      ++objectCount:
      print("HowMany2(const HowMany2&)"):
    }
    void print(const string& msg = "") const {
      if(msg.size() != 0)
        out << msg << endl:
      out << '\t' << name << ": "
          << "objectCount = "
          << objectCount << endl:
    }
  }:

  int HowMany2::objectCount = 0:

  // Передача и возврат ПО ЗНАЧЕНИЮ:
  HowMany2 f(HowMany2 x) {
    x.print("x argument inside f()"):
    out << "Returning from f()" << endl:
    return x:
  }

  int main() {
    HowMany2 h("h"):
    out << "Entering f()" << endl:
    HowMany2 h2 = f(h):
    h2.print("h2 after call to f()"):
    out << "Call f(). no return value" << endl:
    f(h):
    out << "After call to f()" << endl:
  } ///:~
```

В программу добавлен ряд новых элементов, которые помогают разобраться в происходящем. Во-первых, переменная string name используется для идентификации объекта при выводе информации. В конструкторе этот идентификатор (обычно имя объекта) копируется в name при помощи конструктора string. Значение по умолчанию = "" создает пустой объект string. Как и прежде, конструктор увеличивает счетчик objectCount, а деструктор уменьшает его.

Далее идет копирующий конструктор HowMany2(const HowMany2&). Он создает новый объект только на базе существующего объекта, поэтому имя существующего объекта копируется в name с прибавлением суффикса «copy», чтобы было сразу понятно, откуда взялся новый объект. Если присмотреться повнимательнее, становится ясно, что функция name(h.name) в списке инициализирующих значений конструктора в действительности вызывает копирующий конструктор string.

Копирующий конструктор увеличивает счетчик объектов точно так же, как это делается в обычном конструкторе. А это значит, что теперь при передаче и возврате по значению счетчик будет правильно отражать количество существующих экземпляров.

Функция print() тоже была изменена: теперь она выводит сообщение, идентификатор объекта и счетчик экземпляров. Поскольку эта функция должна обращаться к переменной name конкретного объекта, она уже не может быть статической функцией.

Внутри функции main() появился второй вызов f(). В этом вызове используется стандартный прием С: возвращаемое значение игнорируется. Но теперь, когда вы знаете, как организуется возврат значения (код *внутри* функции заносит результат в область памяти, адрес которой передается в скрытом аргументе), может возникнуть вопрос: что происходит, когда возвращаемое значение игнорируется? Выходные данные программы помогут найти ответ.

Но прежде чем приводить результат, рассмотрим небольшую программу, которая нумерует строки произвольного файла с использованием потоков ввода-вывода:

```
//: C11:Linenum.cpp
//{T} Linenum.cpp
// Нумерация строк
#include "../require.h"
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1, "Usage: linenum file\n"
    "Adds line numbers to file");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  string line;
  vector<string> lines;
  while(getline(in, line)) // Чтение всего файла
    lines.push_back(line);
  if(lines.size() == 0) return 0;
  int num = 0;
  // Ширина определяется по количеству строк в файле:
  const int width =
    int(log10((double)lines.size())) + 1;
  for(int i = 0; i < lines.size(); i++) {
    cout.setf(ios::right, ios::adjustfield);
    cout.width(width);
    cout << ++num << ") " << lines[i] << endl;
  }
} ///:~
```

Весь файл загружается в контейнер vector<string> (код, который решает эту задачу, уже приводился в книге). Чтобы сохранить выравнивание при нумерации, необходимо внести поправку на количество строк в файле; номера должны иметь одинаковую ширину независимо от количества цифр. Вообще говоря, количество строк легко определяется функцией vector::size(), но нас интересует лишь превышение определенных пороговых значений: больше 10 строк, больше 100 строк, боль-

ше 1000 строк и т. д. Если вычислить десятичный логарифм от количества строк в файле, взять его целую часть и прибавить 11, мы получим максимальную ширину номера строки (в символах).

В цикле for бросается в глаза пара странных вызовов: setf() и width(). Это управляющие функции класса ostream, которые определяют тип выравнивания и ширину выводимых данных. А поскольку эти функции должны вызываться при каждом выводе строки, они включены в цикл for. Во втором томе книги целая глава посвящена потокам ввода-вывода; в ней будут рассмотрены и эти функции, и другие возможности управления потоками ввода-вывода.

Если запустить пример Linenum.cpp для файла HowMany2.out, будет получен следующий результат:

```
 1) HowMany2()
 2)   h: objetCount = 1
 3) Entering f()
 4) HowMany2(const HowMany2&)
 5)   h copy: objectCount = 2
 6) x argument inside f()
 7)   h copy: objectCount = 2
 8) Returning from f()
 9) HowMany2(const HowMany2&)
10)   h copy: objectCount = 3
11) ~HowMany2()
12)   h copy: objectCount = 2
13) h2 after call to f()
14)   h copy copy: objectCount = 2
15) Call f(), no return value
16) HowMany2(const HowMany2&)
17)   h copy: objectCount = 3
18) x argument inside f()
19)   h copy: objectCount = 3
20) Returning from f()
21) HowMany2(const HowMany2&)
22)   h copy copy: objectCount = 4
23) ~HowMany2()
24)   h copy: objectCount = 3
25) ~HowMany2()
26)   h copy copy: objectCount = 2
27) After call to f()
28) ~HowMany2()
29)   h copy copy: objectCount = 1
30) ~HowMany2()
31)   h: objectCount = 0
```

Как и следовало ожидать, работа программы начинается с вызова обычного конструктора для h, что приводит к увеличению счетчика ссылок на 1. Но затем при входе в f() компилятор автоматически вызывает копирующий конструктор для передачи объекта по значению. В кадре функции f() создается новый объект, который является копией h (отсюда и имя «h copy»). В результате выполнения копирующего конструктора счетчик объектов увеличивается до 2.

Строка 8 обозначает начало возврата из f(). Но перед уничтожением локальной переменной «h copy», выходящей из области видимости в конце функции, она должна быть скопирована в переменную возвращаемого значения h2. Несконструированный объект h2 создается на базе существующего объекта (локальной переменной внутри f()); конечно, это приводит к очередному вызову копирующего

конструктора в строке 9. Теперь идентификатор h2 принимает вид «h copy copy», потому что данные копируются из копии (локального объекта внутри f()). После возвращения объекта, но перед выходом из функции счетчик временно увеличивается до 3, после чего локальный объект «h copy» уничтожается. После завершения вызова f() в строке 13 в программе остаются всего два объекта h и h2, и мы видим, что h2 действительно был присвоен объект с идентификатором «h copy copy».

Временные объекты

Строка 15 начинается с вызова f(h). На этот раз возвращаемое значение функции игнорируется. Из строки 16 видно, что передача аргумента, как и прежде, сопровождается вызовом копирующего конструктора. Кроме того, как и прежде, строка 21 показывает, что для возвращаемого значения тоже вызывается копирующий конструктор. Но для работы копирующего конструктора нужен адрес, который будет использоваться в качестве приемника (указатель this). Откуда он берется?

Оказывается, компилятор создает временные объекты всюду, где они нужны для правильного вычисления выражения. В данном случае объект создается для приема игнорируемого возвращаемого значения f(). Срок жизни этого временного объекта сокращается до абсолютного минимума, чтобы программа не загромождалась временными объектами, которые ожидают уничтожения и лишь зря расходуют ценные ресурсы. Иногда временный объект немедленно передается другой функции, но в нашем случае он не используется после вызова функции. Поэтому сразу же после вызова деструктора локального объекта (строки 23 и 24), завершающего вызов функции, временный объект уничтожается (строки 25 и 26).

Наконец, в строках 28–31 объекты h2 и h уничтожаются, и счетчик экземпляров, как и положено, становится равным нулю.

Копирующий конструктор по умолчанию

Копирующий конструктор обеспечивает передачу и прием данных по значению, поэтому для простых структур компилятор генерирует его за программиста — в сущности, происходит то же, что в языке C. Но до настоящего момента встречался только самый примитивный вариант — поразрядное копирование передаваемых данных.

Для более сложных типов компилятор C++ тоже автоматически генерирует копирующий конструктор, если он не был предоставлен программистом. Но как отмечалось ранее, поразрядное копирование далеко не всегда обеспечивает правильную интерпретацию копирования.

Рассмотрим более разумный подход, предлагаемый компилятором. Допустим, вы создаете новый класс из объектов нескольких существующих классов (этот способ называется *композицией*). Теперь представьте себя на месте наивного пользователя, пытающегося поскорее создать новый класс для решения своих задач. Он ничего не знает о копирующих конструкторах и поэтому не определяет такой конструктор в своем классе. Следующий пример показывает, как компилятор подходит к созданию копирующего конструктора для нового класса:

```
//: C11:DefaultCopyConstructor.cpp
// Автоматическое создание копирующего конструктора
#include <iostream>
```

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

class WithCC { // С копирующим конструктором
public:
  // Обязательно должен присутствовать конструктор по умолчанию:
  WithCC() {}
  WithCC(const WithCC&) {
    cout << "WithCC(WithCC&)" << endl;
  }
};

class WoCC { // Без копирующего конструктора
  string id;
public:
  WoCC(const string& ident = "") : id(ident) {}
  void print(const string& msg = "") const {
    if(msg.size() != 0) cout << msg << ": ";
    cout << id << endl;
  }
};

class Composite {
  WithCC withcc; // Внутренние объекты
  WoCC wocc;
public:
  Composite() : wocc("Composite()") {}
  void print(const string& msg = "") const {
    wocc.print(msg);
  }
};

int main() {
  Composite c;
  c.print("Contents of c");
  cout << "Calling Composite copy-constructor"
       << endl;
  Composite c2 = c;  // Вызов копирующего конструктора
  c2.print("Contents of c2");
} ///:~
```

В класс WithCC включен копирующий конструктор, который просто выводит сообщение о вызове. Возникает интересная ситуация. В классе Composite объект WithCC создается конструктором по умолчанию. Если бы класс WithCC не содержал ни одного конструктора, то компилятор автоматически сгенерировал бы конструктор по умолчанию, который в данном случае ничего бы не делал. Но, определяя копирующий конструктор, вы сообщаете компилятору, что собираетесь создавать конструкторы самостоятельно. Поэтому компилятор не генерирует конструктор по умолчанию и сообщает об ошибке, если этот конструктор не был явно задан в классе (в WithCC такой конструктор присутствует).

Класс WoCC не имеет копирующего конструктора, но у него есть обычный конструктор, сохраняющий сообщение во внутренней переменной string (для вывода сообщения используется функция print()). Этот конструктор вызывается в списке инициализирующих значений конструктора Composite (списки инициализирующих значений кратко представлены в главе 8, а их полное описание приводится в главе 14). Вскоре вы поймете, зачем это делается.

Класс Composite содержит переменные типов WithCC и WoCC (при этом внутренний объект wocc инициализируется в списке инициализирующих значений конструктора, как оно и должно быть), но не имеет явно определенного копирующего конструктора. Однако в функции main() следующее определение приводит к созданию объекта копирующим конструктором:

```
Composite c2 = c:
```

Копирующий конструктор Composite автоматически генерируется компилятором, а выходные данные программы дают представление о том, как это происходит:

```
Contents of c: Composite()
Calling Composite copy-constructor
WithCC(WithCC&)
Contents of c2: Composite()
```

Чтобы создать копирующий конструктор для класса, использующего композицию (а также наследование — глава 14), компилятор рекурсивно вызывает копирующие конструкторы всех объектов класса и базовых классов. Следовательно, если внутренний объект содержит другой вложенный объект, компилятор также вызовет копирующий конструктор этого объекта. Итак, в данном случае компилятор вызывает копирующий конструктор WithCC, о чем свидетельствуют выходные данные программы. Поскольку класс WoCC не имеет своего копирующего конструктора, компилятор генерирует для него конструктор с поразрядным копированием и вызывает его в копирующем конструкторе Composite. Это доказывается вызовом Composite::print() в main() — содержимое c2.wocc идентично содержимому c.wocc.

Всегда определяйте копирующий конструктор самостоятельно, не доверяя компилятору. Только в этом случае можно быть полностью уверенным в его поведении.

Альтернативы

Обычно на этой стадии у читателей начинает кружиться голова. Как же они раньше умудрялись создавать работоспособные классы, не зная о копирующих конструкторах? Но помните: копирующий конструктор необходим только в том случае, если объект вашего класса должен передаваться *по значению*. Если это не нужно, то и копирующий конструктор не нужен.

Предотвращение передачи по значению

«Но если я не создам копирующий конструктор, то компилятор сделает это за меня, — скажете вы. — Тогда откуда я узнаю, что объект где-то передается по значению?»

Для предотвращения передачи по значению существует простой прием: объявите закрытый (private) копирующий конструктор. Вам даже не придется создавать определение, если только какой-нибудь функции класса или дружественной функции не потребуется передавать данные по значению. Но если пользователь попытается передать или вернуть объект по значению, то компилятор выдаст сообщение об ошибке, поскольку копирующий конструктор объявлен закрытым. Компилятор уже не сможет сгенерировать копирующий конструктор по умолчанию, поскольку вы заявили, что будете заниматься этим сами.

Пример:

```
//: C11:NoCopyConstruction.cpp
// Запрет на конструирование копий

class NoCC {
  int i;
  NoCC(const NoCC&); // Определение отсутствует
public:
  NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);

int main() {
  NoCC n;
//! f(n); // Ошибка: вызов копирующего конструктора
//! NoCC n2 = n; // Ошибка: вызов копирующего конструктора
//! NoCC n3(n); // Ошибка: вызов копирующего конструктора
} ///:~
```

Обратите внимание на обобщенную форму, в которой используется ключевое слово const:

```
NoCC(const NoCC&);
```

Изменение внешних объектов в функциях

Ссылочный синтаксис удобнее синтаксиса с указателями, однако он скрывает смысл происходящего. Например, в библиотеке потоков ввода-вывода одна из перегруженных версий функции get() получает аргумент char& и заменяет его прочитанным символом. Однако при чтении программы факт изменения внешнего объекта совершенно не очевиден:

```
char c;
cin.get(c);
```

Вызов функции выглядит так же, как обычная передача по значению, и создает впечатление, что внешний объект *не* меняется.

Если вы хотите упростить сопровождение программы, то при передаче адреса изменяемого аргумента лучше использовать указатели. Если адреса *всегда* передаются в виде ссылок на константы, а в случае модификации внешнего объекта передается указатель на неконстантный объект, читателю будет проще разобраться в логике программы.

Указатели на члены классов

Указатель представляет собой переменную, содержащую адрес некоторой области памяти. Значение указателя может изменяться во время выполнения программы. Указатель может указывать как на данные, так и на функции. *Указатели на члены классов* C++ следуют тому же принципу, но они ссылаются на область памяти внутри класса. Возникает проблема: указатель ассоциируется с адресом, однако внутри класса никаких «адресов» нет — члены класса определяются смещением. Реальный адрес может быть получен лишь в результате применения этого смещения к начальному адресу конкретного объекта. Синтаксис

указателей на члены классов требует, чтобы разыменование указателя происходило с выбором объекта.

Чтобы лучше понять этот синтаксис, рассмотрим простую структуру so и ссылающийся на нее указатель sp. В следующем фрагменте показаны варианты синтаксиса обращения к члену структуры:

```
//: C11:SimpleStructure.cpp
struct Simple { int a; };
int main() {
  Simple so, *sp = &so;
  sp->a;
  so.a;
} ///:~
```

Допустим, в программе имеется обычный указатель на int с именем ip. Чтобы обратиться к данным, на которые ссылается ip, нужно разыменовать указатель:

```
*ip = 4;
```

Остается лишь понять, как должен выглядеть указатель на один из членов объекта класса, который в действительности представляет смещение от начала объекта. Чтобы обратиться к члену объекта, нужно разыменовать указатель оператором разыменования (*). Но так как смещение задается по отношению к объекту, необходимо также задать этот объект. Следовательно, оператор разыменования должен объединиться с разыменованием объекта. Итак, для указателя на объект мы получаем синтаксис ->*, а для объекта или ссылки — .*, как показано ниже:

```
objectPointer->*pointerToMember = 47;
object.*pointerToMember = 47;
```

А как должно выглядеть определение pointerToMember? Как и для любого указателя, мы должны задать тип, на который он ссылается, и включить в определение оператор разыменования (*). Единственное различие состоит в том, что в определении также необходимо указать, с объектами какого класса будет использоваться этот указатель. Естественно, задача решается при помощи имени класса и оператора уточнения области видимости (::). Таким образом, следующая строка определяет переменную с именем pointerToMember, которая ссылается на переменную типа int в классе ObjectClass:

```
int ObjectClass::*pointerToMember;
```

Указатель на член класса может инициализироваться при определении (или позднее):

```
int ObjectClass::*pointerToMember = &ObjectClass::a;
```

Указатель не инициализируется «адресом» ObjectClass::a, поскольку ObjectClass — имя класса, а не конкретного объекта этого класса. Следовательно, конструкции вида &ObjectClass::a могут использоваться только для указателей на члены классов.

Следующий пример показывает, как создать и использовать указатель на переменные класса:

```
//: C11:PointerToMemberData.cpp
#include <iostream>
using namespace std;

class Data {
```

```
public:
  int a, b, c;
  void print() const {
    cout << "a = " << a << ", b = " << b
         << ", c = " << c << endl;
  }
};

int main() {
  Data d, *dp = &d;
  int Data::*pmInt = &Data::a;
  dp->*pmInt = 47;
  pmInt = &Data::b;
  d.*pmInt = 48;
  pmInt = &Data::c;
  dp->*pmInt = 49;
  dp->print();
} ///:~
```

Такой синтаксис крайне неудобен, поэтому прибегать к нему следует только в особых случаях (для которых, собственно, и предназначены указатели на члены классов).

Возможности указателей на члены классов сильно ограничены: им могут присваиваться только значения, связанные с конкретными позициями внутри класса. В частности, в отличие от обычных указателей, к ним не могут применяться операции инкремента или сравнения.

Функции классов

Аналогичные рассуждения приводят к синтаксису указателей на функции классов. Указатели на функцию (см. главу 3) определяются так:

```
int (*fp) (float);
```

Круглые скобки в (*fp) необходимы для того, чтобы компилятор правильно интерпретировал определение. Без них это определение воспринимается компилятором как функция, возвращающая int*.

Круглые скобки также играют важную роль при определении и использовании указателей на функции классов. Если класс содержит функцию, то для задания указателя на нее в стандартное определение указателя на функцию включается имя класса с оператором уточнения области видимости (::):

```
//: C11:PmemFunDefinition.cpp
class Simple2 {
public:
  int f(float) const { return 1; }
};
int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;
int main() {
  fp = &Simple2::f;
} ///:~
```

Как видно из определения fp2, указатель на функцию класса может инициализироваться при создании или позже. Но в отличие от обычных функций при

получении адреса функции класса оператор & *обязателен*. С другой стороны, идентификатор функции может указываться без списка аргументов, поскольку нужная перегруженная версия может быть выбрана по типу указателя на функцию класса.

Пример

Указатели хороши тем, что их можно переводить на другие объекты во время выполнения программы. Тем самым достигается гибкость, столь необходимая при программировании, поскольку через указатель можно динамически выбирать или изменять *поведение* программы на стадии выполнения. То же относится и к указателям на члены класса — они тоже позволяют выбирать члены класса во время выполнения программы. Обычно открытый интерфейс класса состоит только из функций (как правило, переменные класса считаются частью базовой реализации), поэтому в следующем примере указатель используется для выбора одной из функций класса во время выполнения:

```
//: C11:PointerToMemberFunction.cpp
#include <iostream>
using namespace std;

class Widget {
public:
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
};

int main() {
  Widget w;
  Widget* wp = &w;
  void (Widget::*pmem)(int) const = &Widget::h;
  (w.*pmem)(1);
  (wp->*pmem)(2);
} ///:~
```

Конечно, не стоит заставлять рядовых пользователей работать со столь сложными выражениями. Если пользователь должен напрямую работать с указателем на функцию класса, уместно воспользоваться оператором typedef. А чтобы программа стала еще удобнее, указатель на функцию можно включить во внутреннюю реализацию класса. Ниже приведен предыдущий пример, в котором указатель на функцию перемещен *внутрь* класса. Для выбора функции пользователю достаточно ввести ее номер[1]:

```
//: C11:PointerToMemberFunction2.cpp
#include <iostream>
using namespace std;

class Widget {
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
```

[1] Спасибо Оуэну Мортенсену (Owen Mortensen) за этот пример.

```
    void h(int) const { cout << "Widget::h()\n"; }
    void i(int) const { cout << "Widget::i()\n"; }
    enum { cnt = 4 };
    void (Widget::*fptr[cnt])(int) const;
  public:
    Widget() {
      fptr[0] = &Widget::f; // Необходима полная спецификация
      fptr[1] = &Widget::g;
      fptr[2] = &Widget::h;
      fptr[3] = &Widget::i;
    }
    void select(int i, int j) {
      if(i < 0 || i >= cnt) return;
      (this->*fptr[i])(j);
    }
    int count() { return cnt; }
  };

  int main() {
    Widget w;
    for(int i = 0; i < w.count(); i++)
      w.select(i, 47);
  } ///:~
```

В интерфейсе класса и в main() вся реализация, включая функции, полностью скрыта. Даже количество функций приходится определять вызовом count(). Таким образом, автор реализации может изменить количество функций в базовой реализации, и это никак не отразится на работе программ, использующих класс.

Инициализация указателей на функции класса в конструкторе может показаться излишне подробной. Рассмотрим такую запись:

```
fptr[1] = &g;
```

Почему ее нельзя использовать, если имя g принадлежит функции, автоматически входящей в область видимости класса? Дело в том, что это не соответствует синтаксису указателей на члены классов, обязательному для того, чтобы все — и в первую очередь компилятор — могли понять, что происходит в программе. Аналогично, при разыменовании указателя на функцию класса может показаться, что следующая запись слишком подробна (ключевое слово this кажется лишним):

```
(this->*fptr[i])(j);
```

И снова синтаксис требует, чтобы указатель на член класса при разыменовании всегда связывался с конкретным объектом.

Итоги

Указатели C++ почти не отличаются от указателей C, и это хорошо. В противном случае многие существующие программы C не компилировались бы в C++. А так ошибки времени компиляции возникают только при выполнении опасных присваиваний. Если такие присваивания действительно необходимы, ошибки компиляции предотвращаются простыми (и наглядными) операциями приведения типов.

Из языков Algol и Pascal в C++ были позаимствованы *ссылки* — константные указатели, автоматически разыменуемые компилятором. Ссылка содержит адрес, но интерпретируется как объект. Ссылки упрощают синтаксис перегрузки операторов (этой теме посвящена следующая глава), но они также являются удобным синтаксическим средством для передачи и возвращения объектов обычными функциями.

Копирующий конструктор получает ссылку на существующий объект и использует ее для создания нового объекта на базе существующего объекта. Компилятор автоматически вызывает копирующий конструктор при передаче или возврате объекта по значению. Компилятор может автоматически создать копирующий конструктор за программиста, но если вы считаете, что такой конструктор действительно необходим для вашего класса, всегда определяйте его самостоятельно. Только так вы будете полностью уверены в том, что копирование выполняется правильно. Чтобы запретить передачу или возврат объектов по значению, создайте закрытый копирующий конструктор.

Указатели на члены классов предоставляют примерно те же возможности, что и обычные указатели: с их помощью во время выполнения программы динамически выбирается область памяти, содержащая данные или функцию. Просто, в отличие от глобальных данных или функций, они работают с членами классов. Тем самым обеспечивается динамическая модификация поведения программы во время ее выполнения.

Упражнения

1. Преобразуйте фрагмент с bird и rock, приведенный в начале главы, в программу C (используйте структурные переменные) и убедитесь в том, что программа успешно компилируется. Попробуйте откомпилировать ее компилятором C++ и посмотрите, что произойдет.
2. Возьмите фрагменты кода из раздела «Ссылки в C++» и поместите их в функцию main(). Добавьте команды вывода и убедитесь в том, что ссылки ведут себя как указатели с автоматическим разыменованием.
3. Напишите программу, в которой требуется попытаться, во-первых, создать ссылку без инициализации, во-вторых, перевести ссылку на другой объект после инициализации, в-третьих, создать NULL-ссылку.
4. Напишите функцию, которая получает аргумент-указатель, изменяет данные, на которые он ссылается, и затем возвращает эти данные в виде ссылки.
5. Создайте класс с несколькими функциями. Передайте указатель на этот класс в аргументе упражнения 4. Объявите указатель с ключевым словом const, сделайте некоторые функции класса константными и докажите, что внутри функции можно вызывать только константные функции класса. Преобразуйте аргумент функции из указателя в ссылку.
6. Возьмите фрагменты кода из пункта «Ссылки на указатели» в подразделе «Ссылки в функциях» раздела «Ссылки в C++» и преобразуйте их в программу.

7. Создайте функцию, которая в аргументе получает ссылку на указатель, в свою очередь указывающий на указатель (то есть ссылку на указатель на указатель), и модифицирует этот аргумент. Вызовите эту функцию в main().
8. Создайте функцию, которая получает аргумент char& и изменяет его. В функции main() выведите переменную char, вызовите функцию для этой переменной и снова выведите ее. Убедитесь в том, что переменная изменилась. Как это повлияло на удобочитаемость программы?
9. Создайте класс, содержащий константную и неконстантную функции. Напишите три функции, которые получают в аргументе объект этого класса. Первая функция получает объект по значению, вторая — по ссылке, а третья — по ссылке на константу. Попробуйте вызвать обе функции класса внутри этих функций и объясните результаты.
10. (Задача несколько повышенной трудности) Напишите простую функцию, которая получает аргумент int, инкрементирует и возвращает его. Вызовите свою функцию из main(). Выясните, как ваш компилятор генерирует ассемблерный код, и проанализируйте команды ассемблера. Убедитесь в том, что вы понимаете, как происходит передача и возврат аргументов и как локальные переменные индексируются от указателя стека.
11. Напишите функцию с аргументами типов char, int, float и double. Сгенерируйте ассемблерный код и найдите команды занесения аргументов в стек перед вызовом функции.
12. Напишите функцию, которая возвращает double. Сгенерируйте ассемблерный код и определите, как происходит возврат значения.
13. Сгенерируйте ассемблерный код для примера PassingBigStructures.cpp. Проанализируйте его и разберитесь, как компилятор передает и возвращает большие структуры.
14. Напишите простую рекурсивную функцию, которая декрементирует свой аргумент. Если аргумент становится равным нулю, то функция возвращает ноль; в противном случае она вызывает себя с новым значением аргумента. Сгенерируйте ассемблерный код для этой функции и объясните, как в нем поддерживается рекурсия.
15. Напишите программу, которая бы доказывала, что компилятор автоматически генерирует копирующий конструктор, если он не был определен программистом. Продемонстрируйте, что сгенерированный копирующий конструктор выполняет поразрядное копирование для примитивных типов и вызывает копирующий конструктор для пользовательских типов.
16. Напишите класс с копирующим конструктором, который выводит сообщение в cout. Затем создайте функцию, которая передает объект нового класса по значению, и другую функцию, создающую локальный объект нового класса и возвращающую его по значению. Вызовите эти функции и убедитесь в том, что копирующий конструктор действительно автоматически вызывается при передаче и возврате объектов по значению.

17. Создайте класс, содержащий переменную double*. Конструктор инициализирует double*, для чего он вызывает new double и присваивает значения указателя на данные, переданные в аргументе конструктора. Деструктор выводит значение, на которое ссылается указатель, заменяет его значением –1, вызывает delete для выделенной памяти и затем обнуляет указатель. Создайте функцию, которая получает по значению объект класса, и вызовите эту функцию в main(). Что происходит? Напишите копирующий конструктор и исправьте ошибку.
18. Создайте класс с конструктором, который выглядит как копирующий конструктор, но имеет дополнительный аргумент со значением по умолчанию. Покажите, что этот конструктор все равно используется как копирующий.
19. Создайте класс с копирующим конструктором, выводящим сообщение о вызове. Создайте второй класс, содержащий внутренний объект первого класса, но не определяйте в нем копирующий конструктор. Покажите, что копирующий конструктор, сгенерированный для второго класса, автоматически вызывает копирующий конструктор первого класса.
20. Создайте очень простой класс и функцию, которая возвращает объект этого класса по значению. Создайте вторую функцию, которая получает ссылку на объект класса. Передайте результат вызова первой функции в качестве аргумента второй функции и покажите, что аргумент второй функции должен определяться как ссылка на константу.
21. Создайте простой класс без копирующего конструктора и простую функцию, которой объект этого класса передается по значению. Измените класс и включите в него закрытое объявление (и только объявление!) копирующего конструктора. Объясните, что происходит при компиляции функции.
22. В данном упражнении описана методика, которая может помочь отказаться от копирующих конструкторов. Создайте класс X и объявите (но не определяйте!) закрытый копирующий конструктор. Объявите открытую функцию clone() как константную функцию класса, которая возвращает копию объекта, созданную оператором new. Напишите функцию, получающую аргумент const X& и вызывающую clone() для создания локальной копии, которая может модифицироваться. Недостаток такого решения состоит в том, что вы отвечаете за уничтожение копии (вызовом delete) после завершения работы с ней.
23. Объясните, где допущены ошибки в примерах Mem.cpp и MemTest.cpp из главы 7. Исправьте их.
24. Создайте класс, содержащий переменную double и функцию print() для вывода double. В функции main() создайте указатели на переменную и на функцию класса. Создайте объект класса и указатель на этот объект, выполните операции с переменной и функцией класса через указатели (используя как объект, так и указатель на него).
25. Создайте класс, содержащий массив int. Можно ли индексировать элементы массива, используя указатель на него?

26. Измените пример PmemFunDefinition.cpp, добавив в него перегруженную функцию f() (сами выберите аргументы перегруженной версии). Создайте второй указатель на функцию класса, свяжите его с перегруженной версией f() и вызовите функцию через указатель. Как выбирается перегруженная версия в этом случае?
27. Начните с примера FunctionTable.cpp из главы 3. Создайте класс, содержащий вектор (vector) указателей на функции, а также функции add() и remove() для добавления и удаления указателей. Создайте функцию run(), которая перебирает элементы вектора и вызывает все функции.
28. Измените упражнение 27 так, чтобы в векторе хранились указатели на функции класса.

Перегрузка операторов

Механизм перегрузки операторов — не более чем альтернативный и более удобный способ вызова функций. Между «вызовом» оператора и обычным вызовом функции существуют два основных различия. Во-первых, отличается синтаксис; при вызове оператора его идентификатор обычно помещается между аргументами (а иногда — после них). Во-вторых, компилятор сам выбирает, какую «функцию» следует вызвать для данного оператора. Например, если оператор + используется с двумя вещественными аргументами, то компилятор вызывает функцию вещественного сложения (обычно этот «вызов» сводится к вставке небольшого фрагмента кода или команды сопроцессора). Если оператор + вызывается для вещественного и для целого числа, то компилятор сначала вызывает специальную функцию для преобразования int в тип float, а затем снова вызывает код вещественного сложения.

Но в языке C++ стало возможным определение новых операторов, работающих с классами. Определение оператора практически не отличается от определения обычной функции; только имя функции состоит из ключевого слова operator, за которым следует идентификатор оператора. Другие различия отсутствуют. Оператор становится рядовой функцией, которая вызывается компилятором, когда он встречает подходящую сигнатуру.

Предупреждение

На первых порах легко увлечься перегрузкой операторов — что и говорить, игрушка занятная. Но помните: это *всего лишь* «синтаксический сахар», другой способ вызова функций. Следовательно, перегружать операторы стоит только тогда, когда перегрузка заметно упрощает написание или, что еще важнее, *чтение* программ с вашим классом (не забывайте, что программы читаются гораздо чаще, чем пишутся). Если это условие не выполняется, не тратьте время попусту.

Некоторые программисты от перегрузки операторов склонны впадать в панику; знакомые операторы C вдруг утрачивают знакомый смысл. «Все изменилось, и моя

программа на С будет работать не так, как раньше...» Но это неверно. Все операторы в выражениях, содержащих только встроенные типы данных, изменяться не могут. Вам ни за что не удастся переопределить смысл оператора в таком выражении:

```
1 << 4
```

Также не получится заставить работать по-новому следующее выражение:

```
1.414 << 2
```

Перегруженные операторы применимы только к выражениям, содержащим пользовательские типы.

Синтаксис

Определение перегруженного оператора очень похоже на определение функции с именем operator@, где @ — идентификатор перегружаемого оператора. Количество аргументов в списке перегруженного оператора зависит от двух факторов.

- Категории оператора — унарный (один аргумент) или бинарный (два аргумента).
- Способа определения оператора — в виде глобальной функции (один аргумент для унарных, два для бинарных операторов) или функции класса (для унарных операторов аргументы отсутствуют, для бинарных операторов один аргумент, при этом объект класса становится левосторонним аргументом).

Следующий маленький класс демонстрирует синтаксис перегрузки операторов:

```
//: C12:OperatorOverloadingSyntax.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii) : i(ii) {}
  const Integer
  operator+(const Integer& rv) const {
    cout << "operator+" << endl;
    return Integer(i + rv.i);
  }
  Integer&
  operator+=(const Integer& rv) {
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
  }
};

int main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  Integer ii(1), jj(2), kk(3);
  kk += ii + jj;
} ///:~
```

Два перегруженных оператора определяются в виде подставляемых функций класса, которые выводят сообщение о вызове. Единственный аргумент определяет значение, находящееся в правой части бинарных операторов. Унарные операторы, определяемые в виде функций класса, не имеют аргументов — функция вызывается для объекта, находящегося слева от оператора.

Все операторы, кроме условных (которые обычно возвращают логическое значение), почти всегда возвращают объект или ссылку на тип, к которому относятся оба аргумента (для разнотипных аргументов вы сами выбираете интерпретацию результата). Это позволяет строить сложные выражения вида

```
kk += ii + jj;
```

Функция operator+ создает новый (временный) объект Integer, который используется в качестве правостороннего аргумента для функции operator+=. Как только временный объект становится ненужным, он немедленно уничтожается.

Перегружаемые операторы

Хотя C++ позволяет перегружать почти все операторы, доступные в C, возможности перегрузки серьезно ограничены. В частности, запрещены комбинации операторов, которые в настоящее время не имеют смысла в C (например, ** для возведения в степень), отсутствует возможность изменения приоритета или количества аргументов у операторов. Впрочем, такие ограничения оправданны — любое из этих действий приведет к появлению операторов, которые не проясняют, а лишь запутывают программы.

В следующих двух подразделах приводятся примеры всех «стандартных» операторов и их перегрузки в тех формах, которые чаще всего применяются на практике.

Унарные операторы

Ниже продемонстрирован синтаксис перегрузки всех унарных операторов, как в форме глобальных функций (дружественных функций, не принадлежащих классам), так и в форме функций классов. При перегрузке задействован класс Integer, приведенный выше, и новый класс byte. Смысл конкретной версии оператора полностью зависит от того, как вы собираетесь его использовать, но прежде чем делать что-то экстравагантное, подумайте о прикладном программисте.

Итак, рассмотрим каталог всех унарных функций:

```
//: C12:OverloadingUnaryOperators.cpp
#include <iostream>
using namespace std;

// Глобальные функции:
class Integer {
  long i;
  Integer* This() { return this; }
public:
  Integer(long ll = 0) : i(ll) {}
  // Побочные эффекты отсутствуют, оператор получает аргумент const&:
  friend const Integer&
    operator+(const Integer& a);
```

```
  friend const Integer
    operator-(const Integer& a);
  friend const Integer
    operator~(const Integer& a);
  friend Integer*
    operator&(Integer& a);
  friend int
    operator!(const Integer& a);
  // Побочные эффекты, аргумент отличен от const&:
  // Префиксная версия:
  friend const Integer&
    operator++(Integer& a);
  // Постфиксная версия:
  friend const Integer
    operator++(Integer& a, int);
  // Префиксная версия:
  friend const Integer&
    operator--(Integer& a);
  // Постфиксная версия:
  friend const Integer
    operator--(Integer& a, int);
};

// Глобальные операторы:
const Integer& operator+(const Integer& a) {
  cout << "+Integer\n";
  return a; // Унарный оператор + ничего не делает
}
const Integer operator-(const Integer& a) {
  cout << "-Integer\n";
  return Integer(-a.i);
}
const Integer operator~(const Integer& a) {
  cout << "~Integer\n";
  return Integer(~a.i);
}
Integer* operator&(Integer& a) {
  cout << "&Integer\n";
  return a.This(); // &a is recursive!
}
int operator!(const Integer& a) {
  cout << "!Integer\n";
  return !a.i;
}
// Префиксная версия; возвращает значение после инкремента
const Integer& operator++(Integer& a) {
  cout << "++Integer\n";
  a.i++;
  return a;
}
// Постфиксная версия; возвращает значение перед инкрементом
const Integer operator++(Integer& a, int) {
  cout << "Integer++\n";
  Integer before(a.i);
  a.i++;
  return before;
}
// Префиксная версия; возвращает значение после декремента
const Integer& operator--(Integer& a) {
```

```
    cout << "--Integer\n";
    a.i--;
    return a;
  }
  // Постфиксная версия; возвращает значение перед декрементом
  const Integer operator--(Integer& a, int) {
    cout << "Integer--\n";
    Integer before(a.i);
    a.i--;
    return before;
  }

  // Проверка работы перегруженных операторов:
  void f(Integer a) {
    +a;
    -a;
    ~a;
    Integer* ip = &a;
    !a;
    ++a;
    a++;
    --a;
    a--;
  }

  // Функции классов (с неявным аргументом "this"):
  class Byte {
    unsigned char b;
  public:
    Byte(unsigned char bb = 0) : b(bb) {}
    // Побочные эффекты отсутствуют: константная функция класса:
    const Byte& operator+() const {
      cout << "+Byte\n";
      return *this;
    }
    const Byte operator-() const {
      cout << "-Byte\n";
      return Byte(-b);
    }
    const Byte operator~() const {
      cout << "~Byte\n";
      return Byte(~b);
    }
    Byte operator!() const {
      cout << "!Byte\n";
      return Byte(!b);
    }
    Byte* operator&() {
      cout << "&Byte\n";
      return this;
    }
    // Побочные эффекты: неконстантная функция класса
    const Byte& operator++() { // Префиксная версия
      cout << "++Byte\n";
      b++;
      return *this;
    }
    const Byte operator++(int) { // Постфиксная версия
```

```
    cout << "Byte++\n";
    Byte before(b);
    b++;
    return before;
  }
  const Byte& operator--() { // Префиксная версия
    cout << "--Byte\n";
    --b;
    return *this;
  }
  const Byte operator--(int) { // Постфиксная версия
    cout << "Byte--\n";
    Byte before(b);
    --b;
    return before;
  }
};

void g(Byte b) {
  +b;
  -b;
  ~b;
  Byte* bp = &b;
  !b;
  ++b;
  b++;
  --b;
  b--;
}

int main() {
  Integer a;
  f(a);
  Byte b;
  g(b);
} ///:~
```

Функции сгруппированы в соответствии со способом передачи аргументов. Рекомендации относительно того, как организуется передача и возврат данных, будут приведены далее. Представленные формы (а также формы, показанные в следующем подразделе) часто встречаются на практике, поэтому их стоит взять за образец при самостоятельной перегрузке операторов.

Инкремент и декремент

С перегрузкой операторов ++ и -- возникают небольшие затруднения, поскольку эти операторы должны вызывать разные функции в зависимости от того, расположены ли они до (префиксная запись) или после (постфиксная запись) того объекта, с которым они работают. Проблема решается просто, хотя у новичков это решение все равно поначалу вызывает затруднения. Так, когда компилятор видит выражение ++a (префиксный инкремент), он генерирует вызов функции operator++(a), а для выражения a++ генерируется вызов operator++(a,int). Таким образом, компилятор различает эти две формы и вызывает для них разные перегруженные функции. В программе OverloadingUnaryOperators.cpp, если компилятор видит выражение ++b, он для функций классов генерирует вызов B::operator++(), а для выражения b++ — вызов B::operator++(int).

Пользователь видит лишь то, что для префиксной и постфиксной версий вызываются разные функции. Но во внутренней реализации две функции должны обладать разными сигнатурами и поэтому связываются с разными телами функций. Чтобы создать другую сигнатуру для постфиксной версии, компилятор передает в дополнительном аргументе int фиктивную константу (идентификатор этой константе не присваивается, потому что ее значение нигде не используется).

Бинарные операторы

В следующем листинге пример OverloadingUnaryOperators.cpp повторяется для бинарных операторов. Таким образом, у вас под рукой будут образцы всех операторов, которые обычно приходится перегружать. Как и в предыдущем случае, приводятся обе формы: глобальные функции и функции классов.

```
//: C12:Integer.h
// Глобальные перегруженные операторы
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>

// Глобальные функции:
class Integer {
  long i;
public:
  Integer(long ll = 0) : i(ll) {}
  // Операторы, создающие новое, модифицированное значение:
  friend const Integer
    operator+(const Integer& left,
              const Integer& right);
  friend const Integer
    operator-(const Integer& left,
              const Integer& right);
  friend const Integer
    operator*(const Integer& left,
              const Integer& right);
  friend const Integer
    operator/(const Integer& left,
              const Integer& right);
  friend const Integer
    operator%(const Integer& left,
              const Integer& right);
  friend const Integer
    operator^(const Integer& left,
              const Integer& right);
  friend const Integer
    operator&(const Integer& left,
              const Integer& right);
  friend const Integer
    operator|(const Integer& left,
              const Integer& right);
  friend const Integer
    operator<<(const Integer& left,
               const Integer& right);
  friend const Integer
    operator>>(const Integer& left,
               const Integer& right);
```

```
// Комбинированные операторы присваивания
// изменяют и возвращают l-значение:
friend Integer&
  operator+=(Integer& left,
             const Integer& right);
friend Integer&
  operator-=(Integer& left,
             const Integer& right);
friend Integer&
  operator*=(Integer& left,
             const Integer& right);
friend Integer&
  operator/=(Integer& left,
             const Integer& right);
friend Integer&
  operator%=(Integer& left,
             const Integer& right);
friend Integer&
  operator^=(Integer& left,
             const Integer& right);
friend Integer&
  operator&=(Integer& left,
             const Integer& right);
friend Integer&
  operator|=(Integer& left,
             const Integer& right);
friend Integer&
  operator>>=(Integer& left,
              const Integer& right);
friend Integer&
  operator<<=(Integer& left,
              const Integer& right);
// Условные операторы возвращают true или false:
friend int
  operator==(const Integer& left,
             const Integer& right);
friend int
  operator!=(const Integer& left,
             const Integer& right);
friend int
  operator<(const Integer& left,
            const Integer& right);
friend int
  operator>(const Integer& left,
            const Integer& right);
friend int
  operator<=(const Integer& left,
             const Integer& right);
friend int
  operator>=(const Integer& left,
             const Integer& right);
friend int
  operator&&(const Integer& left,
             const Integer& right);
friend int
  operator||(const Integer& left,
             const Integer& right);
// Вывод содержимого в выходной поток:
void print(std::ostream& os) const { os << i; }
```

```
};
#endif // INTEGER_H ///:~

//: C12:Integer.cpp {O}
// Реализация перегруженных операторов
#include "Integer.h"
#include "../require.h"

const Integer
  operator+(const Integer& left,
            const Integer& right) {
  return Integer(left.i + right.i);
}
const Integer
  operator-(const Integer& left,
            const Integer& right) {
  return Integer(left.i - right.i);
}
const Integer
  operator*(const Integer& left,
            const Integer& right) {
  return Integer(left.i * right.i);
}
const Integer
  operator/(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "divide by zero");
  return Integer(left.i / right.i);
}
const Integer
  operator%(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "modulo by zero");
  return Integer(left.i % right.i);
}
const Integer
  operator^(const Integer& left,
            const Integer& right) {
  return Integer(left.i ^ right.i);
}
const Integer
  operator&(const Integer& left,
            const Integer& right) {
  return Integer(left.i & right.i);
}
const Integer
  operator|(const Integer& left,
            const Integer& right) {
  return Integer(left.i | right.i);
}
const Integer
  operator<<(const Integer& left,
             const Integer& right) {
  return Integer(left.i << right.i);
}
const Integer
  operator>>(const Integer& left,
             const Integer& right) {
  return Integer(left.i >> right.i);
```

```
}
// Комбинированные операторы присваивания
// изменяют и возвращают l-значение:
Integer& operator+=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* самоприсваивание */}
   left.i += right.i;
   return left;
}
Integer& operator-=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* самоприсваивание */}
   left.i -= right.i;
   return left;
}
Integer& operator*=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* самоприсваивание */}
   left.i *= right.i;
   return left;
}
Integer& operator/=(Integer& left,
                    const Integer& right) {
   require(right.i != 0, "divide by zero");
  if(&left == &right) {/* самоприсваивание */}
left.i /= right.i;
return left;
}
Integer& operator%=(Integer& left,
const Integer& right) {
require(right.i != 0, "modulo by zero");
if(&left == &right) {/* самоприсваивание */}
left.i %= right.i;
return left;
}
Integer& operator^=(Integer& left,
const Integer& right) {
if(&left == &right) {/* самоприсваивание */}
left.i ^= right.i;
return left;
}
Integer& operator&=(Integer& left,
const Integer& right) {
if(&left == &right) {/* самоприсваивание */}
left.i &= right.i;
return left;
}
Integer& operator|=(Integer& left,
const Integer& right) {
if(&left == &right) {/* самоприсваивание */}
left.i |= right.i;
return left;
}
Integer& operator>>=(Integer& left,
const Integer& right) {
if(&left == &right) {/* самоприсваивание */}
left.i >>= right.i;
return left;
}
```

```
Integer& operator<<=(Integer& left,
const Integer& right) {
if(&left == &right) {/* самоприсваивание */}
left.i <<= right.i;
return left;
}
// Условные операторы возвращают true или false:
int operator==(const Integer& left,
const Integer& right) {
return left.i == right.i;
}
int operator!=(const Integer& left,
const Integer& right) {
return left.i != right.i;
}
int operator<(const Integer& left,
const Integer& right) {
return left.i < right.i;
}
int operator>(const Integer& left,
const Integer& right) {
return left.i > right.i;
}
int operator<=(const Integer& left,
const Integer& right) {
return left.i <= right.i;
}
int operator>=(const Integer& left,
const Integer& right) {
return left.i >= right.i;
}
int operator&&(const Integer& left,
const Integer& right) {
return left.i && right.i;
}
int operator||(const Integer& left,
const Integer& right) {
return left.i || right.i;
} ///:~
```

```
//: C12:IntegerTest.cpp
//{L} Integer
#include "Integer.h"
#include <fstream>
using namespace std;
ofstream out("IntegerTest.out");

void h(Integer& c1, Integer& c2) {
// Сложное выражение:
c1 += c1 * c2 + c2 % c1;
#define TRY(OP) \
out << "c1 = "; c1.print(out); \
out << ", c2 = "; c2.print(out); \
out << "; c1 " #OP " c2 produces "; \
(c1 OP c2).print(out); \
out << endl;
TRY(+) TRY(-) TRY(*) TRY(/)
TRY(%) TRY(^) TRY(&) TRY(|)
TRY(<<) TRY(>>) TRY(+=) TRY(-=)
```

```
TRY(*=) TRY(/=) TRY(%=) TRY(^=)
TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
// Условные операторы:
#define TRYC(OP) \
out << "c1 = "; c1.print(out); \
out << ", c2 = "; c2.print(out); \
out << "; c1 " #OP " c2 produces "; \
out << (c1 OP c2); \
out << endl;
TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
TRYC(>=) TRYC(&&) TRYC(||)
}

int main() {
cout << "friend functions" << endl;
Integer c1(47), c2(9);
h(c1, c2);
} ///:~

//: C12:Byte.h
// Перегрузка операторов в форме функций класса
#ifndef BYTE_H
#define BYTE_H
#include "../require.h"
#include <iostream>
// Функции классов (с неявным аргументом "this"):
class Byte {
unsigned char b;
public:
Byte(unsigned char bb = 0) : b(bb) {}
// Побочные эффекты отсутствуют: константная функция класса:
const Byte
operator+(const Byte& right) const {
return Byte(b + right.b);
}
const Byte
operator-(const Byte& right) const {
return Byte(b - right.b);
}
const Byte
operator*(const Byte& right) const {
return Byte(b * right.b);
}
const Byte
operator/(const Byte& right) const {
require(right.b != 0, "divide by zero");
return Byte(b / right.b);
}
const Byte
operator%(const Byte& right) const {
require(right.b != 0, "modulo by zero");
return Byte(b % right.b);
}
const Byte
operator^(const Byte& right) const {
return Byte(b ^ right.b);
}
const Byte
operator&(const Byte& right) const {
```

```
return Byte(b & right.b);
}
const Byte
operator|(const Byte& right) const {
return Byte(b | right.b);
}
const Byte
operator<<(const Byte& right) const {
return Byte(b << right.b);
}
const Byte
operator>>(const Byte& right) const {
return Byte(b >> right.b);
}
// Комбинированные операторы присваивания
// изменяют и возвращают l-значение:
// оператор = может быть только функцией класса:
Byte& operator=(const Byte& right) {
// Самоприсваивание:
if(this == &right) return *this;
b = right.b;
return *this;
}
Byte& operator+=(const Byte& right) {
if(this == &right) {/* самоприсваивание */}
b += right.b;
return *this;
}
Byte& operator-=(const Byte& right) {
if(this == &right) {/* самоприсваивание */}
b -= right.b;
return *this;
}
Byte& operator*=(const Byte& right) {
if(this == &right) {/* самоприсваивание */}
b *= right.b;
return *this;
}
Byte& operator/=(const Byte& right) {
require(right.b != 0, "divide by zero");
if(this == &right) {/* самоприсваивание */}
b /= right.b;
return *this;
}
Byte& operator%=(const Byte& right) {
require(right.b != 0, "modulo by zero");
if(this == &right) {/* самоприсваивание */}
b %= right.b;
return *this;
}
Byte& operator^=(const Byte& right) {
if(this == &right) {/* самоприсваивание */}
b ^= right.b;
return *this;
}
Byte& operator&=(const Byte& right) {
if(this == &right) {/* самоприсваивание */}
b &= right.b;
return *this;
```

```
}
Byte& operator|=(const Byte& right) {
if(this == &right) {/* самоприсваивание */}
b |= right.b;
return *this;
}
Byte& operator>>=(const Byte& right) {
if(this == &right) {/* самоприсваивание */}
b >>= right.b;
return *this;
}
Byte& operator<<=(const Byte& right) {
if(this == &right) {/* самоприсваивание */}
b <<= right.b;
return *this;
}
// Условные операторы возвращают true или false:
int operator==(const Byte& right) const {
return b == right.b;
}
int operator!=(const Byte& right) const {
return b != right.b;
}
int operator<(const Byte& right) const {
return b < right.b;
}
int operator>(const Byte& right) const {
return b > right.b;
}
int operator<=(const Byte& right) const {
return b <= right.b;
}
int operator>=(const Byte& right) const {
return b >= right.b;
}
int operator&&(const Byte& right) const {
return b && right.b;
}
int operator||(const Byte& right) const {
return b || right.b;
}
// Вывод содержимого в выходной поток:
void print(std::ostream& os) const {
os << "0x" << std::hex << int(b) << std::dec;
}
};
#endif // BYTE_H ///:~

//: C12:ByteTest.cpp
#include "Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

void k(Byte& b1, Byte& b2) {
b1 = b1 * b2 + b2 % b1;

#define TRY2(OP) \
out << "b1 = "; b1.print(out); \
```

```
out << ", b2 = "; b2.print(out); \
out << "; b1 " #OP " b2 produces "; \
(b1 OP b2).print(out); \
out << endl;

b1 = 9; b2 = 47;
TRY2(+) TRY2(-) TRY2(*) TRY2(/)
TRY2(%) TRY2(^) TRY2(&) TRY2(|)
TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
TRY2(=) // Оператор присваивания

// Условные операторы:
#define TRYC2(OP) \
out << "b1 = "; b1.print(out); \
out << ", b2 = "; b2.print(out); \
out << "; b1 " #OP " b2 produces "; \
out << (b1 OP b2); \
out << endl;

b1 = 9; b2 = 47;
TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
TRYC2(>=) TRYC2(&&) TRYC2(||)

// Цепочечное присваивание:
Byte b3 = 92;
b1 = b2 = b3;
}

int main() {
out << "member functions:" << endl;
Byte b1(47), b2(9);
k(b1, b2);
} ///:~
```

Из листинга видно, что оператор = разрешено реализовать только в виде функции класса. Мы еще вернемся к этой теме позднее.

Обратите внимание: все разновидности оператора присваивания содержат проверочный код длясамоприсваивания; это общепринятая практика. Иногда такая проверка не нужна; например, для оператора += часто *реально требуется* использовать выражения A+= A, прибавляя переменную A к самой себе. Проверка путем самоприсваивания особенно важна для оператора =, поскольку в случае сложных объектов неправильное присваивание может привести к катастрофическим последствиям (иногда все обходится благополучно, но при перегрузке оператора = вы всегда должны помнить о проверке путем самоприсваивания).

Все операторы, приведенные в двух предыдущих примерах, перегружаются для работы с одним типом. Также существует возможность перегрузить оператор для работы со смешанными типами — так сказать, «складывать яблоки с апельсинами». Но прежде чем браться за масштабную перегрузку операторов, обязательно просмотрите раздел «Автоматическое приведение типа» этой главы. Нередко своевременное приведение типа избавляет от необходимости определять несколько перегруженных операторов.

Аргументы и возвращаемые значения

Разнообразные способы передачи и приема аргументов в примерах OverloadingUnaryOperators.cpp, Integer.h и Byte.h поначалу вызывают легкое замешательство. В принципе, вы *можете* передавать и возвращать аргументы так, как сочтете нужным, однако варианты в приведенном листинге выбирались не случайно, а по определенной логической схеме. В большинстве случаев вам также следует руководствоваться этой схемой.

- Если аргумент только читается, но не изменяется оператором, его следует передавать как ссылку на константу. Обычные математические операции (сложение, вычитание и т. д.), а также логические операции не изменяют свои аргументы, поэтому передача по ссылке на константу встречается чаще всего. Если функция является членом класса, это правило заменяется объявлением функции класса константной. Только для комбинированных операций (таких, как сложение с присваиванием) и операции присваивания, изменяющих левосторонний аргумент, этот аргумент *не является* константным, но он все равно передается в виде адреса из-за возможной модификации.
- Тип возвращаемого значения выбирается в зависимости от предполагаемого смысла оператора (как упоминалось выше, вы можете интерпретировать аргументы и возвращаемое значение так, как сочтете нужным). Если оператор должен создавать новое значение, то для возвращаемого значения следует создавать новый объект. Например, операторная функция Integer::operator+ создает объект Integer, который интерпретируется как сумма операндов. Объект возвращается по значению как константный, что делает невозможной модификацию результата как l-значения (то есть значения, находящегося в левой части выражения присваивания).
- Все операторы присваивания изменяют левосторонний операнд. Чтобы результат присваивания мог использоваться в цепочках присваивания вида a=b=c, предполагается, что оператор вернет ссылку на тот же левосторонний операнд, который только что был модифицирован. Но должна ли эта ссылка быть ссылкой на константу или нет? Хотя цепочка a=b=c читается слева направо, компилятор обрабатывает ее справа налево, поэтому вы не обязаны возвращать ссылку на неконстантный объект для поддержки цепочечного присваивания. Тем не менее программисты иногда рассчитывают на возможность выполнения операции с результатом присваивания в выражениях вида (a=b).func() (функция func() вызывается для переменной a после присваивания ей b). Следовательно, возвращаемое значение для всех операторов присваивания должно быть ссылкой на неконстантный объект, который может использоваться как l-значение.
- Для логических операторов любой нормальный программист рассчитывает получить в худшем случае int, в лучшем — bool (библиотеки, созданные до повсеместной поддержки встроенного типа C++ bool, используют int или эквивалентное определение typedef).

Операторы инкремента и декремента порождают некоторую двойственность, поскольку они существуют в префиксном и постфиксном вариантах. Обе версии изменяют объект и поэтому не могут интерпретировать его как константу.

Префиксная версия возвращает значение объекта после его изменения, поэтому для нее вполне логично вернуть *this в виде ссылки. Постфиксная версия должна возвращать значение объекта *до* изменения, поэтому в ней приходится создавать отдельный объект, представляющий это значение, и изменять его. Следовательно, чтобы обеспечить предполагаемое поведение постфиксной версии, необходимо возвращать данные по значению (в отдельных ситуациях операторы инкремента и декремента возвращают int или bool, например, как признак достижения конца списка при переборе). Теперь возникает вопрос: должны ли эти значения возвращаться как константы или нет? Если разрешить модификацию объекта, а кто-нибудь использует выражение (++a).func(), то вызов func() будет работать с исходным объектом a, в то же время в выражении (a++).func() вызов func() работает с временным объектом, возвращаемым постфиксным оператором ++. Временные объекты автоматически интерпретируются как константные, поэтому компилятор сообщит об ошибке, но ради логического согласования обеих версий стоит объявить их константными, как это сделано выше. С другой стороны, можно объявить префиксную версию неконстантной, а постфиксную — константной. У операторов инкремента и декремента существует множество разнообразных интерпретаций, поэтому решение приходится принимать в каждом конкретном случае.

Возврат константы по значению

С возвратом по значению константного объекта дело обстоит чуть сложнее, поэтому этот вопрос стоит разобрать подробнее. Рассмотрим бинарный оператор +. В выражениях типа f(a+b) результат a+b становится временным объектом, используемым для вызова f(). Как и всякий временный объект, он автоматически является константным, поэтому возвращаемое значение всегда константно независимо от того, будет оно объявлено таковым или нет.

С другой стороны, вы также можете отправить сообщение возвращаемому значению a+b, вместо того чтобы просто передать его функции. Например, можно использовать выражение (a+b).g(), где g() — некоторая функция класса Integer. Объявляя возвращаемое значение константным, вы указываете, что для него могут вызываться только константные функции класса. Такой подход верен с точки зрения правил константности, поскольку он предотвращает сохранение потенциально нужной информации в объекте, который, по всей вероятности, будет потерян.

Оптимизация возвращаемого значения

Обратите внимание на форму, используемую при создании новых объектов для возврата по значению, например в функции operator+:

```
return Integer(left.i + right.i);
```

На первый взгляд похоже на вызов конструктора, но впечатление обманчиво. Это синтаксис создания временного объекта; команда означает: «Создать временный объект Integer и вернуть его». Кажется, что результат такой же, как если бы мы создали локальный именованный объект и вернули его, но это не так. Если заменить эту команду следующим фрагментом, произойдут три события:

```
Integer tmp(left.i + right.i);
return tmp;
```

Сначала будет создан объект tmp с вызовом конструктора. Затем копирующий конструктор скопирует tmp в область памяти вне возвращаемого значения. Наконец, в конце области видимости для tmp будет вызван деструктор.

Подход с «возвратом временного объекта» работает несколько иначе. Когда компилятор встречает такую конструкцию, он знает, что объект создается только для возврата значения. Компилятор задействует эту информацию и строит объект *непосредственно* в области памяти возвращаемого значения. Для этого достаточно только вызова конструктора; копирующий конструктор не используется, а деструктор не вызывается, потому что локальный объект реально не создается. В сущности, такое решение не требует ничего, кроме внимательности программиста, но обеспечивает довольно заметный выигрыш в эффективности. Иногда это называется *оптимизацией возвращаемого значения*.

Особые операторы

Некоторые операторы обладают специфическим синтаксисом перегрузки.

Так, оператор индексирования [ ] обязательно является функцией класса и вызывается с одним аргументом. Поскольку оператор [ ] подразумевает, что индексируемый объект ведет себя как массив, этот оператор часто возвращает ссылку, чтобы его было удобно использовать в левой части выражения присваивания. Этот оператор перегружается довольно часто; вы неоднократно увидите соответствующие примеры на страницах книги.

Операторы new и delete управляют динамическим выделением памяти; существует несколько вариантов их перегрузки. Эта тема будет рассматриваться в главе 13.

Оператор запятая

Оператор запятая (,) вызывается в том случае, если рядом с объектом, для которого он определен, поставлена запятая. Однако этот оператор не вызывается в списках аргументов функций — он применим только к объектам, находящимся вне списков и разделенным запятыми. Полезных применений у этого оператора немного, он был включен в язык в основном для полноты. Как видно из следующего примера, этот оператор может вызываться и в том случае, когда запятая находится *перед* объектом, а не после него:

```
//: C12:OverloadingOperatorComma.cpp
#include <iostream>
using namespace std;

class After {
public:
const After& operator,(const After&) const {
cout << "After::operator,()" << endl;
return *this;
}
};

class Before {};

Before& operator,(int, Before& b) {
cout << "Before::operator,()" << endl;
return b;
}

int main() {
```

```
After a. b:
a. b: // Вызов функции operator.

Before c:
1. c: // Вызов функции operator.
} ///:~
```

Глобальная функция дает возможность размещать запятую перед объектом, однако такое применение этого оператора выглядит весьма сомнительно. Хотя списки, разделенные запятыми, иногда входят в состав более сложных выражений, обычно их применение в большинстве случаев лишь запутывает синтаксис.

Оператор разыменования указателя

Оператор разыменования указателя (->) обычно перегружается для объектов, имитирующих указатели. Такие объекты превосходят обычные указатели по своим возможностям, поэтому их часто называют *умными*, или *интеллектуальными, указателями*. Основные области применения умных указателей — инкапсуляция указателей в объектах для того, чтобы сделать операции с ними безопасными, а также *итераторы* (объекты, которые перебирают содержимое *коллекций/контейнеров* других объектов и обеспечивают выборку элементов без прямого доступа к реализации контейнера). Контейнеры и итераторы часто встречаются в библиотеках классов, в том числе и в стандартной библиотеке C++, описанной во втором томе книги.

Оператор разыменования указателя должен быть оформлен в виде функции класса. Для него также устанавливаются дополнительные, нетипичные, ограничения: он должен возвращать либо объект (или ссылку на объект), также содержащий оператор разыменования указателя, либо указатель, позволяющий обратиться к объекту, на который «указывает» стрелка оператора. Простой пример:

```
//: C12:SmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std:

class Obj {
static int i. j:
public:
void f() const { cout << i++ << endl: }
void g() const { cout << j++ << endl: }
}:

// Определения статических переменных:
int Obj::i = 47:
int Obj::j = 11:

// Контейнер:
class ObjContainer {
vector<Obj*> a:
public:
void add(Obj* obj) { a.push_back(obj): }
friend class SmartPointer:
}:

class SmartPointer {
```

```
ObjContainer& oc;
int index;
public:
SmartPointer(ObjContainer& objc) : oc(objc) {
index = 0;
}
// Возвращаемое значение содержит признак конца списка:
bool operator++() { // Префиксная версия
if(index >= oc.a.size()) return false;
if(oc.a[++index] == 0) return false;
return true;
}
bool operator++(int) { // Постфиксная версия
return operator++(); // Использование префиксной версии
}
Obj* operator->() const {
require(oc.a[index] != 0, "Zero value "
"returned by SmartPointer::operator->()");
return oc.a[index];
}
};

int main() {
const int sz = 10;
Obj o[sz];
ObjContainer oc;
for(int i = 0; i < sz; i++)
oc.add(&o[i]); // Заполнение контейнера
SmartPointer sp(oc); // Создание итератора
do {
sp->f(); // Вызов оператора разыменования указателя
sp->g();
} while(sp++);
} ///:~
```

Класс Obj определяет объекты, с которыми работает программа, а функции f() и g() просто выводят значения статических переменных класса. Указатели на объекты сохраняются в контейнерах типа ObjContainer функцией add(). Контейнер ObjContainer выглядит как массив указателей, но обратите внимание: класс не содержит функций для получения указателей. Впрочем, класс SmartPointer объявлен дружественным (friend), поэтому ему разрешен доступ к контейнеру. Класс SmartPointer очень похож на усовершенствованный указатель — его можно перемещать вперед оператором ++ (также можно определить оператор --), он не выходит за пределы указываемого им контейнера и позволяет получить значение, на которое он ссылается (посредством оператора разыменования). Учтите, что класс SmartPointer способен работать только с тем контейнером, для которого он создавался; он не является «универсальным» умным указателем. Дополнительная информация об умных указателях, называемых *итераторами*, приводится в последней главе этого тома, а также во втором томе.

После заполнения контейнера oc объектами Obj в функции main() создается указатель SmartPointer sp. Вызовы функций через умный указатель встречаются в выражениях

```
sp->f(); // Вызов оператора разыменования указателя
sp->g();
```

Хотя объект sp в действительности не содержит функций f() и g(), оператор разыменования указателя автоматически вызовет эти функции для Obj*, возвращаемого функцией SmartPointer::operator->. Правильность вызова всех функций проверяется компилятором.

Хотя базовые принципы работы оператора разыменования сложнее, чем других операторов, общая цель остается прежней: предоставить более удобный синтаксис пользователям вашего класса.

Вложенный итератор

Классы умных указателей и итераторов чаще определяются внутри класса, который ими обслуживается. Так, предыдущий пример можно переписать, преобразовав SmartPointer в класс, вложенный в ObjContainer:

```
//: C12:NestedSmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
static int i, j;
public:
void f() { cout << i++ << endl; }
void g() { cout << j++ << endl; }
};

// Определения статических переменных:
int Obj::i = 47;
int Obj::j = 11;

// Контейнер:
class ObjContainer {
vector<Obj*> a;
public:
void add(Obj* obj) { a.push_back(obj); }
class SmartPointer;
friend class SmartPointer;
class SmartPointer {
ObjContainer& oc;
unsigned int index;
public:
SmartPointer(ObjContainer& objc) : oc(objc) {
index = 0;
}
// Возвращаемое значение содержит признак конца списка:
bool operator++() { // Префиксная версия
if(index >= oc.a.size()) return false;
if(oc.a[++index] == 0) return false;
return true;
}
bool operator++(int) { // Постфиксная версия
return operator++(); // Использование префиксной версии
}
Obj* operator->() const {
require(oc.a[index] != 0, "Zero value "
"returned by SmartPointer::operator->()");
return oc.a[index];
```

```
}
};
// Функция для получения умного указателя.
// установленного в начало ObjContainer:
SmartPointer begin() {
return SmartPointer(*this);
}
};

int main() {
const int sz = 10;
Obj o[sz];
ObjContainer oc;
for(int i = 0; i < sz; i++)
oc.add(&o[i]); // Заполнение контейнера
ObjContainer::SmartPointer sp = oc.begin();
do {
sp->f(); // Вызов оператора разыменования указателя
sp->g();
} while(++sp);
} ///:~
```

Мы видим только два отличия, не считая вложения класса. Во-первых, это объявление дружественного класса:

```
class SmartPointer;
friend class SmartPointer;
```

Прежде чем сообщать компилятору, что класс SmartPointer является дружественным, мы должны сообщить о том, что этот класс существует.

Во-вторых, в классе ObjContainer появилась новая функция begin(), которая возвращает объект SmartPointer, установленный в начало последовательности элементов ObjContainer. Хотя эта функция определена лишь для удобства, она следует правилам, используемым в стандартной библиотеке C++.

Оператор ->*

Бинарный оператор ->* работает по тем же правилам, что и остальные бинарные операторы. Он предназначен для имитации поведения встроенных *указателей на функции классов*, описанных в предыдущей главе.

Оператор ->*, как и оператор ->, обычно используется с некоторым объектом, представляющим «умный указатель», хотя в следующем примере его применение было намеренно упрощено, чтобы сделать более понятным. Главное, о чем следует помнить при определении оператора ->*, — он должен возвращать объект, для которого возможен вызов функции operator() с заданными аргументами.

Функция operator(), которая представляет собой операторную функцию для оператора *вызова функций* (( )), должна быть функцией класса с уникальным свойством — у нее может быть произвольное количество аргументов. Это позволяет объекту вести себя так, словно в действительности он является функцией. Хотя теоретически можно определить несколько перегруженных функций operator() с разными аргументами, обычно она используется для типов, имеющих единственную (или, по крайней мере, самую важную) операцию. Во втором томе будет показано, как в стандартной библиотеке C++ оператор вызова функций используется для создания «объектов функций».

Прежде чем создавать оператор ->*, необходимо сначала создать класс с функцией operator() для представления типа объекта, который будет возвращать ->*. Этот класс должен как-то сохранить необходимую информацию, чтобы при вызове operator() (выполняемом автоматически) происходило разыменование указателя на функцию класса для соответствующего объекта. В следующем примере конструктор FunctionObject сохраняет указатель на объект и указатель на функцию класса, которые затем используются функцией operator() для непосредственного вызова функции класса:

```
//: C12:PointerToMemberOperator.cpp
#include <iostream>
using namespace std;

class Dog {
public:
int run(int i) const {
cout << "run\n";
return i;
}
int eat(int i) const {
cout << "eat\n";
return i;
}
int sleep(int i) const {
cout << "ZZZ\n";
return i;
}
typedef int (Dog::*PMF)(int) const;
// оператор ->* должен возвращать объект
// с функцией operator():
class FunctionObject {
Dog* ptr;
PMF pmem;
public:
// Сохранение указателей на объект и на функцию
FunctionObject(Dog* wp, PMF pmf)
: ptr(wp), pmem(pmf) {
cout << "FunctionObject constructor\n";
}
// Вызов функции с использованием указателей на объект и на функцию
int operator()(int i) const {
cout << "FunctionObject::operator()\n";
return (ptr->*pmem)(i); // Вызов
}
};
FunctionObject operator->*(PMF pmf) {
cout << "operator->*" << endl;
return FunctionObject(this, pmf);
}
};

int main() {
Dog w;
Dog::PMF pmf = &Dog::run;
cout << (w->*pmf)(1) << endl;
pmf = &Dog::sleep;
cout << (w->*pmf)(2) << endl;
```

```
pmf = &Dog::eat;
cout << (w->*pmf)(3) << endl;
} ///:~
```

Класс Dog содержит три функции, каждая из которых получает аргумент int и возвращает int. PMF — простой псевдоним (typedef), упрощающий определение указателей на функции класса Dog.

Объект FunctionObject создается и возвращается оператором ->*. Обратите внимание: оператор ->* знает как объект, для которого вызывается указатель на функцию класса (this), так и указатель на функцию. Эти данные передаются конструктору FunctionObject, который сохраняет их в переменных. При вызове функции operator->* компилятор немедленно вызывает функцию operator() для возвращаемого значения функции operator->* и передает аргументы, заданные для функции operator->*. Функция FunctionObject::operator() получает аргументы и вызывает «настоящую» функцию, используя сохраненные данные.

Как и в случае с оператором ->, мы фактически «вклиниваемся» в процесс вызова функции operator->*, что позволяет выполнить дополнительные операции в случае необходимости.

Механизм вызова operator->*, реализованный в этом примере, работает только для функций класса, которые получают и возвращают тип int. Это ограничивает возможности его применения, но попытка создать перегруженные версии для всех возможных комбинаций типов выглядит нереально. К счастью, специально для решения подобных проблем в C++ был включен механизм шаблонов (template), который будет рассмотрен в последней главе этого тома и во втором томе.

Неперегружаемые операторы

Некоторые операторы из стандартного набора C++ перегружаться не могут. Данное ограничение было установлено в основном по соображениям безопасности. Перегрузка этих операторов ставила бы под угрозу механизмы безопасности, усложняла бы программирование или противоречила бы общепринятым правилам.

- Оператор выбора члена класса (.). В настоящее время обращение через точку имеет смысл для любых членов классов, но если разрешить перегрузку этого оператора, то нормальные обращения к членам классов станут невозможными; придется использовать указатели и оператор ->.
- Оператор разыменования указателя на член класса (.*). Причины те же, что и для оператора выбора члена класса (.).
- В C++ отсутствует оператор возведения в степень. Рассматривалась возможность заимствования из языка Fortran популярного оператора возведения в степень (**), но это породило бы проблемы с лексическим разбором программы. Кроме того, поскольку в C нет оператора возведения в степень, было решено, что и в C++ можно обойтись без него (при необходимости всегда можно вызвать функцию). Оператор возведения в степень всего лишь упрощает запись, но не наделяет язык новыми возможностями, которые бы оправдывали усложнение компилятора.
- Пользователь не может определять собственные операторы. Иначе говоря, вы не сможете придумать новый оператор, не входящий в стандартный

языковой набор. Запрет отчасти связан с трудностями определения приоритета новых операторов, а отчасти — с отсутствием реальной необходимости, оправдывающей эти трудности.

- Пользователь не может изменять приоритеты операторов. Приоритеты и так достаточно трудно запомнить, чтобы разрешать программистам менять их по своему усмотрению.

Операторы, не являющиеся членами классов

В некоторых из приведенных примеров операторы могут оформляться как в виде членов классов, так и вне классов, причем особых различий вроде бы не видно. Обычно возникает вопрос, какой из двух вариантов выбрать? Если это ни на что не влияет, лучше оформить операторы как члены классов, чтобы подчеркнуть связь между оператором и его классом. Если левосторонний операнд всегда является объектом текущего класса, такой подход работает нормально.

Но иногда бывает нужно, чтобы левосторонний операнд был объектом другого класса. Самый типичный пример встречается при перегрузке операторов >> и << для потоков ввода-вывода. Поскольку библиотека потоков ввода-вывода является одной из важнейших библиотек C++, вероятно, эти операторы стоит перегрузить для многих из ваших классов, так что эту программу стоит запомнить:

```
//: C12:IostreamOperatorOverloading.cpp
// Пример перегрузки операторов, не являющихся членами классов
#include "../require.h"
#include <iostream>
#include <sstream> // "String streams"
#include <cstring>
using namespace std;

class IntArray {
enum { sz = 5 };
int i[sz];
public:
IntArray() { memset(i, 0, sz* sizeof(*i)); }
int& operator[](int x) {
require(x >= 0 && x < sz,
"IntArray::operator[] out of range");
return i[x];
}
friend ostream&
operator<<(ostream& os, const IntArray& ia);
friend istream&
operator>>(istream& is, IntArray& ia);
};

ostream&
operator<<(ostream& os, const IntArray& ia) {
for(int j = 0; j < ia.sz; j++) {
os << ia.i[j];
if(j != ia.sz -1)
os << ", ";
}
os << endl;
```

```
return os;
}

istream& operator>>(istream& is, IntArray& ia){
for(int j = 0; j < ia.sz; j++)
is >> ia.i[j];
return is;
}

int main() {
stringstream input("47 34 56 92 103");
IntArray I;
input >> I;
I[4] = -1; // Используется перегруженный оператор []
cout << I;
} ///:~
```

Класс также содержит перегруженный оператор [ ], который возвращает ссылку на действительное значение в массиве. Благодаря возврату ссылки следующее выражение не только выглядит более цивилизованно, чем при использовании указателей, но и достигает желаемого эффекта:

```
I[4] = -1;
```

Важно, чтобы перегруженные операторы сдвига передавали и возвращали данные *по ссылке* — их действие должно распространяться на внешние объекты. В определениях функций показанное ниже выражение приводит к вызову *существующих* перегруженных операторных функций (тех, что определены в файле <iostream>):

```
os << os.i[j];
```

В данном случае вызывается функция ostream& operator<<(ostream&,int), потому что ia.i[j] приводится к типу int.

После выполнения всех действий с потоком ввода или вывода оператор возвращает ссылку на поток, что позволяет использовать результат в более сложных выражениях.

В main() используется новый тип потока ввода-вывода stringstream (объявление находится в файле <sstream>). Этот класс получает объект string (который может создаваться из массива char, как показано ранее) и преобразует его в iostream. В предыдущем примере это позволяет проверить работу перегруженных операторов сдвига без открытия файла или ввода данных в командной строке.

Форма операторов ввода и вывода, приведенная в этом примере, считается стандартной. Если вы захотите определить аналогичные операторы для своего класса, скопируйте сигнатуры функций и возвращаемые типы и постройте тело функции по приведенному образцу.

Базовые рекомендации

Роб Мюррей (Rob Murray) предлагает следующие рекомендации по выбору формы оператора[1]:

---

[1] Rob Murray, «C++ Strategies & Tactics», Addison-Wesley, 1993, c. 47.

Оператор	Рекомендуемая форма
Все унарные операторы	Член класса
= ( ) [ ] -> ->*	Обязательно член класса
+= -= /= *= ^= &= |= %= >>= <<=	Член класса
Остальные бинарные операторы	Не член класса

Перегрузка присваивания

У начинающих программистов C++ постоянно возникают недоразумения с присваиванием. Несомненно, это объясняется тем, что присваивание является важнейшей операцией программирования на всех уровнях, вплоть до копирования регистров на уровне машинных команд. Кроме того, вставка в программу оператора = иногда приводит к вызову копирующего конструктора (см. главу 11):

```
MyType b;
MyType a = b;
a = b;
```

Во второй строке *определяется* объект a. Новый объект создается там, где его раньше не было. Вы уже знаете, что компилятор C++ тщательно следит за инициализацией объектов и что в точке определения объекта всегда должен вызываться конструктор. Но какой конструктор? Объект a создается на базе существующего объекта MyType (а именно объекта b справа от оператора присваивания), поэтому существует только один вариант — копирующий конструктор. Хотя формально объект определяется присваиванием, при этом вызывается копирующий конструктор.

В третьей строке дело обстоит иначе. Слева от оператора = находится уже инициализированный объект. Разумеется, для уже созданного объекта конструктор вызывать не нужно. В этом случае для a вызывается функция MyType::operator=, в аргументе которой передается операнд из правой части (класс может содержать несколько функций operator= для разных типов правостороннего аргумента).

Впрочем, процедура создания объекта не ограничивается вызовом копирующего конструктора. Каждый раз, когда вы инициализируете объект, используя оператор = вместо обычного «функционального» вызова конструктора, компилятор ищет подходящий конструктор для типа справа от оператора присваивания:

```
//: C12:CopyingVsInitialization.cpp
class Fi {
public:
Fi() {}
};

class Fee {
public:
Fee(int) {}
Fee(const Fi&) {}
};

int main() {
Fee fee = 1; // Fee(int)
Fi fi;
Fee fum = fi; // Fee(Fi)
} ///:~
```

Имея дело с оператором =, необходимо помнить об этом различии: если объект еще не был создан, то он обязательно проходит инициализацию; в противном случае вызывается функция operator=.

Но еще лучше обойтись без кода, в котором оператор = используется для инициализации, — всегда задействуйте форму с явным вызовом конструктора. В этом случае две показанные в предыдущем примере конструкции с оператором = принимают вид:

```
Fee fee(1):
Fee fum(fi):
```

Такая запись не собьет с толку ваших читателей.

Поведение функции operator=

В заголовочных файлах Integer.h и Byte.h упоминалось о том, что функция operator= может быть только функцией класса. Она неразрывно связана с объектом, находящимся слева от оператора =. Если бы функцию operator= можно было определить глобально, это сделало бы реальными попытки переопределения стандартного поведения оператора =:

```
int operator=(int,MyType): // Глобальное переопределение = запрещено!
```

Компилятор в корне пресекает подобные попытки, требуя, чтобы функция operator= обязательно была функцией класса.

В определении функции operator= вы должны скопировать всю необходимую информацию из правостороннего объекта в текущий объект (то есть объект, для которого вызывается оператор =) и выполнить все действия, в которых воплощается понятие «присваивания» для вашего класса. Для простых объектов это делается элементарно:

```
//: C12:SimpleAssignment.cpp
// Простой оператор operator=()
#include <iostream>
using namespace std:

class Value {
int a, b:
float c:
public:
Value(int aa = 0, int bb = 0, float cc = 0.0)
: a(aa), b(bb), c(cc) {}
Value& operator=(const Value& rv) {
a = rv.a:
b = rv.b:
c = rv.c:
return *this:
}
friend ostream&
operator<<(ostream& os, const Value& rv) {
return os << "a = " << rv.a << ", b = "
<< rv.b << ", c = " << rv.c:
}
}:

int main() {
Value a, b(1, 2, 3.3):
```

```
cout << "a: " << a << endl;
cout << "b: " << b << endl;
a = b;
cout << "a after assignment: " << a << endl;
} ///:~
```

Объект, расположенный слева от оператора =, копирует значения всех переменных из объекта, расположенного справа от оператора =, а затем возвращает ссылку на самого себя, что позволяет строить более сложные выражения.

В приведенном примере допущена распространенная ошибка. При присваивании двух однотипных объектов всегда следует сначала проверить их на самоприсваивание (то есть на присваивание объекта самому себе). В некоторых случаях (в том числе и в этом) присваивание безвредно, но при изменении реализации класса ситуация может измениться. Пусть эта проверка войдет у вас в привычку, иначе когда-нибудь вы забудете о ней, и в программе появятся неуловимые ошибки.

Указатели в классах

Но что произойдет, если объект не так примитивен? Например, если объект содержит указатели на другие объекты? Простое копирование указателя приведет к тому, что в программе появятся два объекта, ссылающихся на одну область памяти. В подобных ситуациях приходится самостоятельно следить за корректностью копирования.

У этой проблемы существуют два стандартных решения. В простейшем варианте данные, на которые ссылается указатель, копируются при выполнении присваивания или конструирования копии. Решение получается простым и прямолинейным:

```
//: C12:CopyingWithPointers.cpp
// Решение проблемы с раздвоением указателей
// посредством дублирования данных, на которые они ссылаются,
// при присваивании или конструировании копий.
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
string nm;
public:
Dog(const string& name) : nm(name) {
cout << "Creating Dog: " << *this << endl;
}
// Сгенерированные копирующий конструктор и оператор = верны.
// Создание Dog по указателю на Dog:
Dog(const Dog* dp, const string& msg)
: nm(dp->nm + msg) {
cout << "Copied dog " << *this << " from "
<< *dp << endl;
}
~Dog() {
cout << "Deleting Dog: " << *this << endl;
}
void rename(const string& newName) {
nm = newName;
```

```
cout << "Dog renamed to: " << *this << endl:
}
friend ostream&
operator<<(ostream& os. const Dog& d) {
return os << "[" << d.nm << "]":
}
}:

class DogHouse {
Dog* p:
string houseName:
public:
DogHouse(Dog* dog. const string& house)
: p(dog). houseName(house) {}
DogHouse(const DogHouse& dh)
: p(new Dog(dh.p. " copy-constructed")).
houseName(dh.houseName
+ " copy-constructed") {}
DogHouse& operator=(const DogHouse& dh) {
// Check for self-assignment:
if(&dh != this) {
p = new Dog(dh.p. " assigned"):
houseName = dh.houseName + " assigned":
}
return *this:
}
void renameHouse(const string& newName) {
houseName = newName:
}
Dog* getDog() const { return p: }
~DogHouse() { delete p: }
friend ostream&
operator<<(ostream& os. const DogHouse& dh) {
return os << "[" << dh.houseName
<< "] contains " << *dh.p:
}
}:

int main() {
DogHouse fidos(new Dog("Fido"). "FidoHouse"):
cout << fidos << endl:
DogHouse fidos2 = fidos: // Конструирование копии
cout << fidos2 << endl:
fidos2.getDog()->rename("Spot"):
fidos2.renameHouse("SpotHouse"):
cout << fidos2 << endl:
fidos = fidos2: // Присваивание
cout << fidos << endl:
fidos.getDog()->rename("Max"):
fidos2.renameHouse("MaxHouse"):
} ///:~
```

Простой класс Dog содержит только объект string, в котором хранится имя объекта (собаки). Тем не менее мы получаем информацию об основных событиях, происходящих с экземплярами Dog, потому что конструкторы и деструкторы выводят сообщения о своем вызове. Обратите внимание: второй конструктор немного похож на копирующий конструктор, только он получает указатель на Dog вместо ссылки, и еще у него есть второй аргумент — сообщение, присоединяемое к идентифи-

катору аргумента Dog. Второй аргумент помогает наблюдать за поведением программы.

Из листинга видно, что при выводе информации функции класса не обращаются к этой информации напрямую, а посылают *this в cout. В свою очередь, это приводит к вызову функции operator<< для ostream. Такая организация весьма полезна, потому что если нам потребуется изменить формат вывода данных Dog (как с выводом символов «[» и «]» в приведенном примере), изменения достаточно внести только в одном месте.

Класс DogHouse содержит указатель Dog*. В нем продемонстрированы четыре функции, которые всегда необходимо определять, если класс содержит указатели: все необходимые «обычные» конструкторы, копирующий конструктор, функцию operator= (либо определяется, либо запрещается) или деструктор. Функция operator= заодно выполняет проверку на самоприсваивание, хотя в данном случае эта проверка не является строго необходимой. Тем самым практически исключается вероятность того, что вы забудете вставить проверку при модификации программы в будущем.

Подсчет ссылок

В приведенном примере копирующий конструктор и оператор = создают новую копию данных, на которые ссылается указатель, а деструктор эту копию удаляет. Но если объект занимает много памяти или долго инициализируется, копирования хотелось бы избежать. Стандартное решение таких проблем называют *подсчетом ссылок*. Объект, на который ссылается указатель, знает, сколько всего объектов на него ссылается. В этом случае в процессе конструирования копии или присваивания новый указатель связывается с существующим объектом, а счетчик ссылок увеличивается. При разрыве связи указателя с объектом счетчик ссылок уменьшается, а когда он становится равным нулю — уничтожается сам объект.

Но что, если вы хотите модифицировать объект (Dog в предыдущем примере)? Этот экземпляр может использоваться несколькими объектами, поэтому вы измените не только свою версию Dog, но и чью-то чужую, а воспитанные люди так не поступают. Для решения проблемы требуется другая методика, которая называется *копированием при записи*. Прежде чем записывать что-либо в блок памяти, вы сначала убеждаетесь в том, что он никем не используется. Если счетчик ссылок больше 1, то перед записью вы должны создать собственную копию этого блока, чтобы не разрушить чужие данные. Следующий простой пример демонстрирует применение механизмов подсчета ссылок и копирования при записи:

```
//: C12:ReferenceCounting.cpp
// Подсчет ссылок, копирование при записи
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
string nm;
int refcount;
Dog(const string& name)
: nm(name), refcount(1) {
cout << "Creating Dog: " << *this << endl;
```

```
}
// Запрет присваивания:
Dog& operator=(const Dog& rv);
public:
// Объекты Dog могут создаваться только в куче:
static Dog* make(const string& name) {
return new Dog(name);
}
Dog(const Dog& d)
: nm(d.nm + " copy"), refcount(1) {
cout << "Dog copy-constructor: "
<< *this << endl;
}
~Dog() {
cout << "Deleting Dog: " << *this << endl;
}
void attach() {
++refcount;
cout << "Attached Dog: " << *this << endl;
}
void detach() {
require(refcount != 0);
cout << "Detaching Dog: " << *this << endl;
// Если объект никем не используется, уничтожить его:
if(--refcount == 0) delete this;
}
// Условное копирование объекта Dog.
// Вызывается перед модификацией Dog, полученный указатель
// присваивается Dog*.
Dog* unalias() {
cout << "Unaliasing Dog: " << *this << endl;
// Не копировать, если на объект существует единственная ссылка:
if(refcount == 1) return this;
--refcount;
// Копирование выполняется при помощи копирующего конструктора:
return new Dog(*this);
}
void rename(const string& newName) {
nm = newName;
cout << "Dog renamed to: " << *this << endl;
}
friend ostream&
operator<<(ostream& os, const Dog& d) {
return os << "[" << d.nm << "], rc = "
<< d.refcount;
}
};

class DogHouse {
Dog* p;
string houseName;
public:
DogHouse(Dog* dog, const string& house)
: p(dog), houseName(house) {
cout << "Created DogHouse: "<< *this << endl;
}
DogHouse(const DogHouse& dh)
: p(dh.p),
houseName("copy-constructed " +
```

```
dh.houseName) {
p->attach();
cout << "DogHouse copy-constructor: "
<< *this << endl;
}
DogHouse& operator=(const DogHouse& dh) {
// Проверка на самоприсваивание:
if(&dh != this) {
houseName = dh.houseName + " assigned";
// Сначала разрывается связь с используемым объектом:
p->detach();
p = dh.p; // Как копирующий конструктор
p->attach();
}
cout << "DogHouse operator= : "
<< *this << endl;
return *this;
}
// Уменьшение счетчика ссылок, условное уничтожение
~DogHouse() {
cout << "DogHouse destructor: "
<< *this << endl;
p->detach();
}
void renameHouse(const string& newName) {
houseName = newName;
}
void unalias() { p = p->unalias(); }
// Копирование при записи. Перед каждой модификацией
// объекта, связанного с указателем, необходимо сначала
// вызвать для него unalias():
void renameDog(const string& newName) {
unalias();
p->rename(newName);
}
Dog* getDog() {
unalias();
return p;
}
friend ostream&
operator<<(ostream& os, const DogHouse& dh) {
return os << "[" << dh.houseName
<< "] contains " << *dh.p;
}
};

int main() {
DogHouse
fidos(Dog::make("Fido"), "FidoHouse"),
spots(Dog::make("Spot"), "SpotHouse");
cout << "Entering copy-construction" << endl;
DogHouse bobs(fidos);
cout << "After copy-constructing bobs" << endl;
cout << "fidos:" << fidos << endl;
cout << "spots:" << spots << endl;
cout << "bobs:" << bobs << endl;
cout << "Entering spots = fidos" << endl;
spots = fidos;
cout << "After spots = fidos" << endl;
```

```
cout << "spots:" << spots << endl;
cout << "Entering self-assignment" << endl;
bobs = bobs;
cout << "After self-assignment" << endl;
cout << "bobs:" << bobs << endl;
// Закомментируйте следующие строки:
cout << "Entering rename(\"Bob\")" << endl;
bobs.getDog()->rename("Bob");
cout << "After rename(\"Bob\")" << endl;
} ///:~
```

В классе DogHouse хранится указатель на объект Dog. Этот объект содержит счетчик ссылок, а также функции для управления счетчиком ссылок и получения его текущего значения. Копирующий конструктор создает новый объект Dog на базе существующего объекта.

Функция attach() увеличивает счетчик ссылок экземпляра Dog, указывая на его использование новым объектом. Функция detach() уменьшает счетчик ссылок. Если счетчик уменьшается до нуля, значит, объект никем не используется, поэтому функция класса уничтожает собственный объект вызовом delete this.

Прежде чем вносить какие-либо изменения (например, изменять строковый идентификатор Dog), необходимо убедиться в том, что изменяемый экземпляр не используется другим объектом. Для этого вызывается функция DogHouse::unalias(), которая, в свою очередь, вызывает Dog::unalias(). Последняя функция возвращает существующий указатель на Dog, если счетчик ссылок равен 1 (то есть при отсутствии других указателей на этот экземпляр). Если количество ссылок больше 1, экземпляр Dog копируется.

Копирующий конструктор не пытается выделять память заново, а копирует указатель на Dog из исходного объекта в создаваемый. Затем, поскольку теперь этот блок памяти используется дополнительным объектом, он увеличивает счетчик ссылок вызовом Dog::attach().

Оператор = работает с левосторонним объектом, который был создан ранее, поэтому он сначала должен вызвать функцию detach() для старого объекта Dog, что приводит к уничтожению этого объекта, если он не используется другими объектами. Затем оператор = повторяет работу копирующего конструктора. Обратите внимание: сначала мы проверяем, не присваивается ли объект сам себе.

Деструктор выполняет условное уничтожение Dog вызовом функции detach().

Чтобы реализовать копирование при записи, необходимо установить полный контроль над всеми операциями записи в блок памяти. Например, функция renameDog() позволяет изменить данные в блоке памяти, но сначала она вызывает функцию unalias(), чтобы предотвратить модификацию общего экземпляра Dog (то есть экземпляра, указатели на который хранятся в нескольких объектах DogHouse). При получении указателя на Dog из DogHouse() его следует также предварительно разыменовать.

Внутри main() тестируются различные функции, задействованные в реализации подсчета ссылок: конструктор, копирующий конструктор, функция operator= и деструктор. Кроме того, вызов функции renameDog() проверяет правильность копирования при записи.

Результат выполнения программы (слегка отформатированный):

```
Creating Dog: [Fido], rc = 1
Created DogHouse: [FidoHouse]
```

```
contains [Fido]. rc = 1
Creating Dog: [Spot]. rc = 1
Created DogHouse: [SpotHouse]
contains [Spot]. rc = 1
Entering copy-construction
Attached Dog: [Fido]. rc = 2
DogHouse copy-constructor:
[copy-constructed FidoHouse]
contains [Fido]. rc = 2
After copy-constructing bobs
fidos:[FidoHouse] contains [Fido]. rc = 2
spots:[SpotHouse] contains [Spot]. rc = 1
bob:[copy-constructed FidoHouse]
contains [Fido]. rc = 2
Entering spots = fidos
Detaching Dog: [Spot]. rc = 1
Deleting Dog: [Spot]. rc = 0
Attached Dog: [Fido]. rc = 3
DogHouse operator= : [FidoHouse assigned]
contains [Fido]. rc = 3
After spots = fidos
spots:[FidoHouse assigned] contains [Fido].rc = 3
Entering self-assignment
DogHouse operator= : [copy-constructed FidoHouse]
contains [Fido]. rc = 3
After self-assignment
bobs:[copy-constructed FidoHouse]
contains [Fido]. rc = 3
Entering rename("Bob")
After rename("Bob")
DogHouse destructor: [copy-constructed FidoHouse]
contains [Fido]. rc = 3
Detaching Dog: [Fido]. rc = 3
DogHouse destructor: [FidoHouse assigned]
contains [Fido]. rc = 2
Detaching Dog: [Fido]. rc = 2
DogHouse destructor: [FidoHouse]
contains [Fido]. rc = 1
Detaching Dog: [Fido]. rc = 1
Deleting Dog: [Fido]. rc = 0
```

Изучение результатов, анализ исходного текста и самостоятельные эксперименты с этой программой углубят ваше понимание этой методики.

Автоматическое создание функции operator=

Поскольку большинство программистов полагают, что объект всегда можно присвоить другому объекту *того же типа*, компилятор автоматически генерирует функцию **type::operator=(type)**, если она не была определена явно. Поведение этого оператора присваивания имитирует поведение автоматически созданного копирующего конструктора: если класс содержит вложенные объекты (или является производным от другого класса), сгенерированный оператор рекурсивно вызывает функцию **operator=** для этих объектов. Пример:

```
//: C12:AutomaticOperatorEquals.cpp
#include <iostream>
using namespace std;

class Cargo {
```

```
public:
Cargo& operator=(const Cargo&) {
cout << "inside Cargo::operator=()" << endl;
return *this;
}
};

class Truck {
Cargo b;
};

int main() {
Truck a, b;
a = b; // Prints: "inside Cargo::operator=()"
} ///:~
```

Автоматически сгенерированная функция operator= класса Truck вызывает функцию Cargo::operator=.

Как правило, поручать определение operator= компилятору не рекомендуется. Для сколько-нибудь сложных классов (особенно содержащих указатели!) функция operator= должна определяться явно. А если вы хотите полностью запретить присваивание, объявите operator= закрытой функцией (при этом определять эту функцию не обязательно, если только она не используется внутри класса).

Автоматическое приведение типа

Когда компилятор С или С++ встречает выражение или вызов функции, в котором используется не совсем тот тип, который ему нужен, нередко он может привести используемый тип к нужному типу. В С++ можно добиться аналогичного эффекта для пользовательских типов, определяя функции автоматического приведения типа. Эти функции существуют в двух разновидностях: особый тип конструктора и перегруженный оператор.

Приведение типа с использованием конструктора

Если определить конструктор, единственный аргумент которого является объектом (или ссылкой) на другой тип, это позволит компилятору выполнять автоматическое приведение типа. Пример:

```
//: C12:AutomaticTypeConversion.cpp
// Конструктор, обеспечивающий приведение типа
class One {
public:
One() {}
};

class Two {
public:
Two(const One&) {}
};

void f(Two) {}

int main() {
```

```
One one:
f(one): // Должно быть Two. передается One
} ///:~
```

Встречая вызов f() с объектом One, компилятор обращается к объявлению f() и видит, что функция должна получать Two. Далее компилятор выясняет, можно ли как-то сделать Two из One, находит конструктор Two::Two(One) и незаметно вызывает его. Полученный объект Two передается функции f().

В этом случае автоматическое приведение типа избавляет вас от хлопот с определением двух перегруженных версий f(). С другой стороны, за это приходится расплачиваться скрытым вызовом конструктора Two, что может быть существенно, если вы беспокоитесь об эффективности вызовов f().

Запрет на приведение типа с использованием конструктора

Иногда автоматическое приведение типа с использованием конструктора порождает лишние проблемы. Чтобы запретить его, начните определение конструктора с ключевого слова explicit (это ключевое слово используется только с конструкторами). Измененный вариант конструктора класса Two из предыдущего примера выглядит так:

```
//: C12:ExplicitKeyword.cpp
// Ключевое слово "explicit"
class One {
public:
One() {}
}:

class Two {
public:
explicit Two(const One&) {}
}:

void f(Two) {}

int main() {
One one:
//! f(one): // Автоматическое приведение типа запрещено
f(Two(one)): // Можно -- приведение типа выполняется пользователем
} ///:~
```

Определяя конструктор Two с ключевым словом explicit, мы запрещаем компилятору выполнять какие-либо операции по автоматическому приведению типа с использованием этого конкретного конструктора (другие конструкторы класса, у которых слово explicit отсутствует, по-прежнему могут задействоваться для автоматических преобразований). Если пользователь хочет выполнить приведение типа, он должен явно указать на это обстоятельство в программе. В рассмотренном фрагменте вызов f(Two(one)) создает временный объект типа Two из объекта one, как это делал компилятор в предыдущей версии.

Оператор приведения типа

Второй способ автоматического приведения типа основан на перегрузке операторов. Вы создаете функцию класса, которая получает текущий тип и приводит его к нужному типу; имя функции состоит из ключевого слова operator и типа, к которо-

му осуществляется приведение. Эта форма перегрузки оператора уникальна тем, что для нее не указывается тип возвращаемого значения, — он определяется *именем* перегружаемого оператора. Пример:

```
//: C12:OperatorOverloadingConversion.cpp
class Three {
int i;
public:
Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
int x;
public:
Four(int xx) : x(xx) {}
operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
Four four(1);
g(four);
g(1); // Вызывает Three(1,0)
} ///:~
```

При использовании конструктора преобразование выполняется классом, *к которому приводится тип*, а при использовании операторов преобразование выполняется *исходным классом*. Ценность приведения типа с конструктором заключается в том, что при создании нового класса в существующую систему можно включить новую «ветку» преобразования. Однако создание конструктора с одним аргументом (и даже конструктора с несколькими аргументами, если для остальных аргументов заданы значения по умолчанию) *всегда* определяет автоматическое приведение типа. Иногда это оказывается нежелательным, и тогда автоматическое приведение можно запретить при помощи ключевого слова explicit. Кроме того, приведение типа с конструктором не годится для перехода от пользовательского типа к встроенному типу; это возможно только путем перегрузки операторов.

Рефлексивность

Одним из важнейших доводов в пользу использования глобальных перегруженных операторов вместо операторных функций классов является то, что в глобальных версиях автоматическое приведение типа может применяться к обоим операндам, тогда как в функциях классов левосторонний операнд должен изначально относиться к нужному типу. Если преобразуются оба операнда, глобальная версия сэкономит немало усилий. Рассмотрим небольшой пример:

```
//: C12:ReflexivityInOverloading.cpp
class Number {
int i;
public:
Number(int ii = 0) : i(ii) {}
const Number
operator+(const Number& n) const {
return Number(i + n.i);
}
```

```
friend const Number
operator-(const Number&, const Number&);
};

const Number
operator-(const Number& n1,
const Number& n2) {
return Number(n1.i - n2.i);
}

int main() {
Number a(47), b(11);
a + b; // OK
a + 1; // Второй аргумент приводится к Number
//! 1 + a; // Ошибка! Первый аргумент не относится к типу Number
a - b; // OK
a - 1; // Второй аргумент приводится к Number
1 - a; // Первый аргумент приводится к Number
} ///:~
```

Класс Number содержит функцию класса operator+ и объявление friend operator-. Поскольку у класса имеется конструктор с единственным аргументом типа int, тип int может автоматически приводиться к типу Number, но только при соблюдении необходимых условий. Из функции main() видно, что прибавление Number к другому объекту Number работает нормально, так как операция точно соответствует сигнатуре перегруженного оператора. Когда компилятор встречает идентификатор Number, за которым следуют + и int, он находит соответствие с функцией Number::operator+ и преобразует аргумент int в Number при помощи конструктора. Но, встречая int, + и Number, компилятор не знает, что делать, потому что у него есть лишь функция Number::operator+, которая требует, чтобы левосторонний операнд уже был объектом Number. Компилятор выдает сообщение об ошибке.

С функцией friend operator- дело обстоит иначе. Компилятор должен заполнить оба аргумента; левосторонний аргумент не ограничен типом Number. Следовательно, когда компилятор встречает следующее выражение, он может привести первый аргумент к типу Number при помощи конструктора:

```
1 - a
```

Иногда объявление операторных функций в форме функций класса позволяет ограничивать выполняемые операции. Например, при умножении матрицы на вектор справа должен находиться вектор. Но если вы хотите, чтобы оператор мог преобразовать любой из аргументов, оформите его операторную функцию в виде дружественной функции.

К счастью, для выражения 1-1 компилятор не пытается привести оба аргумента к типу Number и вызвать для них функцию operator-. Такое поведение означало бы, что существующий код С вдруг начнет работать не так, как он работал раньше. Компилятор всегда начинает поиск с «простейшей» возможности, а для выражения 1-1 «простейшим» окажется встроенный оператор.

Пример приведения типа

Автоматическое приведение типа чрезвычайно полезно во всех классах, инкапсулирующих символьные строки (в нашем примере в реализации класса использу-

ется стандартный класс C++ string, но это намеренное упрощение). Если бы автоматическое приведение типа не поддерживалось, для вызова любой из существующих функций стандартной библиотеки C++ пришлось бы определять отдельную функцию класса:

```
//: C12:Strings1.cpp
// Автоматическое приведение типа отсутствует
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
string s;
public:
Stringc(const string& str = "") : s(str) {}
int strcmp(const Stringc& S) const {
return ::strcmp(s.c_str(), S.s.c_str());
}
// ... и т. д. для всех функций в string.h
};

int main() {
Stringc s1("hello"), s2("there");
s1.strcmp(s2);
} ///:~
```

В этом примере создается только функция strcmp(), но вам пришлось бы создать соответствующую функцию для каждой используемой функции из файла <cstring>. К счастью, для получения доступа ко всем функциям <cstring> достаточно обеспечить автоматическое приведение типа:

```
//: C12:Strings2.cpp
// С автоматическим приведением типа
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
string s;
public:
Stringc(const string& str = "") : s(str) {}
operator const char*() const {
return s.c_str();
}
};

int main() {
Stringc s1("hello"), s2("there");
strcmp(s1, s2); // Стандартная функция C
strspn(s1, s2); // Любые строковые функции!
} ///:~
```

Теперь любой функции с аргументом char* также можно передать аргумент Stringc, потому что компилятор умеет создавать char* из Stringc.

Ошибки при автоматическом приведении типа

Компилятор сам принимает решение о том, как выполнять приведение типа, поэтому просчеты при проектировании системы преобразований приводят к беде. Простой и очевидный пример дает класс X, который умеет преобразовываться в объект класса Y с помощью операторной функции operatorY(). Если в классе Y определен конструктор с одним аргументом типа X, возможно идентичное приведение типа. Компилятор может перейти от X к Y двумя способами, поэтому при выполнении преобразования он сообщает о возникающей неоднозначности:

```
//: C12:TypeConversionAmbiguity.cpp
class Orange; // Class declaration

class Apple {
public:
operator Orange() const; // Приведение Apple к Orange
};

class Orange {
public:
Orange(Apple); // Приведение Apple к Orange
};

void f(Orange) {}

int main() {
Apple a;
//! f(a); // Ошибка: неоднозначность при приведении типа
} ///:~
```

Самое очевидное решение — не создавайте себе проблем и ограничьтесь одним способом автоматического приведения от одного типа к другому.

Более серьезная проблема возникает в том случае, когда в классе определены операции автоматического приведения к нескольким типам. Иногда это называется *разветвлением* (fan-out):

```
//: C12:TypeConversionFanout.cpp
class Orange {};
class Pear {};

class Apple {
public:
operator Orange() const;
operator Pear() const;
};

// Перегрузка eat():
void eat(Orange);
void eat(Pear);

int main() {
Apple c;
//! eat(c);
// Ошибка: Apple -> Orange или Apple -> Pear ???
} ///:~
```

В классе Apple определены операции автоматического приведения к Orange и Pear. Но возникающая ошибка весьма коварна — проблемы проявляются лишь после

того, как кто-нибудь без всякого злого умысла создаст две перегруженные версии cat() (с одной версией код main() работает нормально).

И снова решение, а также общее правило для операций автоматического приведения типа, состоит в том, чтобы для перехода от одного типа к другому определять только одно автоматическое приведение типа. Причем вы можете определять преобразования для других типов; просто они не должны быть *автоматическими*. Определите для них специальные функции с именами вида makeA() или makeB().

Скрытые операции

С автоматическим приведением типа связано больше скрытых действий, чем можно было бы предполагать. Рассмотрим небольшую головоломку — измененную версию примера CopyingVsInitialization.cpp:

```
//: C12:CopyingVsInitialization2.cpp
class Fi {};

class Fee {
public:
Fee(int) {}
Fee(const Fi&) {}
};

class Fo {
int i;
public:
Fo(int x = 0) : i(x) {}
operator Fee() const { return Fee(i); }
};

int main() {
Fo fo;
Fee fee = fo;
} ///:~
```

Не существует конструктора для создания Fee fee из объекта Fo. Тем не менее в классе Fo определено автоматическое приведение к Fee. Копирующий конструктор для создания Fee на базе Fee тоже отсутствует, но это одна из тех специальных функций, которые могут генерироваться компилятором (такая возможность существует для конструктора по умолчанию, копирующего конструктора, функции operator= и деструктора). Следовательно, для следующей внешне безобидной команды вызывается оператор автоматического приведения типа и генерируется копирующий конструктор:

```
Fee fee = fo;
```

К автоматическому приведению типа следует относиться с осторожностью. Как и в случае любых других перегружаемых операторов, иногда работа программиста заметно упрощается, но злоупотреблять этим обычно не рекомендуется.

Итоги

Перегрузка операторов поддерживается только для одной цели: в некоторых ситуациях она упрощает задачу программиста. В перегрузке нет ничего сверхъестественного; перегруженные операторы напоминают обычные функции с необыч-

ными именами. Эти функции автоматически вызываются компилятором там, где он обнаруживает соответствие шаблону. Но если перегрузка операторов не дает сколько-нибудь заметных преимуществ создателю или пользователю класса, не усложняйте класс ее поддержкой.

Упражнения

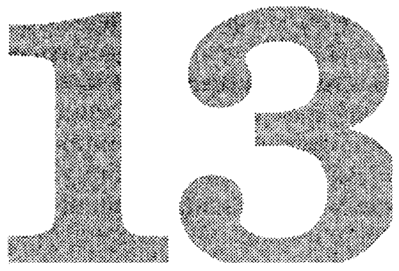
1. Создайте простой класс с перегруженным оператором ++. Попробуйте вызвать этот оператор в префиксной и постфиксной форме и посмотрите, какие предупреждения будут выданы компилятором.
2. Создайте простой класс с переменной int и перегрузите оператор + в форме функции класса. Также включите в класс функцию print(), которая получает аргумент ostream& и выводит данные в заданный поток. Протестируйте класс и продемонстрируйте правильность его работы.
3. Включите в класс из упражнения 2 бинарный оператор - в форме функции класса. Покажите, что объекты класса могут использоваться в сложных выражениях вида a+b-c.
4. Включите в класс из упражнения 2 операторы ++ и -- в префиксной и постфиксной версиях. Убедитесь в том, что постфиксная версия возвращает правильное значение.
5. Измените операторы инкремента и декремента из упражнения 4 так, чтобы префиксные версии возвращали неконстантную ссылку, а постфиксные версии возвращали константный объект. Покажите, что операторы работают правильно, и объясните, почему на практике применяются именно такие возвращаемые значения.
6. Замените функцию print() из упражнения 2 перегруженным оператором <<, как в примере IostreamOperatorOverloading.cpp.
7. Измените упражнение 3 так, чтобы операторы + и - определялись в форме внешних функций. Покажите, что операторы по-прежнему правильно работают.
8. Включите в класс из упражнения 2 унарный оператор - и покажите, что он правильно работает.
9. Создайте класс с переменной private char. Перегрузите операторы ввода-вывода << и >> (как в примере IostreamOperatorOverloading.cpp), протестируйте их. Операторы могут использоваться с объектами fstream, stringstream, а также со стандартными потоками cin и cout.
10. Определите значение фиктивной константы, которая передается вашим компилятором для постфиксных версий операторов ++ и --.
11. Напишите класс Number с переменной double. Включите в него перегруженные операторы +, –, *, /, =. Выберите возвращаемые значения операторных функций так, чтобы выражения могли объединяться в цепочки и повысилась эффективность. Напишите операторную функцию operator double() для автоматического приведения типа.

12. Измените упражнение 11 так, чтобы в нем использовалась *оптимизация возвращаемого значения*, если это не было сделано ранее.
13. Создайте класс, содержащий указатель. Покажите, что функция operator=, сгенерированная компилятором, приводит к появлению двух указателей, ссылающихся на одну область памяти. Исправьте ошибку, определив собственную версию функции operator=. Покажите, что ошибка с дублированием указателей исправлена. Не забудьте о проверке на самоприсваивание и его правильной обработке.
14. Напишите класс Bird, содержащий переменные string и static int. В конструкторе по умолчанию используйте значение int для автоматического построения из имени класса идентификатора с номером, присваиваемого переменной string (Bird#1, Bird#2 и т. д.). Перегрузите оператор << для потоков ostream, чтобы обеспечить вывод объектов Bird. Напишите операторную функцию присваивания operator= и копирующий конструктор. В функции main() убедитесь в том, что все функции работают правильно.
15. Напишите класс BirdHouse, содержащий объект, указатель и ссылку на класс Bird из упражнения 14. Конструктор должен получать в аргументах три объекта Bird. Добавьте оператор << для потоков ostream, чтобы обеспечить вывод объектов BirdHouse. Запретите вызовы функции operator= и копирующего конструктора. В функции main() проверьте правильность работы всех функций. Убедитесь в том, что объекты BirdHouse поддерживают цепочечное присваивание, а также возможность построения выражений с несколькими операторами.
16. Включите переменную int в классы Bird и BirdHouse из упражнения 15. Добавьте операторы +, -, * и /, использующие значение int для выполнения операций с соответствующими членами классов. Проверьте их работу.
17. Повторите упражнение 16 для операторов, реализованных в форме глобальных функций.
18. Добавьте оператор -- в примеры SmartPointer.cpp и NestedSmartPointer.cpp.
19. Измените пример CopyingVsInitialization.cpp так, чтобы все конструкторы выводили сообщение о происходящих событиях. Покажите, что две формы вызова копирующего конструктора (с присваиванием и круглыми скобками) эквивалентны.
20. Попытайтесь определить функцию operator= для класса в глобальной форме. Посмотрите, какое сообщение выдаст компилятор.
21. Создайте класс с оператором присваивания, получающим второй аргумент — объект string со значением по умолчанию "op=call". Создайте функцию, которая присваивает объект класса другому объекту, и покажите, что оператор присваивания работает правильно.
22. В примере CopyingWithPointers.cpp исключите функцию operator= из класса DogHouse. Покажите, что сгенерированный компилятором оператор присваивания правильно копирует string, но дублирует указатель на Dog.
23. В примере ReferenceCounting.cpp добавьте переменные int и static int в оба класса Dog и DogHouse. Все конструкторы обоих классов должны инкремен-

тировать переменную static int и присваивать результат обычной переменной int, отслеживая количество созданных объектов. Внесите необходимые изменения, чтобы во всех командах вывода присутствовали идентификаторы int выводимых объектов.

24. Создайте класс с переменной string, инициализируемой в string в конструкторе. Не включайте в класс ни копирующий конструктор, ни функцию operator=. Создайте второй класс, содержащий внутренний объект первого класса; в этом классе также не должны определяться ни копирующий конструктор, ни функция operator=. Продемонстрируйте, что копирующий конструктор и функция operator= правильно генерируются компилятором.
25. Объедините классы в примерах OverloadingUnaryOperators.cpp и Integer.cpp.
26. В примере PointerToMemberOperator.cpp включите в класс Dog две новые функции, которые вызываются без аргументов и возвращают void. Создайте и протестируйте перегруженный оператор ->*, работающий с обеими новыми функциями.
27. Включите функцию operator->* в пример NestedSmartPointer.cpp.
28. Создайте два класса, Apple и Orange. В классе Apple создайте конструктор, получающий аргумент типа Orange. Создайте функцию с аргументом типа Apple и вызовите ее с аргументом Orange, чтобы продемонстрировать успешное приведение типа. Объявите конструктор Apple с ключевым словом explicit и убедитесь, что автоматическое приведение типа не срабатывает. Измените вызов функции так, чтобы приведение выполнялось явно (и было успешным).
29. Добавьте в пример ReflexivityInOverloading.cpp глобальный оператор * и продемонстрируйте его рефлексивность.
30. Создайте два класса; определите функцию operator+ и функции приведения типа, чтобы сложение было рефлексивным для обоих классов.
31. Исправьте ошибку в примере TypeConversionFanout.cpp, заменив один из операторов автоматического приведения типа специально вызываемой функцией.
32. Напишите простую программу с использованием операторов +, –, * и / для double. Выясните, как ваш компилятор генерирует ассемблерный код, просмотрите его и объясните, как работает внутренняя реализация.

Динамическое создание объектов



Иногда количество, типы и срок жизни объектов в программе точно известны заранее. Но так бывает не всегда.

Сколькими самолетами придется управлять авиадиспетчерской системе? Сколько геометрических тел будет использоваться в системе автоматизированного конструирования? Сколько узлов будет в сети?

Решение проблемы обобщенного программирования требует, чтобы объекты могли создаваться и уничтожаться не на стадии компиляции, а во время выполнения программы. Конечно, для *динамического выделения памяти* в языке С всегда существовали функции malloc() и free(), предназначенные для выделения памяти из *кучи* (динамического пула) во время выполнения.

Тем не менее для C++ эти функции попросту не подходят. При вызове конструктора нельзя передать адрес инициализируемой памяти, и на то есть веские причины. Если бы такая возможность существовала, то вы бы могли:

- забыть это сделать — в результате гарантированная инициализация объектов в C++ перестала бы быть гарантированной;
- случайно выполнить какую-нибудь операцию с объектом до его инициализации, ожидая, что все пройдет нормально;
- передать объект не того размера.

И конечно, даже если бы все было сделано правильно, любые модификации программы снова подвергались бы опасности тех же ошибок. Некорректная инициализация является причиной многих ошибок программирования, поэтому особенно важно гарантировать вызов конструктора для объектов, создаваемых в куче.

Итак, как же в C++ совмещаются гарантии правильной инициализации и зачистки с возможностью динамического создания объектов в куче?

Для этого операции динамического создания объектов вводятся в основной синтаксис языка. Функции malloc() (и ее разновидности) и free() являются биб-

лиотечными функциями, поэтому их работа не может контролироваться компилятором. Но если определить *оператор*, сочетающий динамическое выделение памяти с инициализацией, и еще один оператор, объединяющий зачистку с освобождением памяти, то компилятор по-прежнему мог бы гарантировать вызовы конструкторов и деструкторов для всех объектов.

В этой главе вы узнаете, как операторы C++ new и delete элегантно решают проблему безопасного создания объектов в куче.

Создание объекта

При создании объекта C++ происходят два события.

1. Для объекта выделяется память.
2. Память инициализируется вызовом конструктора.

Вероятно, вы уже поняли, что второе событие происходит *всегда*. В C++ инициализация жестко контролируется, потому что неинициализированные объекты являются основным источником ошибок в программах. Где бы и когда бы ни создавался объект, для него всегда вызывается конструктор.

Однако первое событие может выполняться разными способами или в разное время.

- Память может выделяться перед запуском программы в статической области данных. Эта память существует на протяжении всего жизненного цикла программы.
- Память может выделяться в стеке при достижении определенной точки программы (открывающей фигурной скобки). Выделенная память автоматически освобождается при достижении парной точки (закрывающей фигурной скобки). Операции выделения памяти из стека выполняются на уровне команд процессора и поэтому отличаются очень высокой эффективностью. С другой стороны, чтобы компилятор генерировал правильный код, на момент написания программы должно быть известно требуемое количество переменных.
- Память может выделяться из динамического пула, называемого кучей (динамическое выделение памяти). Память из кучи выделяется вызовом специальной функции во время выполнения программы; следовательно, вы можете в любой момент затребовать дополнительную память в нужном объеме. Кроме того, вы отвечаете за освобождение этой памяти, а это означает, что ее срок жизни может быть сколь угодно долгим — он не зависит от области видимости переменной.

Нередко все три области — статическая память, стек и куча — располагаются в одном непрерывном блоке физической памяти (в порядке, определяемом разработчиком компилятора). Тем не менее никаких правил на этот счет не существует. Стек может находиться в специальной области памяти, а реализация кучи может быть основана на вызове функций памяти операционной системы. Все эти подробности обычно скрываются от программиста, а ему остается думать лишь о том, чтобы при запросе нашлось достаточно свободной памяти.

Динамическое выделение памяти в языке C

Для динамического выделения памяти во время выполнения программы в стандартную библиотеку C включены специальные функции. Функция malloc() и ее разновидности calloc() и realloc() резервируют память в куче, а функция free() возвращает память обратно в кучу. При всей своей практичности эти функции примитивны, а их применение требует хорошего понимания и внимательности от программиста. Например, создание экземпляра класса в куче с применением динамических функций C выглядит примерно так:

```
//: C13:MallocClass.cpp
// Использование malloc() с объектами классов.
// Без "new" нам пришлось бы действовать именно так
#include "../require.h"
#include <cstdlib> // malloc() и free()
#include <cstring> // memset()
#include <iostream>
using namespace std;

class Obj {
  int i, j, k;
  enum { sz = 100 };
  char buf[sz];
public:
  void initialize() { // Нельзя использовать конструктор
    cout << "initializing Obj" << endl;
    i = j = k = 0;
    memset(buf, 0, sz);
  }
  void destroy() const { // Нельзя использовать деструктор
    cout << "destroying Obj" << endl;
  }
};

int main() {
  Obj* obj = (Obj*)malloc(sizeof(Obj));
  require(obj != 0);
  obj->initialize();
  // ... когда-нибудь в будущем:
  obj->destroy();
  free(obj);
} ///:~
```

Выделение памяти для объекта функцией malloc() происходит в строке

```
Obj* obj = (Obj*)malloc(sizeof(Obj));
```

Пользователь должен сам определить размер объекта, что тоже открывает возможности для ошибок. Функция malloc() возвращает void*, поскольку она всего лишь выделяет блок памяти, не создавая объект. C++ не позволяет присвоить void* указателю на любой другой тип, поэтому необходимо явное приведение типа.

Поскольку вызов malloc() может завершиться неудачей из-за нехватки свободной памяти (в этом случае функция возвращает 0), мы должны проверить возвращаемый указатель и убедиться в том, что память успешно выделена.

Но самая серьезная проблема кроется в строке

```
obj->initialize();
```

Если пользователь успешно добрался до этого места, он должен помнить, что объект необходимо инициализировать перед применением. Обратите внимание: конструктор в данном случае не используется, поскольку его нельзя вызвать явно[1], — он автоматически вызывается компилятором при создании объекта. Если пользователь забудет выполнить инициализацию перед обращением к объекту, в программе появляется серьезный источник потенциальных ошибок.

По мнению многих программистов, функции динамического выделения памяти С слишком сложны и запутанны. Не так уж редко встречаются программисты С, использующие системы виртуальной памяти с огромными массивами в статической области только для того, чтобы не приходилось заботиться о динамическом выделении памяти. Поскольку в С++ работа с библиотеками организуется по возможности безопасно и с максимальными удобствами для рядового программиста, подход к выделению и освобождению динамической памяти в стиле С неприемлем.

Оператор new

В С++ было решено объединить все действия, необходимые для создания объекта, в одну операцию, которую выполняет оператор new. При создании объекта с помощью оператора new (так называемого *выражения new*) из кучи выделяется блок памяти, размер которого достаточен для хранения объекта, а затем для этой памяти вызывается конструктор. Например, рассмотрим следующую команду:

```
MyType *fp = new MyType(1.2);
```

При ее выполнении вызывается некий эквивалент функции malloc(sizeof(MyType)), а затем вызывается конструктор MyType, которому передается полученный адрес в виде указателя this и список аргументов (1,2). К тому моменту, когда указатель присваивается fp, он ссылается уже на настоящий инициализированный объект, причем до этого вам не удастся с ним ничего сделать. Объект изначально относится к правильному типу MyType, поэтому приведения типа не требуется.

Перед тем как передавать адрес конструктору, стандартная версия new проверяет, успешно ли была выделена память, поэтому вам не придется явно организовывать такую проверку в программе. Позднее в этой главе вы узнаете, что происходит при нехватке памяти.

Выражения new могут создаваться с использованием любого конструктора, доступного для класса. Если конструктор вызывается без аргументов, то выражение new записывается без списка аргументов конструктора:

```
MyType *fp = new MyType;
```

Обратите внимание, каким простым стал процесс создания объектов из кучи, — единственное выражение решает все вопросы с определением размера объекта, преобразованием и проверками, связанными с безопасностью. Создать объект в куче ничуть не сложнее, чем в стеке.

---

[1] Специальный синтаксис позволяет вызвать конструктор для заранее выделенного блока памяти. Эта возможность будет представлена далее в этой главе.

Оператор delete

У выражений new существуют парные *выражения delete*, в которых сначала вызывается деструктор, а затем освобождается память (довольно часто для этого используется функция free()). По аналогии с тем, как выражение new возвращает указатель на объект, выражение delete должно получать адрес объекта:

```
delete fp;
```

Приведенная команда уничтожает динамически созданный объект MyType и освобождает занимаемую им память.

Оператор delete может вызываться только для объектов, созданных оператором new. Если выделить память под объект функцией malloc() (а также calloc() или realloc()), а затем попытаться освободить ее вызовом delete, последствия будут непредсказуемыми. Так как большинство стандартных реализаций операторов new и delete используют функции malloc() и free(), вероятно, дело кончится освобождением памяти без вызова деструктора.

Если указатель равен нулю, при вызове delete ничего не происходит. По этой причине многие специалисты рекомендуют немедленно обнулить указатель после вызова delete, чтобы предотвратить возможное повторное удаление. Повторное удаление объекта определенно не приведет ни к чему хорошему и вызовет проблемы.

Простой пример

Следующий пример доказывает, что при вызове оператора new выполняется инициализация:

```
//: C13:Tree.h
#ifndef TREE_H
#define TREE_H
#include <iostream>

class Tree {
  int height;
public:
  Tree(int treeHeight) : height(treeHeight) {}
  ~Tree() { std::cout << "*"; }
  friend std::ostream&
  operator<<(std::ostream& os, const Tree* t) {
    return os << "Tree height is: "
              << t->height << std::endl;
  }
};
#endif // TREE_H ///:~

//: C13:NewAndDelete.cpp
// Простая демонстрация new и delete
#include "Tree.h"
using namespace std;

int main() {
  Tree* t = new Tree(40);
  cout << t;
  delete t;
} ///:~
```

Чтобы доказать, что для созданного объекта действительно вызывался конструктор, мы выводим значение Tree. Для этого оператор << перегружается так, чтобы он работал с ostream и Tree*. Обратите внимание на интересную подробность: хотя функция объявлена дружественной (friend), она определяется как подставляемая! Это всего лишь вопрос удобства — определение дружественной функции как подставляемой в классе не изменяет ни ее статуса дружественной, ни того факта, что функция является глобальной и не принадлежит классу. Также обратите внимание на то, что команда return возвращает результат всего выражения, то есть ostream& (как и должно быть в соответствии с типом возвращаемого значения функции).

Затраты на управление памятью

При создании автоматических объектов в стеке размеры объектов и время их существования жестко фиксируются сгенерированным кодом, поскольку компилятору известен точный тип, количество и видимость объектов. Создание объектов в куче требует дополнительных издержек как по времени, так и по занимаемой памяти. Ниже описан типичный сценарий (вызов malloc() можно заменить вызовом calloc() или realloc()).

1. Вызывается функция malloc(), которая запрашивает блок памяти из кучи (соответствующий код может входить в реализацию malloc()).
2. В куче ищется блок памяти, размер которого достаточен для удовлетворения запроса. Задача решается поиском по карте памяти или по каталогу с информацией о том, какие блоки используются в настоящий момент, а какие остаются свободными. Поиск выполняется быстро, но может потребовать нескольких попыток, поэтому затраты времени непредсказуемы. Другими словами, невозможно рассчитывать на то, что время выполнения функции malloc() всегда будет одинаковым.
3. Перед возвращением указателя на выделенный блок необходимо зарегистрировать его размер и начальный адрес, чтобы блок не использовался при последующих вызовах malloc(), а при вызове free() система знала, сколько памяти необходимо освободить.

Конкретные реализации могут изменяться в весьма широких пределах. Например, ничто не мешает реализовать примитивы выделения памяти на уровне процессорных команд. Если вам действительно интересно, попробуйте написать несколько тестовых программ и выяснить, как в вашей системе реализована функция malloc(). Также почитайте исходные тексты библиотеки, если они доступны (впрочем, для GNU C они доступны всегда).

Переработка ранних примеров

Давайте перепишем пример класса Stash, встречавшийся в начале книги, с использованием операторов new и delete. Анализ нового кода даст полезный общий обзор по всем рассматриваемым темам.

В новой версии классы Stash и Stack не «владеют» объектами, на которые они ссылаются. Иначе говоря, выход объекта Stash или Stack из видимости не приводит

к вызову delete для тех объектов, которые в нем хранятся. Дело в том, что для обеспечения максимальной универсальности в контейнерах хранятся указатели на void. Вызов delete для указателя на void приводит только к освобождению памяти, поскольку из-за отсутствия информации о типе компилятор не может определить, какой деструктор нужно вызвать.

Вызов delete void* как вероятный источник ошибки

Стоит особо заметить, что вызов delete void* почти наверняка станет источником ошибки в программе, если только указатель не ссылается на очень простой объект, у которого нет деструктора. Следующий пример показывает, что при этом происходит:

```
//: C13:BadVoidPointerDeletion.cpp
// Вызов delete для указателей на void может привести к утечке памяти
#include <iostream>
using namespace std;

class Object {
  void* data; // Некий блок памяти
  const int size;
  const char id;
public:
  Object(int sz, char c) : size(sz), id(c) {
    data = new char[size];
    cout << "Constructing object " << id
         << ", size = " << size << endl;
  }
  ~Object() {
    cout << "Destructing object " << id << endl;
    delete []data; // OK, просто освобождаем память,
    // вызов деструктора не нужен.
  }
};

int main() {
  Object* a = new Object(40, 'a');
  delete a;
  void* b = new Object(40, 'b');
  delete b;
} ///:~
```

Класс Object содержит указатель void*, который инициализируется необработанными данными (то есть не указывает на объект, у которого имеется деструктор). В деструкторе Object оператор delete вызывается для void* без всяких нежелательных эффектов, поскольку нам нужно лишь освободить выделенную память.

Однако функция main() показывает, что на самом деле очень важно, чтобы оператор delete знал тип освобождаемого объекта. Результат выполнения программы выглядит так:

```
Constructing object a, size = 40
Destructing object a
Constructing object b, size = 40
```

Поскольку в случае команды delete a компилятор знает, что a указывает на Object, деструктор благополучно вызывается, а память, выделенная для data, освобожда-

ется. Но если работа с объектом производится через void*, как в случае команды delete b, освобождается лишь память Object, но без вызова деструктора, поэтому блок памяти, на который ссылается data, освобожден не будет. Вероятно, при компиляции программы даже не появятся предупреждения; компилятор полагает, что вам известно, что вы делаете. Так возникает незаметная утечка памяти.

Если в программе возникает утечка памяти, просмотрите вызовы delete и проверьте тип удаляемого указателя. Если он относится к типу void*, то вполне вероятно, что вы обнаружили один из источников (впрочем, C++ предоставляет массу других возможностей для утечки памяти).

Зачистка при наличии указателей

Чтобы контейнеры Stash и Stack были максимально гибкими (то есть могли хранить объекты любого типа), в них должны храниться указатели на void. Это означает, что указатель, полученный от объекта Stash или Stack, перед использованием необходимо привести к правильному типу; как показано выше, перед удалением указатель тоже должен быть приведен к правильному типу, или в программе возникнет утечка памяти.

Другая потенциальная проблема с утечкой памяти заключается в том, чтобы обеспечить вызов delete для всех указателей на объекты, хранящихся в контейнере. Указатель не «принадлежит» контейнеру, потому что контейнер хранит его в виде типа void* и не сможет правильно выполнить зачистку. Ответственность за нее должна быть возложена на пользователя. Это приводит к серьезным проблемам, если в одном контейнере хранятся указатели на объекты, созданные в стеке, *и* указатели на объекты, созданные в куче. Дело в том, что вызов delete для указателей на память, которая не была выделена из кучи, небезопасен. (С другой стороны, когда вы получаете указатель от контейнера, как узнать, откуда была выделена память?) Таким образом, вы должны быть уверены, что в следующих версиях Stash и Stack хранятся указатели только на объекты, созданные в куче; задача решается либо особо тщательным программированием, либо созданием классов, объекты которых могут создаваться только в куче.

Также очень важно проследить за тем, чтобы прикладной программист освободил все указатели в контейнере. В предшествующих примерах класс Stack проверял в своем деструкторе то, что все объекты Link были извлечены из стека. Впрочем, для указателей Stash необходим другой подход.

Класс Stash с указателями

В новой версии класса Stash, называемой PStash, хранятся *указатели* на объекты, созданные в куче (тогда как старая версия Stash копировала объекты по значению в контейнер Stash). Использование операторов new и delete позволяет легко и надежно организовать хранение указателей на объекты, созданные в куче.

Заголовочный файл для класса Stash с указателями выглядит так:

```
//: C13:PStash.h
// Вместо объектов в контейнере хранятся указатели
#ifndef PSTASH_H
#define PSTASH_H

class PStash {
```

```
  int quantity; // Количество элементов
  int next; // Следующий пустой элемент
   // Память для хранения указателей:
  void** storage;
  void inflate(int increase);
public:
  PStash() : quantity(0), storage(0), next(0) {}
  ~PStash();
  int add(void* element);
  void* operator[](int index) const; // Выборка
  // Удаление ссылки из PStash:
  void* remove(int index);
  // Количество элементов в Stash:
  int count() const { return next; }
};
#endif // PSTASH_H ///:~
```

Теперь storage представляет собой массив указателей на void, а память для этого массива выделяется оператором new вместо функции malloc(). В следующей команде типом создаваемого объекта является void*, поэтому она создает массив указателей на void:

```
void** st = new void*[quantity + increase];
```

Деструктор освобождает память, в которой хранились указатели на void, не пытаясь освободить память, на которую эти указатели ссылаются (что, как отмечалось ранее, привело бы к освобождению памяти без вызова деструктора, поскольку указатель на void не содержит информации о типе).

Другое изменение — замена функции fetch() оператором индексирования [ ], более логичная с точки зрения синтаксиса. Однако оператор тоже возвращает указатель void*, поэтому пользователь должен помнить, какие типы хранятся в контейнере, и преобразовывать указатели при выборке (в будущем мы исправим этот недостаток).

Определения функций класса выглядят так:

```
//: C13:PStash.cpp {O}
// Определения функций контейнера Stash для указателей
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <cstring> // Функции 'mem'
using namespace std;

int PStash::add(void* element) {
  const int inflateSize = 10;
  if(next >= quantity)
    inflate(inflateSize);
  storage[next++] = element;
  return(next - 1); // Индекс
}

// Объекты, на которые ссылаются указатели, не принадлежат контейнеру:
PStash::~PStash() {
  for(int i = 0; i < next; i++)
    require(storage[i] == 0,
      "PStash not cleaned up");
  delete []storage;
}
```

```
// Перегрузка оператора для выборки
void* PStash::operator[](int index) const {
  require(index >= 0.
    "PStash::operator[] index negative");
  if(index >= next)
    return 0; // Признак конца
  // Получение указателя на запрашиваемый элемент:
  return storage[index];
}

void* PStash::remove(int index) {
  void* v = operator[](index);
  // "Удаление" указателя:
  if(v != 0) storage[index] = 0;
  return v;
}

void PStash::inflate(int increase) {
  const int psz = sizeof(void*);
  void** st = new void*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Освобождение старого буфера
  storage = st; // Перевод указателя на новый буфер
} ///:~
```

Функция add() практически не изменилась, если не считать того, что в контейнере теперь сохраняется указатель вместо копии всего объекта.

Код inflate() был изменен: теперь в этой функции выделяется массив void* (тогда как предыдущая архитектура работала с простым байтовым буфером). Вместо прежнего решения с копированием, основанным на индексировании массива, мы сначала заполняем нулями всю новую память при помощи функции стандартной библиотеки C memset() (вообще говоря, обнуление памяти не является строго необходимым, поскольку класс PStash и так должен правильно управлять памятью, однако лишние меры предосторожности не повредят). Функция memcpy() копирует существующие данные из одного блока памяти в другой. Такие функции, как memset() и memcpy(), обычно оптимизируются по времени, поэтому они могут работать быстрее циклов из предыдущей версии. Скорее всего, функции типа inflate() вызываются не так часто, чтобы можно было заметить разницу в быстродействии. С другой стороны, вызовы функций более наглядны по сравнению с циклами, и это помогает предотвратить ошибки программирования.

Чтобы ответственность за зачистку объектов была полностью переложена на прикладного программиста, в PStash предусмотрены два способа обращения к указателям: оператор [ ] просто возвращает указатель, оставляя его храниться в контейнере, а функция класса remove() возвращает указатель и удаляет его из контейнера, обнуляя соответствующую позицию. Деструктор PStash убеждается в том, что из контейнера были удалены все указатели на объекты; в противном случае он выводит предупреждающее сообщение, чтобы программист исправил утечку памяти (более элегантные решения будут представлены в следующих главах).

Тестовая программа

Ниже приведена наша старая тестовая программа для класса Stash, переписанная для PStash:

```
//: C13:PStashTest.cpp
//{L} PStash
// Тестирование контейнера Stash для указателей
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  PStash intStash;
  // 'new' также работает со встроенными типами. Обратите внимание
  // на синтаксис "псевдоконструктора":
  for(int i = 0; i < 25; i++)
    intStash.add(new int(i));
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash[" << j << "] = "
         << *(int*)intStash[j] << endl;
  // Зачистка:
  for(int k = 0; k < intStash.count(); k++)
    delete intStash.remove(k);
  ifstream in ("PStashTest.cpp");
  assure(in, "PStashTest.cpp");
  PStash stringStash;
  string line;
  while(getline(in, line))
    stringStash.add(new string(line));
  // Вывод строк:
  for(int u = 0; stringStash[u]; u++)
    cout << "stringStash[" << u << "] = "
         << *(string*)stringStash[u] << endl;
  // Зачистка:
  for(int v = 0; v < stringStash.count(); v++)
    delete (string*)stringStash.remove(v);
} ///:~
```

Как и прежде, контейнеры создаются и заполняются данными, но на этот раз в контейнере хранятся указатели, полученные от выражений new. В первом тесте обратите внимание на строку:

```
intStash.add(new int(i));
```

В выражении new int(i) используется синтаксис псевдоконструктора. В куче выделяется память для нового объекта int, который инициализируется значением i.

В процессе вывода значение, возвращаемое функцией PStash::operator[], необходимо привести к нужному типу; то же самое делается для всех объектов PStash в программе. Этот нежелательный эффект обусловлен применением указателей void* в базовой реализации и будет исправлен в следующих главах.

Во втором тесте мы открываем файл с исходным текстом программы и читаем его по строкам в другой объект PStash. Строки читаются в string функций getline(), после чего команда new string создает новую независимую копию текущей строки line. Если бы при вызове каждый раз передавался адрес line, то мы бы получили группу указателей на переменную line с последней строкой, прочитанной из файла.

При выборке указателей используется выражение

```
*(string*)stringStash[v]
```

Указатель, возвращаемый оператором [ ], должен быть приведен к правильному типу string*. Затем указатель string* разыменовывается; компилятор видит объект string и передает его в cout.

Объекты, созданные в куче, должны быть уничтожены вызовами remove(). В противном случае во время выполнения программы будет выведено сообщение о том, что не были удалены все объекты PStash. Обратите внимание: в случае с указателями на int приведения типа не требуется, так как для int достаточно простого освобождения памяти без вызова деструктора:

```
delete intStash.remove(k);
```

Но если забыть о приведении типа для указателей на string, в программе возникнет утечка памяти, поэтому приведение абсолютно необходимо:

```
delete (string*)stringStash.remove(k);
```

Некоторые (хотя и не все) из перечисленных проблем решаются применением шаблонов, о которых вы узнаете в главе 16.

Операторы new и delete для массивов

C++ позволяет одинаково легко создавать массивы объектов как в стеке, так и в куче, естественно, с вызовом конструктора для каждого объекта в массиве. Тем не менее действует одно ограничение: в классе должен быть определен конструктор по умолчанию (кроме инициализации агрегатов, созданных в стеке, — см. главу 6), поскольку для каждого объекта вызывается конструктор по умолчанию.

При создании массива объектов в куче оператором new необходимо сделать еще кое-что. Рассмотрим пример такого массива:

```
MyType* fp = new MyType[100];
```

Команда выделяет в куче память для 100 объектов MyType и вызывает конструктор для каждого объекта. Но после ее выполнения вы получаете простой указатель MyType*, который ничем не отличается от указателя, полученного при создании одиночного объекта командой

```
MyType* fp2 = new MyType;
```

Вам, как автору программы, известно, что указатель fp на самом деле определяет начальный адрес массива, поэтому выборка элементов массива выражениями вида fp[3] имеет смысл. Но что произойдет при уничтожении массива? Следующие команды выглядят абсолютно одинаково, и последствия их выполнения тоже будут одинаковыми:

```
delete fp2;  // OK
delete fp;   // Совсем не то
```

Для объекта MyType по указанному адресу вызывается деструктор, после чего занимаемая им память освобождается. Для fp2 это вполне нормально, но для fp остальные 99 вызовов конструктора не выполняются. Впрочем, освобожденная память будет иметь правильный объем, потому что она выделяется одним большим блоком, размер которого сохраняется где-то в служебных данных функцией выделения.

Чтобы решить эту проблему, следует передать компилятору информацию о том, что указатель в действительности ссылается на начальный адрес массива. Синтаксис команды выглядит так:

```
delete []fp:
```

Пустые квадратные скобки приказывают компилятору сгенерировать код для выборки количества элементов массива, сохраненного при создании массива, и вызвать деструктор для соответствующего количества объектов. Этот синтаксис представляет собой усовершенствованную версию более раннего синтаксиса, иногда встречающегося в старых программах:

```
delete [100]fp:
```

В старом варианте синтаксиса программисту приходилось задавать количество объектов в массиве; возникала опасность того, что оно будет задано неверно. Дополнительные затраты на получение количества элементов очень низки, поэтому было решено задавать это количество только в одном месте, а не в двух.

Указатели и массивы

Определенный выше указатель fp можно перевести на любой другой адрес, что не имеет смысла для начального адреса массива. Логичнее объявить его константным, чтобы любые попытки модификации указателя помечались как ошибочные. Оба следующих определения неправильны:

```
int const* q = new int[10]:
const int* q = new int[10]:
```

В обоих случаях ключевое слово const связывается с int; иначе говоря, константность относится к значению, на которое ссылается указатель, а не к самому указателю. Вместо этого следует использовать запись

```
int* const q = new int[10]:
```

Элементы массива q можно изменять, но любые попытки модификации q (например, q++) недопустимы, как и для обычного идентификатора массива.

Нехватка памяти

Что произойдет, если функция operator new() не сможет найти блок памяти, объем которого достаточен для хранения объекта? В этом случае вызывается специальная функция, называемая *обработчиком new*. А вернее, программа проверяет указатель на функцию и, если он отличен от нуля, вызывает функцию, на которую ссылается указатель.

По умолчанию обработчик new *генерирует исключение* (эта тема рассматривается во втором томе книги). Тем не менее, если в программе выделяется память из кучи, рекомендуется, как минимум, заменить стандартный обработчик new другим, который выводит сообщение о нехватке памяти и завершает программу. В этом случае вы хотя бы будете знать, что происходит в процессе отладки. В окончательной версии стоит предусмотреть более мощную схему восстановления и продолжения работы.

Чтобы заменить обработчик new, включите заголовочный файл new.h и вызовите функцию set_new_handler() с адресом функции, которую требуется установить:

```
//: C13:NewHandler.cpp
// Изменение обработчика new
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std:

int count = 0;

void out_of_memory() {
  cerr << "memory exhausted after " << count
    << " allocations!" << endl:
  exit(1):
}

int main() {
  set_new_handler(out_of_memory):
  while(1) {
    count++:
    new int[1000]: // Расходует всю память
  }
} ///:~
```

Функция, назначаемая обработчиком new, вызывается без аргументов и возвращает void. Цикл while продолжает создавать объекты int (и забывать их адреса) до тех пор, пока не будет полностью исчерпана вся свободная память. При следующем вызове next выделить память не удастся, и тогда будет вызван обработчик new.

Поведение обработчика new жестко связывается с функцией operator new(). Если перегрузить operator new() (эта возможность рассматривается в следующем разделе), то по умолчанию обработчик new не вызывается. Если вы хотите, чтобы обработчик new вызывался и после перегрузки, включите соответствующий код в перегруженную версию operator new().

Конечно, вы можете написать более совершенный обработчик new и даже попытаться освободить занимаемую память (эта процедура обычно называется *уборкой мусора*). Впрочем, эта задача не для новичков.

Перегрузка операторов new и delete

При выполнении выражения new происходят две вещи: сначала функция operator new() выделяет память, а затем вызывается конструктор. В выражении delete вызывается деструктор, после чего память освобождается вызовом функции operator delete(). Вызовы конструктора и деструктора программисту неподвластны (иначе он мог бы случайно помешать их вызову), однако вы *можете* изменить функции выделения памяти operator new() и operator delete().

Система выделения памяти, используемая операторами new и delete, разработана для решения общих задач. В нестандартных ситуациях ее возможностей может оказаться недостаточно. Самая распространенная причина для изменения распределителей памяти — эффективность; иногда программе приходится создавать и уничтожать столько объектов какого-нибудь класса, что эти операции становятся «узким местом» в производительности системы. Для решения подобных проблем C++ позволяет перегрузить операторы new и delete и реализовать собственную схему выделения памяти.

Другой фактор — фрагментация кучи. Выделение и освобождение блоков разного размера может привести к такому измельчению блоков памяти в куче, что фактически приведет к нехватке памяти. Другими словами, свободная память может быть доступна, но из-за фрагментации не найдется ни одного блока, размер которого окажется достаточным для удовлетворения запроса. Определяя собственный распределитель памяти для класса, вы предотвращаете такую возможность.

Во встроенных системах и в системах реального времени программам приходится работать очень долго при ограниченных ресурсах. Кроме того, эти системы иногда требуют, чтобы время выделения памяти не менялось, а исчерпание или фрагментация кучи была недопустима. Задача решается при помощи пользовательских распределителей памяти; без них программисты были бы вынуждены обходиться без операторов new и delete, лишившись весьма полезной возможности С++.

При перегрузке функций operator new() и operator delete() необходимо помнить, что изменяется только *низкоуровневый механизм распределения памяти*. Компилятор просто вызывает вашу версию new вместо стандартной версии, а затем вызывает конструктор для выделенной памяти. Следовательно, хотя компилятор выделяет память *и* вызывает конструктор, перегрузка позволяет изменить лишь фазу выделения памяти (аналогичное ограничение действует и в отношении оператора delete).

Перегрузка функции operator new() также изменяет поведение системы при нехватке памяти, поэтому вы должны решить, что в этом случае должна делать эта функция: возвращать ноль, в цикле вызывать обработчик new и повторять попытку выделения памяти или (типичное поведение) генерировать исключение bad_alloc (исключения будут рассматриваться во втором томе книги).

Перегрузка операторов new и delete происходит по тем же правилам, что и перегрузка любых других операторов. Впрочем, вы можете выбрать между перегрузкой глобального распределителя памяти или назначением специального распределителя памяти конкретному классу.

Перегрузка глобальных операторов

Самый радикальный вариант применяется в том случае, когда глобальные версии операторов new и delete не подходят в масштабах всей системы. При такой глобальной перегрузке стандартные версии становятся полностью недоступными — они даже не могут вызываться из перегруженных версий.

Перегруженная версия оператора new должна получать аргумент size_t (стандартный тип С для определения размеров). Этот аргумент генерируется и передается компилятором; он определяет размер объекта, для которого необходимо выделить память. Оператор должен вернуть указатель на объект заданного размера (или большего, если на то есть какие-то причины) или ноль, если память не найдена (в этом случае конструктор *не* вызывается!). Впрочем, при отсутствии свободной памяти лучше не ограничиваться простым возвратом нуля и сделать что-нибудь более содержательное, например вызвать обработчик new или сгенерировать исключение, чтобы сообщить о возникшей проблеме.

Возвращаемое значение функции operator new() относится к типу void* и *не является* указателем на какой-либо конкретный тип. Функция всего лишь резервирует память и не создает законченного объекта; это происходит лишь после вызова конструктора — события, гарантированного компилятором и неподвластного вам.

Функция operator delete() получает указатель void* на память, выделенную функцией operator new(). Использование указателя void* объясняется тем, что operator delete() получает указатель уже *после* вызова деструктора, который лишает блок памяти всяких признаков «объектности». Возвращаемое значение относится к типу void.

Следующий простой пример показывает, как перегружаются глобальные версии new и delete:

```
//: C13:GlobalOperatorNew.cpp
// Перегрузка глобальных операторов new и delete
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
  printf("operator new: %d Bytes\n", sz);
  void* m = malloc(sz);
  if(!m) puts("out of memory");
  return m;
}

void operator delete(void* m) {
  puts("operator delete");
  free(m);
}

class S {
  int i[100];
public:
  S() { puts("S::S()"); }
  ~S() { puts("S::~S()"); }
};

int main() {
  puts("creating & destroying an int");
  int* p = new int(47);
  delete p;
  puts("creating & destroying an s");
  S* s = new S;
  delete s;
  puts("creating & destroying S[3]");
  S* sa = new S[3];
  delete []sa;
} ///:~
```

В этом примере продемонстрирована общая форма перегрузки операторов new и delete. Перегруженные версии используют стандартные библиотечные функции C malloc() и free() (которые, вероятно, используются и в стандартных версиях new и delete!). Наряду с выделением памяти они также выводят сообщения о выполняемых действиях. Обратите внимание на наличие функций printf() и puts() вместо потоков ввода-вывода. Дело в том, что при создании объектов iostream (включая глобальные cin, cout и cerr) вызывается оператор new для выделения памяти. Вызов printf() предотвращает взаимную блокировку, поскольку эта функция не требует для инициализации оператора new.

Создание объектов встроенных типов в main() доказывает, что перегруженные версии new и delete вызываются и в этом случае. Затем создается одиночный объект

типа S, а за ним — массив объектов S. По размеру выделяемого для массива блока видно, что в массиве появляется дополнительная память для хранения информации о количестве содержащихся в нем объектов. Во всех случаях используются глобальные перегруженные версии операторов new и delete.

Перегрузка операторов для класса

Хотя явно использовать ключевое слово static не обязательно, при перегрузке операторов new и delete для класса создаются статические функции класса. Как и прежде, синтаксис не отличается от синтаксиса перегрузки других операторов. Когда компилятор встречает вызов new, предназначенный для создания объекта вашего класса, он отдает предпочтение функции класса operator new() перед глобальной версией. Для остальных типов объектов используются глобальные версии new и delete (если, конечно, для них не были определены собственные версии распределителей).

В следующем примере организуется примитивная система выделения памяти для класса Framis. При запуске программы в статической области данных резервируется блок памяти, который в дальнейшем задействуется для выделения памяти под объекты Framis. Чтобы определить, какие блоки были выделены, а какие остаются свободными, мы используем простой байтовый массив (один байт для каждого блока):

```
//: C13:Framis.cpp
// Локальная перегрузка операторов new и delete
#include <cstddef> // Size_t
#include <fstream>
#include <iostream>
#include <new>
using namespace std;
ofstream out("Framis.out");

class Framis {
  enum { sz = 10 };
  char c[sz]; // Простой заполнитель памяти (не используется)
  static unsigned char pool[];
  static bool alloc_map[];
public:
  enum { psize = 100 };  // Максимальное количество объектов
  Framis() { out << "Framis()\n"; }
  ~Framis() { out << "~Framis() ... "; }
  void* operator new(size_t) throw(bad_alloc);
  void operator delete(void*);
};
unsigned char Framis::pool[psize * sizeof(Framis)];
bool Framis::alloc_map[psize] = {false};

// Размер игнорируется -- предполагаем объект Framis
void*
Framis::operator new(size_t) throw(bad_alloc) {
  for(int i = 0; i < psize; i++)
    if(!alloc_map[i]) {
      out << "using block " << i << " ... ";
      alloc_map[i] = true; // Пометить как используемый
      return pool + (i * sizeof(Framis));
    }
```

```
    out << "out of memory" << endl;
    throw bad_alloc();
  }

  void Framis::operator delete(void* m) {
    if(!m) return; // Проверка null-указателя
    // Предполагается, что объект был создан в пуле.
    // Вычисляем номер блока:
    unsigned long block = (unsigned long)m
      - (unsigned long)pool;
    block /= sizeof(Framis);
    out << "freeing block " << block << endl;
    // Mark it free:
    alloc_map[block] = false;
  }

  int main() {
    Framis* f[Framis::psize];
    try {
      for(int i = 0; i < Framis::psize; i++)
        f[i] = new Framis;
      new Framis; // Нехватка памяти
    } catch(bad_alloc) {
      cerr << "Out of memory!" << endl;
    }
    delete f[10];
    f[10] = 0;
    // Использование освобожденной памяти:
    Framis* x = new Framis;
    delete x;
    for(int j = 0; j < Framis::psize; j++)
      delete f[j]; // Освобождаем f[10] OK
  } ///:~
```

Пул памяти создается выделением массива байтов, размер которого достаточен для хранения psize объектов Framis. Карта свободных блоков состоит из psize элементов, по одному bool для каждого блока. Все элементы в карте свободных блоков инициализируются значением false, при этом используется трюк агрегатной инициализации: при задании значения первого элемента компилятор автоматически инициализирует все остальные элементы своими значениями по умолчанию (false в случае bool).

По своему синтаксису локальная версия функции operator new() не отличается от глобальной версии. Она ищет по карте свободных блоков элемент со значением false, присваивает ему значение true (признак выделенного блока) и возвращает адрес соответствующего блока памяти. Если найти память не удалось, функция выводит сообщение в трассировочный файл и генерирует исключение bad_alloc.

Это первый пример использования исключений в этой книге. Поскольку исключения будут подробно рассмотрены лишь во втором томе, мы ограничиваемся лишь их кратким описанием. В функции operator new() имеются два места, связанные с обработкой исключений. Во-первых, за списком аргументов функции следует секция throw(bad_alloc), которая сообщает компилятору и читателю программы, что функция может генерировать исключение типа bad_alloc. Во-вторых, при отсутствии свободной памяти функция генерирует исключение командой throw bad_alloc. После выдачи исключения функция перестает выполняться, а управление передается *обработчику исключения*, представленному секцией catch.

В функции main() встречается вторая часть механизма обработки исключений — блок try-catch. Секция try заключается в фигурные скобки и содержит весь код, при выполнении которого могут возникнуть исключения; в данном случае это все вызовы new для объектов Framis. Сразу же за секцией try следуют одна или несколько секций catch, в каждой из которых задается тип перехватываемого исключения. В нашем примере условие catch(bad_alloc) сообщает, что эта секция перехватывает исключение bad_alloc. Эта конкретная секция catch выполняется только в том случае, если программа генерирует исключение bad_alloc, а выполнение программы продолжается за последней секцией catch в группе (в нашем случае определена только одна секция catch, но их может быть больше).

На этот раз мы можем использовать потоки ввода-вывода, потому что глобальные версии функций operator new() и operator delete() остались без изменений.

Функция operator delete() предполагает, что адрес Framis был взят из пула. Такое предположение оправданно, поскольку локальная версия operator new() будет вызываться при создании в куче отдельного объекта Framis, а не массива объектов: для массивов требуется глобальная версия new. Таким образом, пользователь может случайно вызвать функцию operator delete() без пустых квадратных скобок — признака уничтожения массива. Это приведет к возникновению проблем. Кроме того, пользователь может попытаться удалить указатель на объект, созданный в стеке. Если вы считаете, что в программе может произойти нечто подобное, включите в программу проверку того, что адрес находится в границах пула и выровнен по правильной границе (заодно это дает некоторое представление о возможном применении перегруженных версий new и delete для обнаружения утечки памяти).

Функция operator delete() вычисляет, какой блок пула представлен данным указателем, и сбрасывает флаг занятости этого блока, тем самым показывая, что блок был освобожден.

В функции main() объекты Framis динамически создаются в количестве, достаточном для исчерпания всей свободной памяти; тем самым проверяется поведение программы при недостатке памяти. Затем программа освобождает один из объектов и создает другой объект, показывая, что освобожденная память может снова использоваться в системе.

Данная схема выделения памяти работает только с объектами Framis, поэтому она, вероятно, гораздо быстрее универсальной схемы выделения памяти, используемой стандартными версиями new и delete. Однако следует учитывать, что ее работоспособность может быть нарушена из-за наследования (эта тема рассматривается в главе 14).

Перегрузка операторов для массивов

Если перегрузить операторы new и delete для класса, то перегруженные операторы будут вызываться при каждом создании объекта этого класса. Но при создании *массива* объектов класса вызывается глобальная функция operator new(), которая выделяет блок памяти сразу для всего массива, а затем глобальная функция operator delete() эту память освобождает. Чтобы управлять выделением памяти для массивов объектов, следует перегрузить для класса специальные версии функций — функции operator new[] и operator delete[]. Следующий пример показывает, что происходит при вызове двух разных версий:

```
//: C13:ArrayOperatorNew.cpp
// Операторы new и delete для массивов
#include <new> // Определение size_t
#include <fstream>
using namespace std;
ofstream trace("ArrayOperatorNew.out");

class Widget {
  enum { sz = 10 };
  int i[sz];
public:
  Widget() { trace << "*"; }
  ~Widget() { trace << "~"; }
  void* operator new(size_t sz) {
    trace << "Widget::new: "
         << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete(void* p) {
    trace << "Widget::delete" << endl;
    ::delete []p;
  }
  void* operator new[](size_t sz) {
    trace << "Widget::new[]: "
         << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    trace << "Widget::delete[]" << endl;
    ::delete []p;
  }
};

int main() {
  trace << "new Widget" << endl;
  Widget* w = new Widget;
  trace << "\ndelete Widget" << endl;
  delete w;
  trace << "\nnew Widget[25]" << endl;
  Widget* wa = new Widget[25];
  trace << "\ndelete []Widget" << endl;
  delete []wa;
} ///:~
```

В данном случае вызываются глобальные версии операторов new и delete, поэтому все происходит так, как если бы операторы не перегружались (разве что выводятся дополнительные трассировочные сообщения). Очевидно, что в перегруженных версиях операторов new и delete может использоваться любая схема распределения памяти по желанию программиста.

Как видите, синтаксис операторов new и delete для массивов не отличается от синтаксиса для отдельных объектов, если не считать появления квадратных скобок. В обоих случаях передается размер выделяемого блока памяти, только для массива это размер не отдельного объекта, а всего массива. Помните, что перегруженная функция operator new() *обязана* сделать только одно — вернуть указатель на достаточно большой блок памяти. Хотя эта память может инициализироваться, обычно инициализацию выполняет конструктор, автоматически вызываемый компилятором для выделенной памяти.

Конструктор и деструктор класса Widget просто выводят символы, чтобы вы знали об их вызове. Вот как выглядит трассировочный файл для одного из компиляторов:

```
new Widget
Widget::new: 40 bytes
*
delete Widget
~Widget::delete

new Widget[25]
Widget::new[]: 1004 bytes
*************************
delete [] Widget
~~~~~~~~~~~~~~~~~~~~~~~~~Widget::delete[]
```

Как и следовало ожидать, для создания отдельного объекта требуется 40 байт (на этом компьютере int представляется четырьмя байтами). Затем вызываются функция operator new() и конструктор (на это указывает оператор *). И наоборот, при вызове delete сначала вызывается деструктор, а затем функция operator delete().

При создании массива объектов Widget, как и предполагалось, используется версия operator new() для массива. Но обратите внимание: размер запрашиваемого блока на четыре байта больше предполагаемого. В дополнительных четырех байтах система хранит информацию о массиве, а именно о количестве объектов в нем. Таким образом, при выполнении следующей команды квадратные скобки сообщают компилятору, что речь идет о массиве объектов:

```
delete []Widget;
```

Поэтому компилятор генерирует код для определения количества элементов в массиве и вызывает деструктор соответствующее количество раз. Из результатов видно, что хотя функции operator new() и operator delete() для всего блока вызываются только один раз, стандартные конструктор и деструктор вызываются для каждого объекта в массиве.

Вызовы конструктора

Рассмотрим следующую команду:

```
MyType* f = new MyType;
```

Учитывая, что эта команда вызывает оператор new для выделения блока памяти размером MyType, а затем вызывает для этого конструктор MyType, что произойдет, если выделение памяти оператором new завершится неудачей? Конструктор в этом случае не вызывается, поэтому, хотя объект все равно не создается, по крайней мере предотвращается вызов конструктора с нулевым указателем this. Следующий пример доказывает это:

```
//: C13:NoMemory.cpp
// При неудачном вызове new конструктор не вызывается
#include <iostream>
#include <new> // Определение bad_alloc
using namespace std;

class NoMemory {
public:
  NoMemory() {
```

```
    cout << "NoMemory::NoMemory()" << endl;
  }
  void* operator new(size_t sz) throw(bad_alloc){
    cout << "NoMemory::operator new" << endl;
    throw bad_alloc(); // Нехватка памяти
  }
};

int main() {
  NoMemory* nm = 0;
  try {
    nm = new NoMemory;
  } catch(bad_alloc) {
    cerr << "Out of memory exception" << endl;
  }
  cout << "nm = " << nm << endl;
} ///:~
```

Если запустить эту программу, она не выведет сообщение конструктора — только сообщения функции operator new() и обработчика исключений. Нормального возврата из оператора new не происходит никогда, поэтому конструктор никогда не вызывается и его сообщение не выводится.

Очень важно, чтобы переменная nm инициализировалась нулем — поскольку выражение new завершается некорректно, указатель должен быть равен нулю, чтобы предотвратить его возможное использование в программе. С другой стороны, обработчик исключения должен делать нечто большее, чем просто выводить сообщение и продолжать работу так, словно создание объекта прошло успешно. В идеальном варианте следует как-то восстановить нормальную работу программы после ошибки или, по крайней мере, завершить ее после сохранения информации об ошибке.

В ранних версиях C++ при неудачной попытке выделения памяти было принято, чтобы оператор new возвращал 0, чтобы предотвратить вызов конструктора. Но если попытаться вернуть 0 в компиляторе, соответствующем стандарту, компилятор предложит вместо этого сгенерировать исключение bad_alloc.

Операторы new и delete с дополнительными аргументами

У перегруженной функции operator new() существуют два других, менее распространенных применения.

- Иногда объект требуется разместить в определенном участке памяти. Это особенно важно во встроенных системах, ориентированных на обслуживание аппаратуры, где объект может представлять некоторое устройство.
- В некоторых ситуациях при вызове new бывает удобно выбрать один из нескольких распределителей памяти.

Обе задачи решаются одним способом: перегруженная функция operator new() может получать более одного аргумента. Как было показано ранее, первый аргумент всегда определяет размер объекта; он незаметно вычисляется и передается компилятором. Но другие аргументы могут содержать все, что угодно, — адрес, по которому должен размещаться объект, ссылку на функцию выделения памяти или объект, другие данные.

Способ передачи дополнительных аргументов функции operator new() в процессе вызова на первый взгляд выглядит довольно любопытно. Список аргументов (*без* аргумента size_t, автоматически обрабатываемого компилятором) размещается после ключевого слова new и перед именем класса, объект которого вы создаете. Например, следующая команда передает a во втором аргументе функции operator new():

```
X* xp = new(a) X;
```

Конечно, это возможно только в том случае, если функция operator new() была объявлена в классе.

Следующий пример показывает, как разместить объект в заданном месте памяти:

```
//: C13:PlacementOperatorNew.cpp
// Размещение по заданному адресу с использованием функции operator new()
#include <cstddef> // Size_t
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {
    cout << "this = " << this << endl;
  }
  ~X() {
    cout << "X::~X(): " << this << endl;
  }
  void* operator new(size_t, void* loc) {
    return loc;
  }
};

int main() {
  int l[10];
  cout << "l = " << l << endl;
  X* xp = new(l) X(47); // X по адресу l
  xp->X::~X(); // Явный вызов деструктора
  // Использовать ТОЛЬКО при размещении объекта по заданному адресу!
} ///:~
```

Обратите внимание: функция operator new просто возвращает переданный ей указатель. Таким образом, вызывающая сторона решает, где должен находиться объект, а вызов конструктора для этой памяти является частью выражения new.

Хотя в этом примере используется всего один дополнительный аргумент, ничто не мешает вам передать другие аргументы, если они потребуются для других целей.

При уничтожении объекта мы сталкиваемся с дилеммой. Существует всего одна версия функции operator delete, поэтому нельзя заявить об использовании для этого объекта специальной версии функции освобождения памяти. Нужно вызвать деструктор, но не освобождать память объекта с помощью механизма динамического выделения памяти, поскольку она не была выделена из кучи.

На помощь приходит весьма специфический синтаксис. Вы можете явно вызвать деструктор в программе, это делается так:

```
xp->X::~X();  // Явный вызов деструктора
```

Здесь самое место для строгого предупреждения! Некоторые программисты рассматривают этот синтаксис как средство уничтожения объектов до их выхода из блока (вместо того, чтобы либо отрегулировать границы блока, либо, что правильнее, создавать объекты динамически). Однако для обычного объекта, созданного в стеке, подобный вызов деструктора становится источником серьезных проблем, поскольку деструктор будет повторно вызван в конце блока. Если же деструктор вызывается таким образом для объекта, созданного в куче, то, несмотря на вызов деструктора, память не освободится — вероятно, это совсем не то, на что вы рассчитывали. Единственная причина для явного вызова деструктора — поддержка синтаксиса функции operator new с дополнительными аргументами.

Также существует специальная версия функции operator delete, которая вызывается только в том случае, если конструктор для выражения new с дополнительными аргументами генерирует исключение (чтобы память была автоматически очищена при обработке исключения). Специальная версия функции operator delete имеет аргументы, которые соответствуют аргументам специальной версии функции operator new, вызванной перед тем, как конструктор сгенерирует исключение. Данная тема будет рассматриваться в главе, посвященной исключениям, второго тома книги.

Итоги

Создание автоматических объектов в стеке удобно и наиболее эффективно. Но для решения общих задач программирования должны существовать средства для создания и уничтожения объектов в любой момент во время выполнения, особенно в ответ на поступление внешней информации. Хотя для распределения памяти в куче можно воспользоваться функциями динамического выделения памяти C, эти функции не обеспечивают простоты использования и гарантий конструирования объектов, необходимых в языке C++. Поддержка динамического создания объектов на уровне языка операторами new и delete позволяет создавать объекты в куче так же легко, как они создаются в стеке. Кроме того, эти операторы чрезвычайно гибки. Программист может изменить поведение операторов new и delete, если они его почему-либо не устраивают, например, по соображениям эффективности. Наконец, программист может выбрать собственный вариант действий, выполняемых при исчерпании свободной памяти в куче.

Упражнения

1. Создайте класс Counted, содержащий переменные int id и static int count. Конструктор по умолчанию должен начинаться так: Counted() : id(count++) {. Он также должен выводить значение id и сообщать о создании объекта. Деструктор должен сообщать об уничтожении объекта и выводить свое значение id. Протестируйте класс.
2. Убедитесь в том, что операторы new и delete всегда вызывают конструкторы и деструкторы. Для этого создайте объект класса Counted (из упражнения 1)

оператором new и уничтожьте его оператором delete. Также создайте в куче и уничтожьте массив этих объектов.

3. Создайте объект PStash и заполните его объектами из упражнения 1, созданными оператором new. Проследите за тем, что происходит при выходе объекта PStash из области видимости и вызове его деструктора.
4. Создайте вектор vector<Counted*> и заполните его указателями на объекты Counted (см. упражнение 1), созданные оператором new. Переберите элементы вектора и выведите объекты Counted, затем переберите их снова и вызовите оператор delete для каждого объекта.
5. Повторите упражнение 4, включив в класс Counted функцию f() для вывода сообщения. Переберите элементы вектора и вызовите f() для каждого объекта.
6. Повторите упражнение 5 для класса PStash.
7. Повторите упражнение 5, используя файл Stack4.h из главы 9.
8. Динамически создайте массив объектов класса Counted (см. упражнение 1). Вызовите оператор delete для полученного указателя *без квадратных скобок*. Объясните результат.
9. Создайте объект класса Counted (см. упражнение 1) оператором new, преобразуйте полученный указатель в void* и вызовите для него оператор delete. Объясните результат.
10. Выполните пример NewHandler.cpp на своем компьютере, проверьте полученное значение count. Вычислите объем памяти в куче, доступной для вашей программы.
11. Создайте класс с перегруженными операторами new и delete для одиночных объектов и для массивов. Продемонстрируйте, что обе версии правильно работают.
12. Разработайте для примера Framis.cpp тест, по результатам которого можно было бы примерно оценить, насколько пользовательские версии операторов new и delete работают быстрее глобальных версий.
13. Измените пример NoMemory.cpp так, чтобы он содержал массив int и реально выделял память, вместо того чтобы генерировать исключение bad_alloc. Включите в функцию main() цикл while (наподобие того, как это сделано в примере NewHandler.cpp) для исчерпания свободной памяти и посмотрите, что произойдет, если ваша функция operator new не будет проверять успешность выделения памяти. Затем включите такую проверку в функцию operator new, генерируя исключение bad_alloc.
14. Создайте класс со специальной версией оператора new, получающей второй аргумент типа string. Этот класс должен содержать переменную static vector<string>, в которой сохраняется аргумент. Специальная версия new должна выделять память обычным образом. В main() вызовите new со строковыми аргументами, содержащими описания вызовов (попробуйте воспользоваться препроцессорными макросами __FILE__ и __LINE__).

15. Измените пример ArrayOperatorNew.cpp, добавив в класс переменную static vector<Widget*>. Класс должен сохранять в векторе адрес каждого объекта Widget, созданного вызовом operator new(), и удалять его при освобождении объекта вызовом operator delete() (информацию о векторе можно найти в документации по стандартной библиотеке C++ или во втором томе книги). Создайте второй класс с именем MemoryChecker с деструктором, который выводит количество указателей на Widget в векторе. Создайте программу с одним глобальным экземпляром объекта MemoryChecker; в функции main() динамически создайте и уничтожьте несколько объектов и массивов Widget. Покажите, что MemoryChecker обнаруживает утечки памяти.

Наследование и композиция

Одной из самых привлекательных особенностей C++ является возможность многократного использования программного кода. Конечно, чтобы эту возможность можно было назвать действительно новой и революционной, она не должна сводиться к простому копированию и вставке кода. Такой подход был характерен для C, и нельзя сказать, что там он применялся успешно. Как обычно в C++, проблема решается на уровне классов. Чтобы заново использовать существующий код, программист создает новый класс, но не «с нуля», а на базе готовых классов, построенных и отлаженных кем-то другим.

Фокус заключается в том, чтобы задействовать классы, не нарушая работоспособности существующего кода. В этой главе будут продемонстрированы два способа решения задачи. Первый способ прост и прямолинеен: объекты существующих классов создаются внутри новых классов. Это называется *композицией*, поскольку новый (композитный) класс «собирается» из объектов существующих классов.

Второй способ не столь тривиален. Новый класс создается как *подтип* существующего класса. Программист буквально берет существующий класс как «заготовку» и добавляет в него новый код, не изменяя существующего класса. Это мистическое действо называется *наследованием*, и большая часть работы по его реализации выполняется компилятором. Наследование является краеугольным камнем объектно-ориентированного программирования, а его применение приводит к дополнительным последствиям, которые будут рассматриваться в главе 15.

Оказывается, композиция и наследование обладают сходным синтаксисом и поведением (и это логично: оба способа предназначены для создания новых типов на базе существующих). В этой главе вы познакомитесь с этими механизмами многократного использования кода.

Синтаксис композиции

Вообще говоря, мы уже неоднократно применяли композицию для создания новых классов. Классы в приводимых примерах в основном строились из встроенных

типов (иногда из строк). Оказывается, для пользовательских типов композиция реализуется почти так же просто.

Допустим, следующий класс почему-либо представляет ценность для нас:

```
//: C14:Useful.h
// Класс, который должен использоваться многократно
#ifndef USEFUL_H
#define USEFUL_H

class X {
  int i;
public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};
#endif // USEFUL_H ///:~
```

Переменные класса объявлены закрытыми, поэтому мы можем абсолютно безопасно включить объект типа X в новый класс как открытый (public) объект, что значительно упрощает интерфейс класса:

```
//: C14:Composition.cpp
// Многократное использование кода посредством композиции
#include "Useful.h"

class Y {
  int i;
public:
  X x; // Внутренний объект
  Y() { i = 0; }
  void f(int ii) { i = ii; }
  int g() const { return i; }
};

int main() {
  Y y;
  y.f(47);
  y.x.set(37); // Обращение к внутреннему объекту
} ///:~
```

Обращение к функциям внутреннего объекта (в дальнейшем мы будем называть его *подобъектом*) просто требует дополнительного уточнения имени объекта.

На практике внутренние объекты чаще объявляются закрытыми и, таким образом, становятся частью базовой реализации (это означает, что при желании реализацию можно изменить). Функции открытого интерфейса нового класса работают с внутренним объектом, но не обязательно повторяют его интерфейс:

```
//: C14:Composition2.cpp
// Закрытые внутренние объекты
#include "Useful.h"

class Y {
  int i;
  X x; // Внутренний объект
public:
  Y() { i = 0; }
  void f(int ii) { i = ii; x.set(ii); }
```

```
  int g() const { return i * x.read(); }
  void permute() { x.permute(); }
};

int main() {
  Y y;
  y.f(47);
  y.permute();
} ///:~
```

Функция внутреннего объекта permute() переносится в интерфейс композитного класса, а остальные функции класса X используются в функциях Y.

Синтаксис наследования

Синтаксис композиции очевиден, а для наследования применяется совершенно иная синтаксическая форма.

Используя наследование, вы фактически говорите: «Новый класс похож на такой-то существующий класс». Для этого в программе, как обычно, задается имя класса, но перед открывающей фигурной скобкой через двоеточие указывается имя *базового класса* (или базовых классов, разделенных запятыми, в случае множественного наследования). После этого в новый класс автоматически включаются все переменные и функции базового класса. Пример:

```
//: C14:Inheritance.cpp
// Простое наследование
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X {
  int i; // Переменная i отлична от i класса X
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // Вызов по другому имени
    return i;
  }
  void set(int ii) {
    i = ii;
    X::set(ii); // Вызов по тому же имени
  }
};

int main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // Обращение к интерфейсу функций X:
  D.read();
  D.permute();
  // Переопределенные функции скрывают базовые версии:
  D.set(12);
} ///:~
```

Класс Y наследует от X; это означает, что Y содержит все переменные класса X, а также его функции. Фактически Y содержит подобъект X точно так же, как если бы вы создали объект X внутри Y, вместо того чтобы наследовать от X.

Все закрытые элементы X остаются закрытыми в классе Y; иначе говоря, тот факт, что Y наследует от X, не означает, что Y может нарушить работу механизма защиты. Закрытые элементы X никуда не исчезли, они по-прежнему находятся в памяти, — просто к ним нельзя обращаться напрямую.

Из функции main() видно, что элементы данных Y объединяются с данными X, потому что размер, возвращаемый функцией sizeof(Y), вдвое больше размера, возвращаемого функцией sizeof(X).

Обратите внимание: перед именем базового класса находится ключевое слово public. При наследовании по умолчанию используется уровень доступа private. Если бы перед именем базового класса не было ключевого слова public, то все открытые члены базового класса стали бы закрытыми в производном классе. Такое поведение почти всегда нежелательно: обычно открытые члены базового класса должны оставаться открытыми в производном классе. Для этого при наследовании указывается ключевое слово public.

В функции change() вызывается функция permute() базового класса. Производный класс может напрямую вызывать все открытые функции базового класса.

Функция set() производного класса *переопределяет* функцию set() базового класса. Иначе говоря, при вызове функций read() и permute() для объекта типа Y будут вызваны версии базового класса (в функции main() показано, как это происходит). Но если вызвать для объекта Y функцию set(), будет вызвана переопределенная версия. Следовательно, если вас не устраивает то, как работает унаследованная функция, вы можете изменить ее, а также добавить новые функции, такие как change().

Однако при переопределении функций все же должна сохраняться возможность вызова их версий для базового класса. Если внутри set() просто вызвать set(), то будет вызвана локальная версия функции, то есть возникнет рекурсия. Чтобы вызвать версию функции базового класса, необходимо явно задать имя базового класса и оператор уточнения области видимости (::).

Список инициализирующих значений конструктора

Вы уже знаете, какая важная роль в C++ отводится гарантированной инициализации; это в полной мере относится к композиции и наследованию. При создании объекта компилятор гарантирует вызов конструктора для всех его подобъектов. До настоящего момента все подобъекты имели конструкторы по умолчанию, которые автоматически вызывались компилятором. Но что, если подобъекты не имеют конструкторов по умолчанию или вы захотите изменить аргумент конструктора? Возникает проблема, поскольку конструктор нового класса не имеет доступа к закрытым данным подобъекта, а значит, не может инициализировать их напрямую.

Однако проблема решается просто: следует вызвать конструктор подобъекта. В C++ для этого предусмотрен специальный синтаксис, называемый *списком инициализирующих значений конструктора*. Синтаксис списка инициализирующих

значений конструктора имитирует синтаксис наследования. При наследовании базовые классы перечисляются после двоеточия, но до открывающей фигурной скобки тела класса. В списке инициализирующих значений вызовы конструкторов подобъектов размещаются после списка аргументов конструктора и двоеточия, но перед открывающей фигурной скобкой тела функции. Для класса MyType, наследующего от Bar, это может выглядеть так (если у Bar имеется конструктор, вызываемый с одним аргументом типа int):

```
MyType::MyType(int i) : Bar(i) { // ...
```

Инициализация объектов внутри класса

Оказывается, синтаксис инициализации объектов также подходит и для композиции. Правда, в этом случае указываются имена объектов вместо имен классов. Если список инициализирующих значений содержит более одного вызова конструктора, то вызовы разделяются запятыми:

```
MyType2::MyType2(int i) : Bar(i), m(i+1) { // ...
```

Так начинается конструктор класса MyType2, наследующего от Bar и содержащего переменную m. Обратите внимание: хотя в списке инициализирующих значений конструктора присутствует тип базового класса, для внутреннего объекта указывается его идентификатор.

Встроенные типы в списке инициализирующих значений

Список инициализирующих значений конструктора позволяет явно вызывать конструкторы для внутренних объектов. Более того, вызвать эти конструкторы другим способом нельзя. Идея заключается в том, что все конструкторы внутренних объектов должны быть вызваны до передачи управления в тело конструктора нового класса. В этом случае любые вызовы функций внутренних объектов всегда будут относиться к инициализированным объектам. Невозможно войти за открывающую фигурную скобку конструктора без вызова *какого-либо* конструктора для каждого внутреннего объекта и объектов базовых классов, даже если компилятору приходится скрыто вызывать конструктор по умолчанию. Это дополнительно подкрепляет гарантии C++ относительно того, что ни один объект (или его часть) не может использоваться в программе без предварительного вызова его конструктора.

Гарантия инициализации внутренних объектов к моменту достижения открывающей фигурной скобки конструктора также оказывается удобной для программиста. Достигнув этой точки, можно предположить, что все внутренние объекты были успешно инициализированы, и сосредоточиться на конкретных задачах, выполняемых конструктором. Но тут кроется подвох: что делать с объектами встроенных типов, *не имеющих* конструкторов?

Чтобы синтаксис стал более последовательным, было решено интерпретировать встроенные типы так, словно у них имеется конструктор с единственным аргументом, тип которого соответствует типу инициализируемой переменной. Таким образом, возможна инициализация вида

```
//: C14:PseudoConstructor.cpp
class X {
```

```
  int i;
  float f;
  char c;
  char* s;
public:
  X() : i(7), f(1.4), c('x'), s("howdy") {}
};

int main() {
  X x;
  int i(100); // Применение к обычному определению
  int* ip = new int(47);
} ///:~
```

Все эти «вызовы псевдоконструкторов» просто присваивают значения соответствующим переменным. Этот прием удобен и считается хорошим стилем программирования, поэтому вы часто будете встречать его в программах.

Синтаксис «псевдоконструкторов» может использоваться даже при создании переменных встроенных типов за пределами классов:

```
int i(100);
int* ip = new int(47);
```

Такой способ инициализации несколько приближает встроенные типы к объектам. Однако следует помнить, что речь идет не о настоящих конструкторах. В частности, если псевдоконструктор не будет явно вызван в программе, инициализация окажется невыполненной.

Объединение композиции с наследованием

Конечно, композиция может использоваться совместно с наследованием. В следующем примере продемонстрировано создание более сложного класса с применением обоих механизмов:

```
//: C14:Combined.cpp
// Наследование и композиция

class A {
  int i;
public:
  A(int ii) : i(ii) {}
  ~A() {}
  void f() const {}
};

class B {
  int i;
public:
  B(int ii) : i(ii) {}
  ~B() {}
  void f() const {}
};

class C : public B {
  A a;
public:
  C(int ii) : B(ii), a(ii) {}
```

```
  ~C() {} // Вызывает ~A() и ~B()
  void f() const {  // Переопределение
    a.f();
    B::f();
  }
};

int main() {
  C c(47);
} ///:~
```

Класс C наследует от B и содержит внутренний объект типа A. Из листинга видно, что список инициализирующих значений конструктора содержит вызовы конструкторов как базового класса, так и внутреннего объекта.

Функция C::f() переопределяет унаследованную функцию B::f(), а также вызывает версию базового класса. Кроме того, она вызывает a.f(). Следует помнить, что переопределение функций может выполняться только путем наследования; при использовании внутренних объектов можно работать с открытым интерфейсом объекта, но переопределять его нельзя. Кроме того, если функция C::f() не определена, то вызов f() для объекта класса C не приведет к вызову a.f(); вместо этого *будет* вызвана функция B::c().

Автоматический вызов деструктора

Хотя необходимость в явном вызове конструкторов в списке инициализирующих значений возникает достаточно часто, вам никогда не понадобится явно вызывать деструкторы, поскольку в каждом классе может быть только один деструктор, и он вызывается без аргументов. Тем не менее компилятор по-прежнему гарантирует вызов *всех* деструкторов, то есть всех деструкторов во всей иерархии наследования, начиная с последних классов в цепочке наследования и до корневого базового класса.

Стоит особо подчеркнуть, что конструкторы и деструкторы принципиально отличаются от других функций: они вызываются все от начала и до конца иерархии, тогда как при вызове обычной функции вызывается только эта функция, но не ее версии из базовых классов. Если вы хотите вызвать базовую версию обычной переопределенной функции, это придется сделать явно.

Порядок вызова конструкторов и деструкторов

Интересно разобраться в том, в каком порядке вызываются конструкторы и деструкторы при большом количестве подобъектов. Следующий пример позволяет точно понять, что при этом происходит:

```
//: C14:Order.cpp
// Порядок вызова конструкторов и деструкторов
#include <fstream>
using namespace std;
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
  ID(int) { out << #ID " constructor\n"; } \
  ~ID() { out << #ID " destructor\n"; } \
```

```
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);

class Derived1 : public Base1 {
  Member1 m1;
  Member2 m2;
public:
  Derived1(int) : m2(1), m1(2), Base1(3) {
    out << "Derived1 constructor\n";
  }
  ~Derived1() {
    out << "Derived1 destructor\n";
  }
};

class Derived2 : public Derived1 {
  Member3 m3;
  Member4 m4;
public:
  Derived2() : m3(1), Derived1(2), m4(3) {
    out << "Derived2 constructor\n";
  }
  ~Derived2() {
    out << "Derived2 destructor\n";
  }
};

int main() {
  Derived2 d2;
} ///:~
```

Сначала мы создаем объект ofstream для записи всех результатов в файл. Затем для уменьшения объема работы мы определяем макрос для построения классов, который затем будет использоваться при наследовании и композиции (в главе 16 мы применим гораздо более совершенную методику). Каждый конструктор и деструктор выводит сообщение о своем вызове в файл. Обратите внимание: конструкторы не являются конструкторами по умолчанию; у них имеется аргумент типа int. Этот аргумент не имеет идентификатора; он существует лишь для того, чтобы заставить нас явно вызывать конструкторы в списке инициализирующих значений (аргумент объявляется без идентификатора, чтобы предотвратить выдачу предупреждений компилятором).

Результат выполнения программы:

```
Base1 constructor
Member1 constructor
Member2 constructor
Derived1 constructor
Member3 constructor
Member4 constructor
Derived2 constructor
Derived2 destructor
Member4 destructor
Member3 destructor
```

```
Derived1 destructor
Member2 destructor
Member1 destructor
Member4 destructor
Base1 destructor
```

Видно, что конструирование начинается с корня иерархии классов, и на каждом уровне сначала вызывается конструктор базового класса, а затем конструкторы внутренних объектов. Порядок вызова деструкторов в точности противоположен порядку вызова конструкторов; это важно из-за потенциальных зависимостей (в конструкторе или деструкторе производного класса должно действовать допущение о том, что подобъект базового класса может использоваться и он уже сконструирован или еще не уничтожен).

Также интересно заметить, что порядок вызова конструкторов для внутренних объектов совершенно не зависит от порядка вызовов в списке инициализирующих значений. Он полностью определяется порядком объявления внутренних объектов в классе. Если бы порядок вызова конструкторов можно было изменить при помощи списка инициализирующих значений, то у двух разных конструкторов могли бы быть две разные последовательности вызовов, а бедный деструктор не знал бы, как правильно выстроить обратный порядок вызовов при уничтожении объектов. В итоге возникла бы проблема зависимости.

Скрытие имен

Если вы объявляете производный класс и предоставляете новое определение для одной из его функций, возможны два варианта. Сигнатура и тип возвращаемого значения определения в производном классе могут точно соответствовать определению в базовом классе. Но что происходит, если в производном классе изменяется список аргументов функции или возвращаемое значение? Пример:

```
//: C14:NameHiding.cpp
// Замещение перегруженных имен в результате наследования
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Переопределение:
```

```
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // Изменение типа возвращаемого значения:
  void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base {
public:
  // Изменение списка аргументов:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // Строковая версия скрывается
  Derived3 d3;
//!  x = d3.f(); // Версия с возвращаемым значением int скрывается
  Derived4 d4;
//!  x = d4.f(); // Версия f() скрывается
  x = d4.f(1);
} ///:~
```

В классе Base присутствует перегружаемая функция f(). Класс Derived1 не изменяет f(), но переопределяет g(). В функции main() видно, что обе перегруженные версии f() доступны в Derived1. В классе Derived2 переопределяется лишь одна из двух перегруженных версий f(), в результате вторая перегруженная форма становится недоступной. В Derived3 изменение типа возвращаемого значения скрывает обе версии базового класса, а Derived4 показывает, что изменение списка аргументов также скрывает обе версии из базового класса.

В общем случае можно сказать, что любое переопределение перегруженного имени функции из базового класса автоматически скрывает все остальные версии в новом классе. В главе 15 будет показано, что ключевое слово virtual вносит некоторые дополнительные изменения в перегрузку.

Изменение интерфейса базового класса посредством модификации сигнатуры и/или типа возвращаемого значения означает, что класс используется не тем способом, для которого обычно предназначается наследование. Впрочем, это еще не говорит о том, что вы делаете что-то неверно. Просто конечной целью наследования является поддержка *полиморфизма*, а изменение сигнатуры функции или типа возвращаемого значения приводит к фактическому изменению интерфейса базового класса. Если вы именно этого и добивались, значит, наследование применяется в основном для многократного использования кода, а не для сохранения едино-

го интерфейса базового класса (важного аспекта полиморфизма). Как правило, такой вариант наследования означает, что класс общего назначения специализируется для конкретного случая, а для таких задач обычно (хотя и не всегда) рекомендуется применять композицию.

Для примера возьмем класс Stack из главы 9. Один из недостатков этого класса — необходимость приведения типа при каждой выборке указателя из контейнера. Частые операции приведения типа не только утомительны, но и небезопасны, поскольку указатель может быть приведен к любому типу на усмотрение программиста.

Решение, которое на первый взгляд кажется более удачным, основано на специализации обобщенного класса Stack посредством наследования. Ниже приведен пример с использованием класса из главы 9:

```
//: C14:InheritStack.cpp
// Специализация класса Stack
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack : public Stack {
public:
  void push(string* str) {
    Stack::push(str);
  }
  string* peek() const {
    return (string*)Stack::peek();
  }
  string* pop() {
    return (string*)Stack::pop();
  }
  ~StringStack() {
    string* top = pop();
    while(top) {
      delete top;
      top = pop();
    }
  }
};

int main() {
  ifstream in("InheritStack.cpp");
  assure(in, "InheritStack.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) { // Без преобразования!
    cout << *s << endl;
    delete s;
  }
} ///:~
```

Так как все функции в файле Stack4.h оформлены как подставляемые, никакой дополнительной компоновки не требуется.

Класс StringStack специализирует Stack так, что функция push() принимает только указатели на String. Раньше класс Stack принимал указатели на void, поэтому пользователь не мог воспользоваться средствами проверки типа и убедиться в том, что в контейнере сохраняются правильные указатели. Кроме того, функции peek() и pop() в новой версии возвращают указатели на String вместо указателей на void, поэтому использование указателя не требует приведения типа.

Но самое интересное заключается в том, что повышение безопасности за счет проверки типов в функциях push(), peek() и pop() достигается практически без издержек! Компилятор получает дополнительную информацию о типе, которую он использует на стадии компиляции, а подстановка функций предотвращает появление лишнего кода.

В этом примере также задействован механизм скрытия имен; у функции push() изменилась сигнатура, а именно список аргументов. Если бы две версии push() находились в одном классе, то функция была бы перегружена. Однако в нашем случае перегрузка *нежелательна*, поскольку она бы позволила передать push() любой тип указателя в виде void*. К счастью, C++ скрывает версию push(void*) базового класса ради новой версии, определенной в производном классе, а следовательно, позволяет заносить в StringStack функцией push() только указатели на string.

Поскольку в новой версии точно известен тип объектов, хранящихся в контейнере, деструктор работает так, как положено, и проблема принадлежности объектов успешно решается или, по крайней мере, появляется один из возможных подходов к ее решению. Сохранение в StringStack указателя на string функцией push() приводит (в соответствии с семантикой StringStack) к передаче прав на этот объект контейнеру. При извлечении указателя функцией pop() вы получаете не только указатель, но и права владения этим указателем. Все указатели, остающиеся в StringStack на момент вызова деструктора, будут уничтожены деструктором. Но так как эти указатели всегда ссылаются на string, а команда delete работает с указателями на string вместо указателей на void, объекты успешно уничтожаются, и все работает правильно.

У представленного решения есть недостаток: этот класс работает *только* с указателями на string. Если вам понадобится контейнер Stack для работы с другим типом объектов, придется писать новую версию класса, рассчитанную только на новый тип. Подобное дублирование классов быстро начинает утомлять. В конечном счете задача решается при помощи шаблонов, как будет показано в главе 16.

И еще одно замечание по поводу этого примера: в процессе наследования изменяется интерфейс класса Stack. Если интерфейс изменился, то StringStack уже нельзя рассматривать как частный случай Stack, поэтому вы никогда не сможете использовать StringStack вместо Stack. В результате наследование начинает выглядеть сомнительно: если создаваемый класс StringStack *не является* подтипом Stack, то зачем же наследовать? Более подходящая версия класса StringStack приводится далее в этой главе.

Функции, которые не наследуются автоматически

Не все функции автоматически наследуются производным классом от базового. Конструкторы и деструкторы участвуют в создании и уничтожении объекта, поэтому они умеют работать только с аспектами объекта, относящимися к их кон-

кретному классу. Для остальных аспектов приходится вызывать конструкторы и деструкторы классов, расположенных ниже в иерархии наследования. Из-за этого конструкторы и деструкторы не наследуются, а явно определяются для каждого производного класса.

Функция operator= тоже не наследуется, поскольку выполняемые ею действия сходны с конструированием. Другими словами, даже если вы знаете, как присвоить значения переменных объекта, находящегося справа от оператора =, переменным объекта, находящимся слева, это вовсе не означает, что смысл присваивания не изменится в результате наследования.

При наследовании эти функции генерируются компилятором, если они не были определены явно (причем для того, чтобы компилятор мог сгенерировать конструктор по умолчанию и копирующий конструктор, в классе не может быть определено *ни одного* конструктора). Эта тема кратко рассматривалась в главе 6. Сгенерированные конструкторы используют инициализацию, а сгенерированная функция operator= — присваивание на уровне переменных. Рассмотрим пример с функциями, автоматически сгенерированными компилятором:

```
//: C14:SynthesizedFunctions.cpp
// Функции, сгенерированные компилятором
#include <iostream>
using namespace std;

class GameBoard {
public:
  GameBoard() { cout << "GameBoard()\n"; }
  GameBoard(const GameBoard&) {
    cout << "GameBoard(const GameBoard&)\n";
  }
  GameBoard& operator=(const GameBoard&) {
    cout << "GameBoard::operator=()\n";
    return *this;
  }
  ~GameBoard() { cout << "~GameBoard()\n"; }
};

class Game {
  GameBoard gb; // Композиция
public:
  // Вызов конструктора по умолчанию для GameBoard:
  Game() { cout << "Game()\n"; }
  // Копирующий конструктор GameBoard должен вызываться явно,
  // иначе вместо него будет автоматически вызван
  // конструктор по умолчанию:
  Game(const Game& g) : gb(g.gb) {
    cout << "Game(const Game&)\n";
  }
  Game(int) { cout << "Game(int)\n"; }
  Game& operator=(const Game& g) {
    // Оператор присваивания для GameBoard должен вызываться явно,
    // иначе gb вообще не будет присвоено значение!
    gb = g.gb;
    cout << "Game::operator=()\n";
    return *this;
  }
  class Other {}; // Вложенный класс
  // Автоматическое приведение типа:
```

```
    operator Other() const {
      cout << "Game::operator Other()\n";
      return Other();
    }
    ~Game() { cout << "~Game()\n"; }
};

class Chess : public Game {};

void f(Game::Other) {}

class Checkers : public Game {
public:
  // Вызов конструктора по умолчанию для базового класса:
  Checkers() { cout << "Checkers()\n"; }
  // Копирующий конструктор базового класса должен вызываться явно,
  // иначе вместо него будет автоматически вызван
  // конструктор по умолчанию:
  Checkers(const Checkers& c) : Game(c) {
    cout << "Checkers(const Checkers& c)\n";
  }
  Checkers& operator=(const Checkers& c) {
    // Версия operator=() для базового класса должна вызываться явно,
    // иначе подобъект базового класса не участвует в присваивании!
    Game::operator=(c);
    cout << "Checkers::operator=()\n";
    return *this;
  }
};

int main() {
  Chess d1;  // Конструктор по умолчанию
  Chess d2(d1); // Копирующий конструктор
//! Chess d3(1); // Ошибка: нет конструктора с int
  d1 = d2; // Генерируется функция operator=
  f(d1); // Приведение типа НАСЛЕДУЕТСЯ
  Game::Other go;
//!  d1 = go; // Для другого типа operator= не генерируется
  Checkers c1, c2(c1);
  c1 = c2;
} ///:~
```

Конструкторы и функция operator= для GameBoard и Game выводят сообщения о своем вызове, чтобы вы видели, когда они используются компилятором. Кроме того, функция operator Other() выполняет автоматическое приведение типа объекта Game к типу объекта вложенного класса Other. Класс Chess просто наследует от Game и не определяет новых функций (чтобы вы могли понаблюдать за реакцией компилятора). Функция f() получает объект Other для тестирования операции автоматического приведения типа.

В функции main() вызываются сгенерированные конструктор по умолчанию и копирующий конструктор производного класса Chess. Версии этих конструкторов класса Game вызываются в составе иерархии вызовов конструкторов. Хотя внешне все выглядит как наследование, новые конструкторы в действительности генерируются компилятором. Как и следовало ожидать, конструкторы с аргументами компилятором не создаются, поскольку это потребовало бы слишком большой проницательности со стороны компилятора.

Функция operator= также генерируется как новая функция класса Chess с применением поразрядного присваивания (то есть с вызовом версии из базового класса), поскольку эта функция не была явно включена в новый класс. Конечно, деструктор тоже автоматически генерируется компилятором.

На фоне многочисленных правил переопределения функций, связанных с созданием объектов, наследование оператора автоматического приведения типа поначалу выглядит немного странно. Однако в действительности это вполне разумно — если Game содержит достаточно фрагментов для создания объекта Other, эти фрагменты останутся в любом классе, производном от Game, поэтому оператор приведения типа остается действительным (даже если вы захотите переопределить его).

Функция operator= генерируется *только* для присваивания однотипных объектов. Чтобы присвоить объект одного типа объекту другого типа, функцию operator= всегда приходится определять самостоятельно.

Если внимательнее присмотреться к Game, можно заметить, что копирующий конструктор и операторы присваивания содержат явные вызовы копирующего конструктора и оператора присваивания для внутреннего объекта. Обычно такие вызовы должны присутствовать в программе, поскольку без них для копирующего конструктора будет вызван конструктор по умолчанию внутреннего объекта, а в случае оператора присваивания для внутреннего объекта присваивание *вообще* не выполняется!

Напоследок взгляните на класс Checkers, в котором явно определены конструктор по умолчанию, копирующий конструктор и операторы присваивания. Для конструктора по умолчанию автоматически вызывается конструктор по умолчанию базового класса, и обычно это именно то, что требуется. Но важнее другое: как только вы решите написать собственные копирующий конструктор и оператор присваивания, компилятор считает, что вы знаете, что делаете, и *не вызывает* автоматически версии базового класса, как это делается в сгенерированных функциях. Если вам требуются версии базового класса (а они обычно требуются), вам придется явно вызвать их в программе. В копирующем конструкторе Checkers вызов производится из списка инициализирующих значений:

```
Checkers(const Checkers& c) : Game(c) {
```

В операторе присваивания Checkers вызов версии базового класса производится в самом начале тела функции:

```
Game::operator=(c);
```

Привыкайте к тому, что эти вызовы являются частью канонической формы, которая всегда используется при наследовании.

Наследование и статические функции классов

Статические функции классов при наследовании ведут себя так же, как и обычные функции.

- Они наследуются производными классами.
- Переопределение статической функции класса скрывает все остальные перегруженные версии в базовом классе.
- Изменение сигнатуры функции приводит к скрытию всех версий из базового класса с тем же именем (впрочем, это «вариации на тему» предыдущего пункта).

Тем не менее статические функции класса не могут быть виртуальными (эта тема подробно рассматривается в главе 15).

Выбор между композицией и наследованием

Как композиция, так и наследование приводят к созданию подобъектов в новом классе. В обоих случаях эти подобъекты конструируются в списке инициализирующих значений конструктора. Возникает вопрос: чем различаются эти два способа и когда следует применять каждый из них?

Композиция обычно применяется тогда, когда новый класс должен поддерживать некоторые возможности существующего класса, но не его интерфейс. Иначе говоря, объект внедряется для реализации функциональности нового класса, но пользователь класса работает с ним не через интерфейс исходного класса, а через интерфейс, определенный программистом. Обычно это делается по классическому рецепту включения закрытых объектов существующих классов в новый класс.

Тем не менее в отдельных случаях бывает разумно предоставить пользователю класса прямой доступ к внутренним объектам, для чего внутренний объект объявляется открытым. Внутренние объекты сами контролируют доступ к себе, поэтому такое объявление безопасно. С другой стороны, когда пользователь знает, из каких частей состоит объект, ему бывает проще понять его интерфейс. Хорошим примером служит класс Car («автомобиль»):

```
//: C14:Car.cpp
// Композиция с открытыми внутренними объектами

class Engine {
public:
  void start() const {}
  void rev() const {}
  void stop() const {}
};

class Wheel {
public:
  void inflate(int psi) const {}
};

class Window {
public:
  void rollup() const {}
  void rolldown() const {}
};

class Door {
public:
  Window window;
  void open() const {}
  void close() const {}
};

class Car {
public:
  Engine engine;
```

```
  Wheel wheel[4];
  Door left, right; // У автомобиля две двери
};

int main() {
  Car car;
  car.left.window.rollup();
  car.wheel[0].inflate(72);
} ///:~
```

Поскольку композиция Car входит в концептуальное представление проблемы (а не просто является частью базовой архитектуры), объявление внутренних объектов открытыми помогает прикладному программисту разобраться в том, как использовать этот класс, и упрощает программирование для создателя класса.

Если немного подумать, становится ясно, что нет смысла создавать Car посредством композиции с включением объекта Vehicle («транспортное средство»), поскольку автомобиль не содержит транспортное средство, а *является* им. Связи типа «а является частным случаем b» выражаются путем наследования, а связи «а содержит b» — путем композиции.

Выделение подтипов

Допустим, вы хотите создать разновидность объекта ifstream, которая не только открывает файл, но и запоминает имя открытого файла. Для этого можно воспользоваться композицией и включить в новый класс объекты ifstream и string:

```
//: C14:FName1.cpp
// Файловый поток, сохраняющий имя файла
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName1 {
  ifstream file;
  string fileName;
  bool named;
public:
  FName1() : named(false) {}
  FName1(const string& fname)
    : fileName(fname), file(fname.c_str()) {
    assure(file, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // Не заменять имя файла
    fileName = newName;
    named = true;
  }
  operator ifstream&() { return file; }
};

int main() {
  FName1 file("FName1.cpp");
  cout << file.name() << endl;
```

```
  // Ошибка: в классе нет функции с именем close():
//!  file.close();
} ///:~
```

При таком решении возникает проблема. Мы хотим, чтобы объект FName1 мог использоваться везде, где используются объекты ifstream, для чего определяется оператор автоматического приведения типа FName1 к ifstream. Однако в функции main() следующая команда не компилируется, потому что автоматическое приведение типа применяется только при вызовах функций, но не при обращении к функциям класса:

```
file.close();
```

Следовательно, такое решение не работает.

В другом решении в класс FName1 включается определение функции close():

```
void close() { file.close(); }
```

При небольшом количестве функций, переносимых из класса ifstream, такой подход работает. В этом случае мы используем только часть класса, и композиция при этом уместна.

Но что делать, если в новый класс должна быть включена вся функциональность прежнего класса? Такая ситуация называется *выделением подтипа*, потому что мы создаем новый тип на базе существующего типа. При этом новый тип имеет точно такой же интерфейс, как существующий тип (а также содержит новые функции, которые вы захотите в него добавить), чтобы он всегда мог использоваться вместо существующего типа. В таких ситуациях уместно применить наследование. Следующий пример показывает, что наследование идеально справляется с этой задачей:

```
//: C14:FName2.cpp
// Решение проблемы с применением наследования
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName2 : public ifstream {
  string fileName;
  bool named;
public:
  FName2() : named(false) {}
  FName2(const string& fname)
    : ifstream(fname.c_str()), fileName(fname) {
    assure(*this, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // Не заменять имя файла
    fileName = newName;
    named = true;
  }
};

int main() {
  FName2 file("FName2.cpp");
```

```
  assure(file, "FName2.cpp");
  cout << "name: " << file.name() << endl;
  string s;
  getline(file, s); // Эти вызовы тоже работают!
  file.seekg(-200, ios::end);
  file.close();
} ///:~
```

Теперь все функции объекта ifstream доступны для объекта FName2. Также мы видим, что внешние функции вроде getline(), получающие объекты ifstream, также работают с FName2. Дело в том, что FName2 не просто содержит объект ifstream, а *является* подтипом ifstream. Это очень важное различие, которое будет рассмотрено в конце этой главы и в следующей главе.

Закрытое наследование

Новый класс может быть связан с базовым классом закрытым наследованием; для этого следует убрать из списка базовых классов ключевое слово public или явно указать ключевое слово private (вероятно, такой вариант лучше, потому что он однозначно сообщает читателю о ваших намерениях). Закрытое наследование подразумевает «ограниченную реализацию»; другими словами, вы создаете новый класс, содержащий все данные и всю функциональность базового класса, но эта функциональность остается скрытой; она доступна только базовой реализации. Пользователь класса не обладает доступом к базовой реализации, а объект уже не может интерпретироваться как экземпляр базового класса (в отличие от FName2.cpp).

Возможно, у вас возник вопрос: зачем нужно закрытое наследование, если для создания закрытого подобъекта в новом классе существует другое, более подходящее средство, а именно композиция? Поддержка закрытого наследования была включена в язык для полноты картины, но на практике вместо него обычно лучше использовать композицию (хотя бы для предотвращения возможных недоразумений). Тем не менее в некоторых ситуациях требуется, чтобы объект поддерживал часть интерфейса базового класса, но при этом *не мог* интерпретироваться как объект базового класса. Закрытое наследование предоставляет такую возможность.

Доступ к закрыто унаследованным членам класса

В случае закрытого наследования все открытые члены базового класса становятся закрытыми. Если вы хотите, чтобы некоторые из них оставались видимыми в производном классе, просто перечислите их имена (без аргументов и возвращаемых значений) с ключевым словом using в секции public производного класса:

```
//: C14:PrivateInheritance.cpp
class Pet {
public:
  char eat() const { return 'a'; }
  int speak() const { return 2; }
  float sleep() const { return 3.0; }
  float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // Закрытое наследование
public:
  using Pet::eat; // Предоставление открытого доступа к функции
  using Pet::sleep; // Обе перечисленные функции становятся открытыми
```

```
};

int main() {
  Goldfish bob;
  bob.eat();
  bob.sleep();
  bob.sleep(1);
//! bob.speak();// Ошибка: закрытая функция производного класса
} ///:~
```

Итак, закрытое наследование полезно в тех случаях, когда вы хотите скрыть часть функциональности базового класса.

Учтите, что открытие доступа к имени перегруженной функции предоставляет доступ ко всем перегруженным версиям этой функции, содержащимся в базовом классе.

Хорошенько подумайте перед тем, как задействовать закрытое наследование вместо композиции; при идентификации типов на стадии выполнения закрытое наследование порождает некоторые специфические проблемы (эта тема рассматривается во втором томе книги).

Защищенность

После знакомства с наследованием упомянутое ключевое слово protected наконец-то обретает смысл. В идеале закрытые члены класса всегда должны оставаться закрытыми, но в реальных проектах иногда бывает удобно скрыть функцию или переменную класса от внешнего мира, но сохранить доступ к ней из производных классов. Ключевое слово protected отдает дань прагматизму; оно означает: «Закрытый с точки зрения пользователей класса, но доступный для всех классов, производных от данного».

Переменные класса лучше всего оставлять закрытыми — тем самым вы сохраняете возможность менять базовую реализацию по своему усмотрению. Доступ из производных классов контролируется при помощи защищенных функций класса:

```
//: C14:Protected.cpp
// Ключевое слово protected
#include <fstream>
using namespace std;

class Base {
  int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int value(int m) const { return m*i; }
};

class Derived : public Base {
  int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }
};
```

```
int main() {
  Derived d;
  d.change(10);
} ///:~
```

Примеры, в которых может пригодиться ключевое слово **protected**, будут приведены далее в этой книге, а также в ее втором томе.

Защищенное наследование

При наследовании все открытые функции базового класса по умолчанию становятся закрытыми для пользователей производного класса. Однако на практике обычно применяется открытое наследование, чтобы интерфейс базового класса также стал интерфейсом производного класса. Кроме того, наследование можно сделать защищенным, для чего применяется ключевое слово **protected**.

Защищенное наследование подразумевает «ограниченную реализацию» для других классов при полноценном наследовании для дружественных и производных классов. На практике защищенное наследование используется довольно редко, а его поддержка была включена в язык для полноты.

Перегрузка операторов и наследование

Все операторы, кроме оператора присваивания, автоматически наследуются производными классами. Чтобы убедиться в этом, мы воспользуемся заголовочным файлом **Byte.h** из главы 12:

```
//: C14:OperatorInheritance.cpp
// Наследование перегруженных операторов
#include "../C12/Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

class Byte2 : public Byte {
public:
  // Конструкторы не наследуются:
  Byte2(unsigned char bb = 0) : Byte(bb) {}
  // Функция operator= не наследуется, но генерируется автоматически
  // для присваивания на уровне членов класса.
  // Однако при этом генерируется только версия operator=
  // для однотипного присваивания, а для других типов
  // операторы присваивания приходится явно определять в программе:
  Byte2& operator=(const Byte& right) {
    Byte::operator=(right);
    return *this;
  }
  Byte2& operator=(int i) {
    Byte::operator=(i);
    return *this;
  }
};

// Тестовая функция, аналогичная приведенной в C12:ByteTest.cpp:
void k(Byte2& b1, Byte2& b2) {
  b1 = b1 * b2 + b2 % b1;
```

```
  #define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << "; b1 " #OP " b2 produces "; \
    (b1 OP b2).print(out); \
    out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // Оператор присваивания

  // Условные операторы:
  #define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << "; b1 " #OP " b2 produces "; \
    out << (b1 OP b2); \
    out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // Цепочечное присваивание:
  Byte2 b3 = 92;
  b1 = b2 = b3;
}

int main() {
  out << "member functions:" << endl;
  Byte2 b1(47), b2(9);
  k(b1, b2);
} ///:~
```

Тестовая программа почти не отличается от программы из примера C12:ByteTest.cpp, только вместо Byte используется класс Byte2. Так мы проверяем, что все операторы, унаследованные Byte2, сохранили свою работоспособность.

Анализ класса Byte2 показывает, что конструктор должен определяться явно, а из операторов присваивания генерируется автоматически только та версия operator=, которая позволяет присваивать объектам Byte2 значения типа Byte2. Все остальные операторы присваивания вам придется определять самостоятельно.

Множественное наследование

Если класс умеет наследовать от одного базового класса, может показаться, что было бы разумно сделать следующий шаг и разрешить наследование сразу от нескольких базовых классов. Такая возможность действительно существует, но стоит ли использовать ее в проектах? По этому вопросу до сих пор идут незатихающие дискуссии. Спорщики сходятся лишь в одном: множественное наследование — удел опытных программистов, досконально разбирающихся во всех тонкостях языка. Наверное, вы уже понимаете, что в любой ситуации, даже если вам кажется, что

множественное наследование абсолютно необходимо, всегда можно обойтись одиночным наследованием.

На первый взгляд идея множественного наследования кажется достаточно простой: при наследовании в список базовых классов включаются имена нескольких классов, разделенные запятыми. Тем не менее множественное наследование порождает потенциальные проблемы с неоднозначностью имен, поэтому во втором томе книги этой теме будет посвящена целая глава.

Пошаговая разработка

К достоинствам наследования и композиции следует отнести возможность *пошаговой разработки*, то есть включения нового кода в проект с гарантированным сохранением работоспособности существующего кода. Все ошибки локализуются в новом коде. Создавая новые классы на базе готовых работоспособных классов посредством наследования или композиции и добавляя в них новые функции, а также переопределяя существующие функции при наследовании, вы оставляете старый код (который может использоваться еще кем-то) в прежнем состоянии без изменений и ошибок. Если в программе возникает ошибка, вы знаете, что она находится в новом коде, поэтому поиск ошибок выполняется гораздо быстрее и требует меньших усилий, чем при модификации существующего кода.

Просто удивительно, насколько четко разделяются логические функции программы при наличии классов. Чтобы воспользоваться готовыми функциями классов, вам даже не понадобятся их исходные тексты — достаточно заголовков с описанием класса и объектных или библиотечных файлов с откомпилированными функциями (это относится как к наследованию, так и к композиции).

Важно понимать, что разработка программы является пошаговым процессом. Каким бы долгим ни был предварительный анализ, вы все равно не будете знать всех ответов, пока не начнете работу над проектом. Работа пойдет гораздо успешней и быстрее принесет результаты, если вы начнете «выращивать» проект как живое развивающееся существо, а не строить его сразу целиком как бетонный ангар.

Хотя наследование помогает в проведении экспериментов, после стабилизации на какой-то стадии необходимо заново взглянуть на иерархию классов и попробовать свернуть ее в осмысленную структуру. Помните, что в своей основе наследование должно выражать связи вида «Новый класс является *подтипом* старого класса». В программировании следует ориентироваться не на манипуляции с битами, а на создание и работу с объектами разных типов для представления модели в контексте пространства задачи.

Повышающее приведение типа

Ранее в этой главе было показано, что объект класса, производного от ifstream, обладает всеми характеристиками и особенностями поведения объекта ifstream. В программе FName2.cpp для объекта FName2 можно было вызывать любые функции класса ifstream.

И все же самый важный смысл наследования заключается не в том, что оно обеспечивает доступ к функциям базового класса. Наследование выражает связь

между новым и базовым классами, и эта связь выражается так: «Новый класс является подтипом существующего класса».

Не стоит полагать, что эта формулировка всего лишь является хитроумным объяснением наследования, — она напрямую поддерживается компилятором. В качестве примера рассмотрим базовый класс Instrument, представляющий музыкальный инструмент, и производный класс Wind (духовой инструмент). Поскольку наследование означает, что все функции базового класса доступны в производном классе, любые сообщения, предназначенные для базового класса, также могут отправляться производному классу. Если в классе Instrument имеется функция play(), то эта функция будет присутствовать и в классе Wind. Это позволяет нам рассматривать объект Wind как частный случай объекта Instrument. Следующий пример показывает, как эта интерпретация поддерживается компилятором:

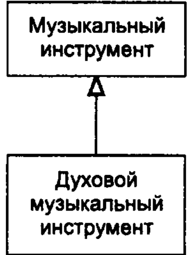
```
//: C14:Instrument.cpp
// Наследование и повышающее приведение типа
enum note { middleC, Csharp, Cflat }; // И т. д.

class Instrument {
public:
  void play(note) const {}
};

// Духовые музыкальные инструменты (Wind) являются музыкальными
// инструментами (Instrument), поскольку обладают тем же интерфейсом:
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Повышающее приведение типа
} ///:~
```

В этом примере наибольший интерес представляет функция tune(), которая должна получать ссылку на Instrument. Тем не менее в функции main() ей передается ссылка на объект Wind. Если вспомнить, как строго C++ относится к проверке типов, происходящее сначала выглядит несколько странно — вместо положенного типа функция спокойно принимает другой тип. Но все становится на свои места, когда вы понимаете, что объект Wind также является объектом Instrument, и все функции Instrument, которые могут вызываться из tune(), заведомо присутствуют в Wind (это гарантируется наследованием). Код, содержащийся в tune(), работает с Instrument и любыми классами, производными от Instrument. Процедура приведения ссылки или указателя на Wind к ссылке или указателю на Instrument называется *повышающим приведением типа* (upcasting).

Понятие повышающего приведения типа

Происхождение термина «повышающее приведение типа» объясняется историческими причинами. Традиционно на диаграммах наследования базовые классы изображались в верхней части листа, а линии наследования направлялись сверху вниз

(конечно, вы можете рисовать диаграммы так, как вам удобнее). Таким образом, для Instrument.cpp диаграмма наследования выглядела бы так:



Переход от производного класса к базовому означает перемещение *вверх* на диаграмме наследования, поэтому он обычно называется повышающим приведением типа. Повышающее приведение типа всегда безопасно, потому что вы переходите от специализированного типа к более общему, — это может сопровождаться только тем, что некоторые функции класса становятся недоступными, новые функции не появляются. Именно поэтому компилятор разрешает повышающее приведение типа без явного указания на это и без какой-либо специальной нотации.

Повышающее приведение типа и копирующий конструктор

Если разрешить компилятору генерировать копирующий конструктор для производного класса, он автоматически вызовет копирующий конструктор базового класса, а затем — копирующие конструкторы всех внутренних объектов (или выполнит поразрядное копирование для встроенных типов):

```
//: C14:CopyConstructor.cpp
// Правильная разработка копирующего конструктора
#include <iostream>
using namespace std;

class Parent {
  int i;
public:
  Parent(int ii) : i(ii) {
    cout << "Parent(int ii)\n";
  }
  Parent(const Parent& b) : i(b.i) {
    cout << "Parent(const Parent&)\n";
  }
  Parent() : i(0) { cout << "Parent()\n"; }
  friend ostream&
    operator<<(ostream& os, const Parent& b) {
    return os << "Parent: " << b.i << endl;
  }
};

class Member {
  int i;
public:
```

```
  Member(int ii) : i(ii) {
    cout << "Member(int ii)\n";
  }
  Member(const Member& m) : i(m.i) {
    cout << "Member(const Member&)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Member& m) {
    return os << "Member: " << m.i << endl;
  }
};

class Child : public Parent {
  int i;
  Member m;
public:
  Child(int ii) : Parent(ii), i(ii), m(ii) {
    cout << "Child(int ii)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Child& c){
    return os << (Parent&)c << c.m
              << "Child: " << c.i << endl;
  }
};

int main() {
  Child c(2);
  cout << "calling copy-constructor: " << endl;
  Child c2 = c; // Вызывает копирующий конструктор
  cout << "values in c2:\n" << c2;
} ///:~
```

Оператор **<<** в классе **Child** интересен для нас тем, как в нем вызывается функция **operator<<** для подобъекта **Parent**: для этого объект **Child** приводится к **Parent&** (приведение к типу *объекта* базового класса вместо ссылки на объект обычно приводит к нежелательным результатам):

```
return os << (Parent&)c << c.m
```

Поскольку компилятор далее воспринимает объект как **Parent**, он вызывает версию **operator<<** для класса **Parent**.

Из листинга видно, что в классе **Child** отсутствует явно определяемый копирующий конструктор. Компилятор генерирует копирующий конструктор, так как он входит в число четырех автоматически генерируемых функций наряду с конструктором по умолчанию (если в классе не определены никакие конструкторы), функцией **operator=** и деструктором; для этого вызываются копирующие конструкторы **Parent** и **Member**. В этом можно убедиться по результатам выполнения программы:

```
Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent(const Parent&)
Member(const Member&)
values in c2:
Parent: 2
Member: 2
Child: 2
```

Но давайте предположим, что вы попытались написать собственный копирующий конструктор для Child и допустили небольшую ошибку:

```
Child(const Child& c) : i(c.i), m(c.m) {}
```

В этом случае для базового подобъекта Child будет автоматически вызван ***конструктор по умолчанию***, потому что именно так поступает компилятор, если у него не остается выбора (вспомните, что для каждого объекта всегда должен быть вызван ***какой-нибудь*** конструктор независимо от того, является он подобъектом другого класса или нет). В этом случае результат будет таким:

```
Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent()
Member(const Member&)
values in c2:
Parent: 0
Member: 2
Child: 2
```

Вероятно, такой результат окажется несколько неожиданным, потому что обычно подобъект базового класса должен копироваться из существующего объекта в новый объект в процессе конструирования копии.

Для решения этой проблемы вы должны помнить об обязательном вызове копирующего конструктора базового класса (как это делает компилятор) при написании собственного копирующего конструктора. На первый взгляд это выглядит несколько странно, но в действительности перед нами лишь еще один пример повышающего приведения типа:

```
Child(const Child& c)
  : Parent(c), i(c.i), m(c.m) {
  cout << "Child(Child&)\n";
}
```

Странно выглядит вызов копирующего конструктора Parent — Parent(c). Что это значит? Что объект Child передается конструктору Parent? Но Child наследует от Parent, поэтому ссылка на Child *является* ссылкой на Parent. Вызов копирующего конструктора базового класса повышает ссылку на Child до ссылки на Parent и использует ее для конструирования копии. Когда вы начнете писать собственные копирующие конструкторы, в них почти всегда следует делать то же самое.

Снова о композиции и наследовании

Один из самых наглядных критериев выбора между композицией и наследованием состоит в следующем: спросите себя, понадобится ли вам когда-нибудь выполнять повышающее приведение типа от нового класса. Ранее в этой главе класс Stack специализировался с применением наследования. Однако весьма вероятно, что объекты StringStack будут использоваться только как контейнеры для string и никогда не будут повышаться до базового класса, поэтому композиция в данном случае более уместна:

```
//: C14:InheritStack2.cpp
// Композиция и наследование
#include "../C09/Stack4.h"
```

```
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack {
  Stack stack; // Внедрение объекта вместо наследования
public:
  void push(string* str) {
    stack.push(str);
  }
  string* peek() const {
    return (string*)stack.peek();
  }
  string* pop() {
    return (string*)stack.pop();
  }
};

int main() {
  ifstream in("InheritStack2.cpp");
  assure(in, "InheritStack2.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) // Без приведения типа!
    cout << *s << endl;
} ///:~
```

Файл очень похож на InheritStack.cpp, не считая того, что объект Stack внедрен в StringStack, а для внутреннего объекта вызываются функции класса. Такое решение не требует дополнительных затрат памяти или времени, поскольку подобъект занимает точно такой же объем памяти, а вся дополнительная проверка типов осуществляется на стадии компиляции.

Также можно воспользоваться закрытым наследованием для представления «ограниченной реализации». Хотя решение получается более сложным и запутанным, его тоже следует признать адекватным. Однако в некоторых ситуациях выбор между этими двумя решениями оказывается принципиальным, например, если возникает потребность во множественном наследовании. Если вам удастся спроектировать архитектуру, при которой наследование заменяется композицией, возможно, вам удастся обойтись без множественного наследования.

Повышающее приведение типа указателей и ссылок

В примере Instrument.cpp повышающее приведение типа происходит при вызове функции — ссылка на объект Wind превращается в ссылку на Instrument внутри функции. Повышающее приведение типа также может выполняться при обычном присваивании указателю или ссылке:

```
Wind w;
Instrument* ip = &w; // Повышающее приведение типа
Instrument* ir = w; // Повышающее приведение типа
```

Как и при вызове функций, ни в одном из этих случаев явного приведения типа не требуется.

Проблемы

Конечно, любое повышающее приведение типа приводит к потере информации о типе объекта. Например, после выполнения следующего фрагмента компилятор сможет интерпретировать ip только как указатель на Instrument и ничего более:

```
Wind w;
Instrument* ip = &w;
```

Иначе говоря, компилятор не знает, что ip *на самом деле* указывает на объект Wind. Следовательно, при вызове функции play(), как показано ниже, компилятор знает лишь то, что он вызывает play() через указатель на Instrument, и получит версию базового класса Instrument::play() вместо предполагаемой функции Wind::play():

```
ip->play(middleC);
```

Возникает весьма серьезная проблема. В главе 15 вы узнаете, как она решается при помощи третьего краеугольного камня объектно-ориентированного программирования — полиморфизма, реализованного в C++ при помощи *виртуальных* функций.

Итоги

Наследование и композиция позволяют создавать новые типы на базе существующих типов; оба механизма внедряют в новые типы подобъекты существующих типов. Тем не менее они применяются в разных ситуациях: композиция обычно обеспечивает многократное использование существующих типов как части базовой реализации нового типа, а наследование позволяет интерпретировать новый тип как тип базового класса (эквивалентность типов гарантирует эквивалентность интерфейсов). Поскольку базовый класс содержит весь интерфейс базового класса, становится возможным *повышающее приведение* к базовому типу. Как будет показано в главе 15, повышающее приведение типа чрезвычайно важно для полиморфизма.

Хотя многократное использование кода посредством композиции и наследования способствует ускоренной разработке проектов, обычно стоит перепроектировать иерархию классов до того, как от нее будут зависеть другие программисты. Ваша цель — такая иерархия, в которой каждый класс обладает конкретным предназначением и все классы не слишком велики (класс предлагает столько функциональности, что с ним неудобно работать) и не раздражающе малы (без добавления новой функциональности класс бесполезен).

Упражнения

1. Измените класс из примера Car.cpp так, чтобы он наследовал от класса Vehicle. Включите в Vehicle какие-нибудь функции и добавьте конструктор, отличный от конструктора по умолчанию; этот конструктор должен вызываться в конструкторе Car.

2. Создайте два класса A и B с конструкторами по умолчанию, которые выводят сообщения о своем вызове. Создайте класс C, производный от A; включите в него внутренний объект класса B, но не определяйте конструктор для C. Создайте объект класса C и проследите за результатами.
3. Создайте трехуровневую иерархию классов, содержащих конструкторы по умолчанию и деструкторы; и те и другие должны выводить сообщения в cout. Убедитесь в том, что для объекта, находящегося на последней ступени иерархии, автоматически вызываются все три конструктора и деструктора. Объясните порядок вызовов.
4. Добавьте в пример Combined.cpp новый уровень наследования и новый внутренний объект. Включите код, который бы демонстрировал порядок вызова конструкторов и деструкторов.
5. Включите в пример Combined.cpp класс D, производный от B; включите в него внутренний объект класса C. Добавьте код вывода сообщений при вызове конструкторов и деструкторов.
6. Включите в пример Order.cpp дополнительный уровень наследования Derived3 с внутренними объектами классов Member4 и Member5. Проанализируйте выходные данные программы.
7. В примере NameHiding.cpp убедитесь в том, что в классах Derived2, Derived3 и Derived4 недоступна ни одна из версий f() базового класса.
8. Измените пример NameHiding.cpp, включив в класс Base три перегруженные функции h(). Покажите, что переопределение любой из этих функций в производном классе скрывает остальные версии.
9. Создайте класс StringVector, производный от vector<void*>. Переопределите функции push_back() и operator[] так, чтобы они принимали и возвращали string*. Что произойдет при попытке вызова push_back() для void*?
10. Создайте класс, содержащий переменную типа long. Выполните инициализацию переменной в конструкторе, используя синтаксис вызова псевдоконструктора.
11. Создайте класс с именем Asteroid. Путем наследования определите специализацию класса PStash из главы 13 (PStash.h и PStash.cpp), которая бы получала и возвращала указатели на Asteroid. Внесите изменения в тестовую программу PStashTest.cpp и протестируйте класс. Затем включите объект PStash с применением композиции.
12. Повторите упражнение 11, используя класс vector вместо PStash.
13. В примере SynthesizedFunctions.cpp включите в класс Chess конструктор по умолчанию, копирующий конструктор и оператор присваивания. Покажите, что эти функции работают правильно.
14. Создайте два класса Traveler и Pager, не содержащие конструкторов по умолчанию. В каждом из классов должен быть определен конструктор с аргументом типа string, который просто копируется во внутреннюю переменную string. Напишите для каждого класса копирующий конструктор и оператор присваивания. Создайте класс BusinessTraveler, наследующий от Traveler, и включите

в него внутренний объект типа Pager. Напишите для BusinessTraveler конструктор по умолчанию, конструктор с аргументом типа string, копирующий конструктор и оператор присваивания.

15. Создайте класс с двумя статическими функциями. Объявите класс, производный от него, и переопределите одну из функций. Покажите, что другая функция скрывается в производном классе.
16. Найдите информацию о других функциях класса ifstream. В примере FName2.cpp опробуйте их на объекте file.
17. Воспользуйтесь закрытым и защищенным наследованием для создания двух классов, производных от одного базового класса. Попробуйте выполнить повышающее приведение типа объектов производного класса к базовому классу. Объясните, что при этом происходит.
18. В примере Protected.cpp включите в класс Derived новую функцию, которая вызывает защищенную функцию read() класса Base.
19. Измените пример Protected.cpp так, чтобы класс Derived создавался защищенным наследованием. Выясните, можно ли вызвать value() для объекта Derived.
20. Создайте класс SpaceShip, содержащий функцию fly(). Создайте класс Shuttle, производный от SpaceShip, включите в него функцию land(). Создайте объект Shuttle, выполните повышающее приведение к типу SpaceShip через указатель или ссылку, и попытайтесь вызвать метод land(). Объясните результат.
21. Измените пример Instrument.cpp и включите в класс Instrument функцию prepare(). Вызовите prepare() из tune().
22. Измените пример Instrument.cpp так, чтобы функция play() выводила сообщение в cout. Переопределите play() в классе Wind так, чтобы функция выводила другое сообщение. Запустите программу и объясните, почему происходящее нежелательно. Вставьте ключевое слово virtual (которое будет описано в главе 15) перед объявлением play() в классе Instrument и проанализируйте изменения в поведении программы.
23. В примере CopyConstructor.cpp создайте новый класс, производный от Child, и включите в него переменную Member m. Напишите конструктор, копирующий конструктор, а также функции operator= и operator<< для потоков вывода. Протестируйте класс в main().
24. В примере CopyContructor.cpp добавьте в класс Child свой копирующий конструктор, в котором *не вызывается* копирующий конструктор базового класса. Объясните происходящее. Исправьте ошибку, включив вызов копирующего конструктора базового класса в список инициализирующих значений копирующего конструктора Child.
25. Измените пример InheritStack2.cpp так, чтобы вместо Stack в нем использовался класс vector<string>.
26. Создайте класс Rock, в котором определены конструктор по умолчанию, копирующий конструктор, оператор присваивания и деструктор; каждая функция должна выводить в cout сообщение о своем вызове. В функции main()

создайте класс vector<Rock> (вектор, содержащий объекты Rock) и занесите в него несколько объектов Rock. Запустите программу и объясните полученные результаты. Проверьте, вызываются ли деструкторы для объектов Rock в vector. Теперь повторите упражнение для vector<Rock*>. Можно ли создать vector<Rock&>?

27. В этом упражнении используется архитектурная идиома, в которой задействован так называемый *заместитель* (proxy). Начните с создания базового класса Subject и определите три функции: f(), g() и h(). Создайте производные от Subject классы Proxy, а также классы Implementation1 и Implementation2. Класс Proxy содержит указатель на Subject, а все функции класса Proxy должны перенаправлять свои вызовы через указатель на Subject. Конструктор Proxy получает указатель на Subject, который запоминается в переменной класса. Создайте в функции main() два объекта Proxy, использующие две разные реализации. Измените класс Proxy так, чтобы он позволял динамически переключаться с одной реализации на другую.
28. Измените пример ArrayOperatorNew.cpp из главы 13 так, чтобы при наследовании от Widget выделение памяти все равно работало правильно. Объясните, почему наследование в примере Framis.cpp из главы 13 работало неправильно.
29. Измените пример Framis.cpp из главы 13. Создайте новый класс, производный от Framis, и определите в нем новые версии операторов new и delete. Продемонстрируйте их работу.

Полиморфизм и виртуальные функции

Третьим важнейшим аспектом объектно-ориентированных языков программирования (после абстрактного представления данных и наследования) является *полиморфизм*, реализованный в C++ при помощи виртуальных функций.

Полиморфизм выводит принцип отделения интерфейса от реализации на качественно новый уровень: он, если можно так выразиться, разделяет «что» и «как». Полиморфизм улучшает структуру программы и делает ее более наглядной, а также помогает создавать *расширяемые* программы, возможности которых могут наращиваться по мере необходимости.

Механизм инкапсуляции позволяет создавать новые типы данных за счет объединения характеристик и аспектов поведения. Механизм управления доступом позволяет отделить интерфейс от реализации, подробности которой скрываются в закрытой части класса. Подобная механическая организация выглядит вполне логичной для программиста с опытом работы на процедурных языках. С другой стороны, виртуальные функции подходят к проблеме логического разделения с точки зрения *типов*. В главе 14 было показано, что благодаря наследованию объект можно интерпретировать как объект своего *или* базового типа. Эта возможность чрезвычайно важна, поскольку она позволяет интерпретировать разные типы, производные от общего базового типа, как относящиеся к одному типу. Таким образом, вы можете написать фрагмент программы, который будет работать с любым из множества типов. Виртуальные функции позволяют одному типу выразить свое отличие от другого сходного типа — при условии, что они оба наследуют от одного базового типа. Отличие проявляется в поведении функций, вызываемых через базовый класс.

В этой главе мы рассмотрим виртуальные функции. Изучение начнется с простейших примеров, из которых для наглядности убрано все, кроме «виртуальности».

Эволюция программирования на C++

Программисты C обычно осваивают язык C++ в три этапа. На первом этапе они рассматривают его как «улучшенный язык C» — C++ требует объявлять все функ-

ции перед использованием и гораздо строже следит за применением переменных. Довольно часто ошибки в программах C обнаруживаются путем простой компиляции программ на C++.

На втором этапе они имеют дело с «объектно-базированным» языком C++. Речь идет об очевидных преимуществах: организации кода, при которой структуры данных группируются с функциями, работающими с этими данными; удобстве конструкторов и деструкторов, а также простейших применениях наследования.

Большинство программистов с опытом работы на C быстро осознают пользу объектно-базированного подхода, поскольку именно эти задачи обычно приходится решать при разработке библиотек. В C++ компилятор помогает в их решении.

Однако многие «застревают» на объектно-базированном уровне. До этого уровня легко добраться, причем программист обретает массу полезных инструментов без особых умственных усилий. Он создает классы и объекты, организует передачу сообщений объектам, и на первый взгляд все идет просто замечательно.

Но не стоит заблуждаться. Если остановиться на этом этапе, вы упустите самое главное в языке — «настоящее» объектно-ориентированное программирование. Переход к этому этапу можно сделать только при помощи виртуальных функций.

Виртуальные функции выводят концепцию типа за пределы простой инкапсуляции кода в структурах с ограничением доступа. Несомненно, именно эту концепцию будет труднее всего усвоить новичкам C++. С другой стороны, она становится решающим моментом в понимании сути объектно-ориентированного программирования. Если вы не используете виртуальные функции, значит, вы еще не доросли до этого.

Так как виртуальные функции тесно связаны с концепцией типа, а тип занимает центральное место в объектно-ориентированном программировании, виртуальные функции не имеют аналогов в традиционных процедурных языках. Возможности процедурных языков можно понять на алгоритмическом уровне, но виртуальные функции представимы только с точки зрения архитектуры.

Повышающее приведение типа

В главе 14 было показано, что объект может интерпретироваться как относящийся к своему или к базовому типу. Кроме того, с объектом можно работать через адрес базового типа. Получение адреса объекта (в виде указателя или ссылки) и его дальнейшая интерпретация как адреса объекта базового типа называется ***повышающим приведением типа*** (как уже упоминалось, этот термин появился из-за того, что базовые классы традиционно изображались в верхней части диаграмм иерархии наследования).

Также была описана проблема, воплощенная в следующем примере:

```
//: C15:Instrument2.cpp
// Наследование и повышающее приведение типа
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // И т. д.

class Instrument {
public:
  void play(note) const {
```

```
    cout << "Instrument::play" << endl;
  }
};

// Духовые музыкальные инструменты (Wind) являются музыкальными
// инструментами (Instrument), поскольку обладают тем же интерфейсом:
class Wind : public Instrument {
public:
  // Переопределение интерфейсной функции:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Повышающее приведение типа
} ///:~
```

Функция tune() получает (по ссылке) объект Instrument, но она также беспрекословно примет любой объект класса, производного от Instrument. Так, внутри функции main() в функцию tune() передается объект Wind, не требуя приведения типа. Это вполне логично, весь интерфейс Instrument заведомо присутствует в Wind, потому что Wind открыто наследует от Instrument. Повышающее приведение типа Wind к Instrument «сужает» интерфейс производного класса, но никогда не нарушает границ полного интерфейса Instrument.

Эти рассуждения остаются истинными при работе с указателями. Единственное отличие состоит в том, что пользователь должен явно получить адрес объекта, передаваемого функции.

Проблема

При запуске программы Instrument2.cpp неожиданно выявляется проблема: программа выводит строку Instrument::play. Конечно, это не тот результат, на который мы рассчитывали, поскольку известно, что объект в действительности относится к классу Wind, а не Instrument. Предполагалось, что программа выведет Wind::play. Итак, для любого объекта класса, производного от Instrument, в любом случае должна вызваться его версия функции play().

Учитывая отношение к функциям в языке C, поведение программы Instrument2.cpp вполне объяснимо. Но чтобы лучше понять происходящее, необходимо разобраться в концепции *связывания*.

Связывание вызовов функций

Сопоставление вызова функции с телом функции называется *связыванием*. Если связывание выполняется до запуска программы (то есть компилятором и компоновщиком), оно называется *ранним связыванием*. Возможно, этот термин вам

раньше не встречался, потому что в процедурных языках нет других вариантов: компиляторы С поддерживают только один вид вызова функций, а именно раннее связывание.

Проблемы в приведенной выше программе возникли именно из-за раннего связывания, поскольку компилятор не может определить правильный адрес вызываемой функции, располагая только адресом Instrument.

На помощь приходит *позднее связывание*; этот термин означает, что связывание производится во время выполнения программы в зависимости от фактического типа объекта. Позднее связывание также иногда называют *динамическим* связыванием. Если язык программирования поддерживает позднее связывание, в нем также должен присутствовать некий механизм, который бы определял тип объекта во время выполнения и вызывал нужную функцию класса. В компилируемых языках компилятор не знает фактического типа объекта, но вставляет в программу фрагмент кода для определения и вызова правильной функции. Реализация механизмов позднего связывания зависит от языка, но можно предположить, что в объекте сохраняется информация о его типе. Вскоре вы увидите, как этот механизм работает в C++.

Виртуальные функции

Чтобы в C++ для функции выполнялось позднее связывание, она должна быть объявлена *виртуальной*, то есть ее объявление в базовом классе должно содержать ключевое слово virtual. Позднее связывание работает только с виртуальными функциями и только при использовании адреса базового класса, содержащего виртуальные функции (хотя функции также могут определяться в одном из более ранних базовых классов).

Итак, для создания виртуальной функции достаточно включить в ее объявление ключевое слово virtual. Ключевое слово virtual обязательно только для объявления функции, но не для ее определения. Если функция объявляется виртуальной в базовом классе, она также становится виртуальной во всех производных классах. Повторное определение виртуальной функции в производном классе обычно называется *переопределением*.

Обратите внимание: ключевое слово virtual обязательно только при объявлении функции в базовом классе. Все функции производных классов, по сигнатуре соответствующие объявлению в базовом классе, будут вызываться с помощью механизма виртуальных функций. Ключевое слово virtual *может* использоваться в объявлениях производных классов, вреда от него не будет, однако оно является избыточным и может вызвать недоразумения.

Чтобы добиться желаемого поведения от программы Instrument2.cpp, просто включите ключевое слово virtual в базовый класс перед объявлением функции play():

```
//: C15:Instrument3.cpp
// Позднее связывание с ключевым словом virtual
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // И т. д.

class Instrument {
public:
```

```
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Духовые музыкальные инструменты (Wind) являются музыкальными
// инструментами (Instrument), поскольку обладают тем же интерфейсом:
class Wind : public Instrument {
public:
  // Переопределение интерфейсной функции:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Повышающее приведение типа
} ///:~
```

Если не считать появления ключевого слова virtual, пример идентичен примеру Instrument2.cpp, но поведение программы принципиально изменилось: в новой версии выводится строка Wind::play.

Расширяемость

После определения в базовом классе виртуальной функции play() в программу можно включить сколько угодно новых типов, не изменяя функции tune(). В хорошо спроектированной объектно-ориентированной программе многие функции следуют примеру tune() и взаимодействуют только с интерфейсом базового класса. Такие программы хорошо *расширяются*, потому что программист может добавлять в них новые возможности посредством наследования новых типов данных от общего базового класса. При этом функции, работающие через интерфейс базового класса, изменять вообще не придется.

Ниже приведен еще один пример с музыкальными инструментами, в который были добавлены новые виртуальные функции и новые классы; все они правильно работают с прежней функций tune(), оставшейся неизменной:

```
//: C15:Instrument4.cpp
// Расширяемость программ в ООП
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // И т. д.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
```

```
  }
  // Предполагается, что эта функция будет изменять объект
  // ("настройка музыкального инструмента"):
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Функция осталась неизменной:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// Новая функция:
void f(Instrument& i) { i.adjust(1); }

// Повышающее приведение типа при инициализации массива:
```

```
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Как видите, ниже класса Wind добавился новый уровень наследования, но механизм виртуальных функций работает правильно независимо от количества уровней. Функция adjust() *не переопределяется* в классах Brass и Woodwind. В этом случае автоматически используется «ближайшее» определение в иерархии наследования — компилятор гарантирует, что у виртуальной функции всегда имеется *хоть какое-то* определение и никогда не возникнет ситуации, при которой вызов не будет связан с телом функции (что привело бы к катастрофическим результатам).

Массив A[ ] содержит указатели на базовый класс Instrument, поэтому в процессе инициализации массива происходит повышающее приведение типа. Этот массив и функция f() будут использоваться в дальнейших объяснениях.

При вызове tune() для разных типов объектов выполняется повышающее приведение типа, но в результате всегда происходит именно то, что нужно. Ситуацию можно описать так: «Мы отправляем объекту сообщение, и пусть объект сам решает, как с ним поступить». Анализируя проект, попытайтесь взглянуть на него через призму виртуальных функций: на каком уровне должны располагаться базовые классы и в каком направлении пойдет возможное расширение программы? Но даже если интерфейсы базовых классов и виртуальных функций не удастся выявить на начальной стадии разработки программы, они довольно часто обнаруживаются позднее — и даже гораздо позднее, когда вам приходится расширять или обеспечивать иное сопровождение программы. Не стоит считать это признаком ошибки анализа или проектирования; возможно, ранее вы просто не располагали (или не могли располагать) всей необходимой информацией. Благодаря жесткому обособлению классов в C++ это не создает особых проблем, поскольку изменения, вносимые в одной части системы, обычно не распространяются в другие части системы, как это происходит в C.

Позднее связывание в C++

Итак, как же происходит позднее связывание? Всю работу незаметно выполняет компилятор, который запускает механизм позднего связывания по требованию

программиста (то есть при создании виртуальных функций). Знание принципов работы механизма виртуальных функций в C++ часто оказывается полезным для программистов, поэтому в этом разделе мы подробно рассмотрим, как же компилятор реализует этот механизм.

Ключевое слово virtual сообщает компилятору, что он не должен выполнять раннее связывание. Вместо этого компилятор активизирует все механизмы, необходимые для позднего связывания. Это означает, что при вызове play() для объекта Brass через *адрес базового класса* Instrument будет вызвана правильная функция.

Для этого типичный компилятор[1] создает таблицу (часто называемую таблицей виртуальных функций, или VTABLE) для каждого класса, содержащего виртуальные функции. Компилятор помещает адреса виртуальных функций данного класса в таблицу виртуальных функций. В каждом классе, содержащем виртуальные функции, скрыто сохраняется указатель VPTR, который ссылается на таблицу виртуальных функций объекта. При вызове виртуальной функции через указатель на базовый класс (то есть при полиморфном вызове) компилятор незаметно вставляет в программу фрагмент кода, который получает указатель VPTR и ищет адрес функции в таблице виртуальных функций. В результате вызывается правильная функция, то есть происходит позднее связывание.

Все эти операции — создание таблицы виртуальных функций для каждого класса, инициализация указателя VPTR, вставка кода вызова виртуальной функции — выполняются автоматически, поэтому вам вообще не придется о них беспокоиться. При использовании виртуальных функций для объекта будет вызвана правильная функция даже в том случае, если компилятору неизвестен фактический тип этого объекта.

Далее этот процесс рассматривается более подробно.

Хранение информации о типе

Ни один из созданных нами классов не содержал явно заданной информации о типе. Тем не менее предыдущие примеры и простая логика подсказывают, что в объектах должна храниться какая-то информация о типе, иначе тип объекта не удалось бы определить во время выполнения программы. Это действительно так, но информация о типе хранится в скрытом виде. Следующий пример доказывает этот факт: мы сравним размеры одинаковых классов, различающихся только наличием виртуальных функций:

```
//: C15:Sizes.cpp
// Сравнение размеров объектов с виртуальными функциями и без них
#include <iostream>
using namespace std;

class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
```

[1] Компиляторы могут реализовать механизм виртуального вызова по своему усмотрению, но описанный здесь способ считается практически общепринятым.

```
};

class OneVirtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "OneVirtual: "
       << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: "
       << sizeof(TwoVirtuals) << endl;
} ///:~
```

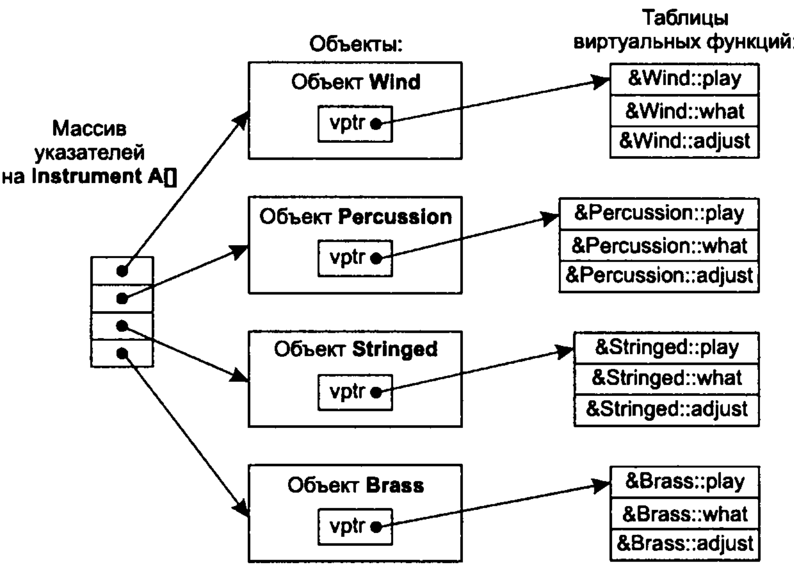
Без виртуальных функций размер объекта точно соответствует нашим предположениям: он равен размеру одного[1] значения int. При одной виртуальной функции размер объекта OneVirtual равен размеру NoVirtual с прибавлением размера указателя на void. Оказывается, компилятор вставляет один указатель VPTR в каждую структуру, содержащую *одну и более* виртуальных функций. Так, объекты OneVirtual и TwoVirtual имеют одинаковые размеры. Это объясняется тем, что VPTR указывает на таблицу адресов функций. Реально нужна только одна таблица, содержащая адреса всех виртуальных функций.

В нашем примере классы должны содержать хотя бы одну переменную. Классы, в которых полностью отсутствуют переменные, автоматически приводятся компилятором C++ к ненулевому размеру, потому что каждый объект должен обладать уникальным адресом. Представьте процесс индексирования в массиве объектов нулевого размера, и все станет ясно. В объекты, размер которых должен быть равен нулю, включается фиктивная переменная. Но если из-за присутствия ключевого слова virtual в объекте сохраняется информация о типе, она занимает место этой фиктивной переменной. Попробуйте закомментировать в классах из предыдущего примера строку int a и убедитесь в этом сами.

Механизм вызова виртуальных функций

Чтобы вы досконально поняли, как вызываются виртуальные функции, будет полезно разобраться в операциях, выполняемых «за кулисами» компилятором. На следующем рисунке изображен массив указателей A[ ] из примера Instrument4.cpp:

[1] У некоторых компиляторов возникают специфические проблемы с размером, но такие случаи относительно редки.



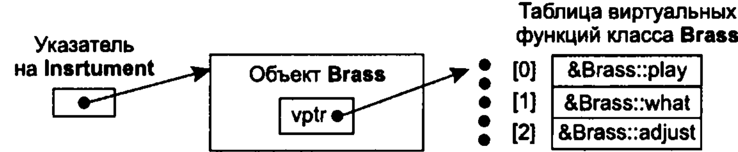
Массив указателей на Instrument не содержит информации о типе; каждый указатель ссылается на объект типа Instrument. Все объекты Wind, Percussion, Stringed и Brass относятся к этой категории, поскольку эти классы являются производными от Instrument (а следовательно, обладают одинаковым интерфейсом с Instrument и реагируют на те же сообщения), поэтому адреса этих объектов тоже могут храниться в массиве. Тем не менее компилятор не знает, что объекты представляют собой нечто большее, чем базовые объекты Instrument; в обычном случае он просто вызвал бы версии этих функций из базового класса. Но в данной ситуации все функции объявлялись с ключевым словом virtual, поэтому происходит нечто иное.

Каждый раз, когда вы определяете класс с виртуальными функциями или создаете класс, производный от него, компилятор создает для этого класса уникальную таблицу виртуальных функций (в правой части рисунка). В этой таблице размещаются адреса всех функций, объявленных виртуальными в этом или базовом классе. Если функция, объявленная виртуальной в базовом классе, не переопределялась, то компилятор использует в производном классе адрес версии базового класса (см. запись функции adjust в таблице Brass). Затем в класс включается указатель VPTR, существование которого мы обнаружили с помощью программы Sizes.cpp. При подобном простом наследовании для каждого объекта создается только один указатель VPTR, инициализируемый начальным адресом соответствующей таблицы виртуальных функций (как будет показано далее, это происходит в конструкторе).

После инициализации VPTR адресом правильной таблицы виртуальных функций объект фактически «узнает», к какому типу он относится. Однако это «самопознание» остается бесполезным вплоть до точки вызова виртуальной функции.

При вызове виртуальной функции через адрес базового класса (в ситуации, когда компилятор не обладает всей информацией, необходимой для раннего связывания) происходит нечто особенное. Вместо простой ассемблерной команды CALL с заданным адресом, реализующей типичный вызов функции, компилятор генери-

рует особый фрагмент кода. Ниже показано, как выглядит вызов adjust для объекта Brass через указатель на Instrument (то же самое происходит при использовании ссылки на Instrument).



Компилятор начинает с указателя на Instrument, ссылающегося на начальный адрес объекта. Во всех объектах класса Instrument или классов, производных от него, указатель VPTR хранится в одном месте (чаще всего в начале объекта), поэтому компилятор узнает его текущее значение. VPTR указывает на начальный адрес таблицы виртуальных функций. Все адреса функций в таблице виртуальных функций всегда следуют в одинаковом порядке независимо от фактического типа объекта. Функция play() находится на первом месте, функция what() — на втором, а функция adjust() — на третьем. Компилятор знает, что при любом фактическом типе объекта адрес функции adjust() хранится со смещением VPTR+2. Таким образом, вместо вызова функции, хранящейся по абсолютному адресу Instrument::adjust (раннее связывание; ошибка), генерируется код, который фактически означает «вызов функции по адресу, хранящемуся со смещением VPTR+2». Поскольку выборка VPTR и определение адреса функции происходят во время выполнения программы, происходит желаемое позднее связывание. Вы посылаете сообщение объекту, а объект сам решает, что с ним делать.

Внутренние механизмы

Давайте рассмотрим пример ассемблерного кода, сгенерированного при вызове виртуальной функции; так мы узнаем, как же в действительности выполняется позднее связывание. Ниже приведен код, выданный одним из компиляторов для вызова команды i.adjust(1); в функции f(Instrument& i):

```
push   1
push   si
mov    bx, word ptr [si]
call   word ptr [bx+4]
add    sp, 4
```

Аргументы функции в C++ (впрочем, как и в C) заносятся в стек справа налево; этот порядок необходим для поддержки переменных списков аргументов C. Следовательно, первым в стек заносится аргумент 1. Регистр si (в архитектуре процессоров семейства Intel X86) содержит адрес переменной i, который тоже заносится в стек как начальный адрес передаваемого объекта. Вспомните: начальный адрес соответствует значению this, а this автоматически заносится в стек как скрытый аргумент при каждом вызове функции класса, чтобы функция знала, с каким конкретным объектом она должна работать. По этой причине количество значений, заносимых в стек при вызове, всегда на 1 больше количества аргументов (исключение составляют статические функции классов, которым this не передается).

Теперь выполняется собственно вызов виртуальной функции. Сначала необходимо получить указатель VPTR для нахождения таблицы виртуальных функций. Наш компилятор сохраняет VPTR в начале объекта, поэтому адрес VPTR соответствует this. Следующая команда получает VPTR (слово, на которое указывает si, то есть this), и помещает его в регистр bx.

```
mov bx, word ptr [si]
```

Указатель VPTR, хранящийся в bx, указывает на начальный адрес таблицы виртуальных функций, но адрес вызываемой функции хранится в таблице не с нулевым смещением, а со смещением 2 (потому что функция находится на третьем месте в списке). В используемой модели памяти указатель на функцию занимает два байта, поэтому для вычисления адреса нужной функции компилятор увеличивает VPTR на 4. Обратите внимание: смещение является константой, определяемой на стадии компиляции, поэтому важно лишь то, чтобы указатель на функцию в третьей позиции таблицы относился к функции adjust(). К счастью, компилятор берет на себя всю черновую работу и следит за тем, чтобы указатели на функции во всех таблицах виртуальных функций определенной иерархии классов всегда следовали в одинаковом порядке, независимо от порядка их возможного переопределения в производных классах.

После того как адрес нужного указателя на функцию в таблице виртуальных функций вычислен, происходит вызов этой функции. Выборка адреса совмещается с вызовом в команде

```
call word ptr [bx+4]
```

Наконец указатель стека смещается вверх, что приводит к стиранию аргументов, занесенных в стек перед вызовом. В ассемблерном коде, сгенерированном компиляторами C и C++, ответственность за удаление аргументов из стека часто лежит на вызывающей стороне, но это зависит от процессора и реализации компилятора.

Инициализация указателя VPTR

Указатель VPTR определяет поведение виртуальных функций объекта, поэтому очень важно, чтобы он всегда правильно указывал на таблицу виртуальных функций. Виртуальные функции ни при каких условиях не должны вызываться до правильной инициализации VPTR. Конечно, инициализация могла бы гарантированно выполняться в конструкторе, но ни в одном из примеров класса Instrument конструктор не определен.

В этой ситуации очень важная роль отводится созданию конструктора по умолчанию. В примерах класса Instrument компилятор создает конструктор по умолчанию, который не делает ничего, кроме инициализации VPTR. Разумеется, этот конструктор автоматически вызывается для всех объектов Instrument перед тем, как с ними можно будет что-либо сделать, поэтому вызов виртуальных функций заведомо безопасен.

Далее обсуждаются последствия автоматической инициализации VPTR в конструкторе.

Повышающее приведение типа и объекты

Очень важно понимать, что повышающее приведение типа применимо только к адресам. Если компилятор работает с объектом, он знает его фактический тип и по-

этому (в C++) не использует позднее связывание для вызова функций (или, как минимум, ему *не нужно* использовать позднее связывание). По соображениям эффективности многие компиляторы выполняют раннее связывание при вызове виртуальной функции для объекта, потому что им точно известен его тип. Пример:

```
//: C15:Early.cpp
// Раннее связывание и виртуальные функции
#include <iostream>
#include <string>
using namespace std;

class Pet {
public:
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
  string speak() const { return "Bark!"; }
};

int main() {
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Позднее связывание в обоих случаях:
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Раннее связывание (вероятно):
  cout << "p3.speak() = " << p3.speak() << endl;
} ///:~
```

При вызовах p1->speak() и p2.speak() используются адреса, а это означает отсутствие полной информации: p1 и p2 могут представлять адрес объекта Pet *или* объекта класса, производного от Pet, поэтому должен быть задействован механизм виртуального вызова. При вызове p3.speak() такая неоднозначность отсутствует. Компилятору известен точный тип объекта, поэтому вариант с объектом класса, производного от Pet, исключен — объект относится *именно* к классу Pet. Вероятно, в этом случае потребуется раннее связывание. Но если компилятор не захочет напрягаться, он может выполнить позднее связывание — результат будет тем же.

Виртуальные функции и эффективность

К этому моменту у многих возникает вопрос: «Если механизм виртуальных функций так важен и если в каждом конкретном случае он позволяет вызывать „правильную" функцию, почему бы не сделать его обязательным?»

Хороший вопрос. Ответ на него является частью основополагающей философии C++: «Потому, что он недостаточно эффективен». Из приведенного выше листинга на ассемблере видно, что вместо одной команды CALL, передающей управление по абсолютному адресу, используются две более сложные ассемблерные команды, что приводит к дополнительным затратам как памяти, так и времени выполнения.

Разработчики некоторых объектно-ориентированных языков решили, что позднее связывание играет настолько важную роль в объектно-ориентированном программировании, что оно должно выполняться всегда, а пользователям об этом знать даже не обязательно. Такие решения принимаются при проектировании языка, причем во многих конкретных случаях они вполне оправданны[1]. Но C++ происходит от языка C, в котором определяющим фактором является эффективность. Собственно, язык C создавался как замена для ассемблера при написании операционных систем (возможно, именно поэтому операционная система Unix обладает гораздо лучшей переносимостью, чем ее предшественницы). В частности, одной из главных причин для создания C++ стала необходимость в повышении эффективности работы программистов C[2]. А когда программист C впервые сталкивается с C++, он обычно спрашивает: «И как это отразится на размерах и быстродействии моих программ?» Если бы на него приходилось отвечать: «Все хорошо, только каждый вызов функции всегда сопровождается небольшими дополнительными затратами», то многие программисты до сих пор работали бы на C, отказавшись от перехода на C++. Кроме того, при полном переходе на виртуальные функции стала бы невозможной подстановка функций, поскольку виртуальная функция должна обладать адресом для его сохранения в таблице виртуальных функций. По этим причинам виртуальные функции остаются альтернативным решением, а по умолчанию в языке используется невиртуальный вызов, обеспечивающий максимальную скорость работы программ. Страуструп утверждал, что он руководствовался принципом: «Если не пользуешься — не плати».

Впрочем, при проектировании классов не стоит беспокоиться о проблемах эффективности. Если вы собираетесь использовать полиморфизм, объявляйте все функции виртуальными. Функции, которые можно объявить невиртуальными, лучше отыскивать в ходе оптимизации программы (причем максимальный выигрыш часто удается получить в других областях — хороший профайлер обычно справляется с поиском «узких мест» лучше, чем вы со своими догадками).

Считается, что по размерам и быстродействию программ C++ уступает C не более чем на 10 %, но гораздо чаще их характеристики находятся на более или менее одинаковом уровне. Столь высокая эффективность программ по размерам и быстродействию объясняется тем, что на архитектурном уровне программы C++ получаются более быстрыми и компактными, чем программы C.

Абстрактные базовые классы и чисто виртуальные функции

В процессе проектирования часто бывает нужно, чтобы базовый класс *только* определял интерфейс для своих производных классов. Иначе говоря, никто не должен создавать объекты базового класса; последний задействуется только в повышающем приведении типа для использования его интерфейса. С этой целью класс объявляется *абстрактным*, то есть в него включается хотя бы одна *чисто виртуальная* функция. Объявление чисто виртуальной функции должно начинаться

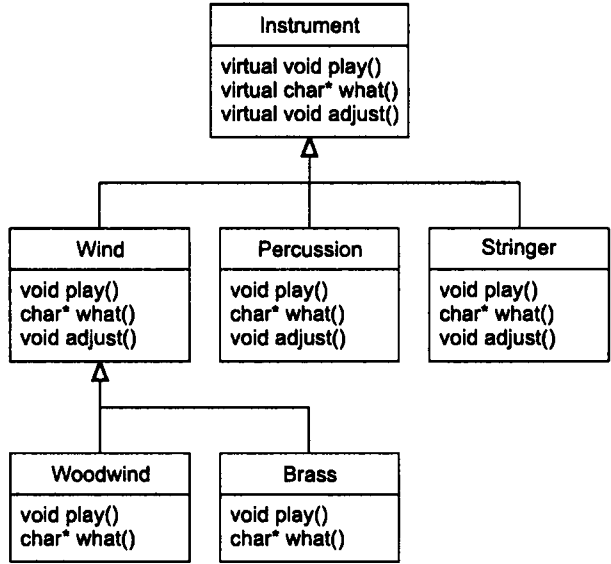
[1] Например, этот подход с большим успехом применяется в Smalltalk, Java и Python.

[2] В лаборатории Bell, где был рожден язык C++, было *очень много* программистов C. Любое даже самое незначительное повышение эффективности их работы экономило компании много миллионов.

с ключевого слова virtual и заканчиваться символами =0. Если кто-нибудь попытается создать объект абстрактного класса, компилятор пресекает эти попытки. Абстрактные классы вынуждают программистов использовать определенную архитектуру в своих программах.

При наследовании от абстрактного класса все чисто виртуальные функции должны быть реализованы, иначе производный класс также становится абстрактным. Определение чисто виртуальной функции позволяет включить в интерфейс функцию класса, не связывая с ней бессмысленный фрагмент кода. В то же время производные классы обязаны предоставить реализацию для чисто виртуальной функции.

Во всех примерах с музыкальными инструментами все функции базового класса Instrument были фиктивными. Вызов любой из этих функций означал бы, что в программе что-то идет не так. Дело в том, что класс Instrument предназначен только для определения обобщенного интерфейса всех классов, производных от него.



Обобщенный интерфейс определяется только для одной цели: чтобы его можно было по-разному выразить в разных производных типах. Интерфейс создает базовую формулировку, определяющую общие черты всех производных классов, — и ничего более. Итак, Instrument является хорошим кандидатом для определения в виде абстрактного класса. Абстрактные классы создаются в ситуации, когда вы просто хотите работать с набором классов через общий интерфейс, но этот общий интерфейс не обязан иметь реализацию (или, по крайней мере, полную реализацию).

Абстрактные классы вроде Instrument воплощают отвлеченные понятия, поэтому создание объектов этого класса почти всегда бессмысленно. Иначе говоря, класс Instrument предназначен только для выражения интерфейса, но не его конкретной реализации. Создавать объекты на этом уровне не имеет смысла, поэтому пользователю лучше запретить делать это. В принципе, можно заставить все вир-

туальные функции Instrument выводить сообщения об ошибках, но это откладывает появление информации об ошибке до стадии выполнения и требует надежного полноценного тестирования со стороны пользователя. Такие проблемы гораздо лучше перехватывать на стадии компиляции.

Синтаксис объявления чисто виртуальной функции выглядит так:

```
virtual void f() = 0;
```

Объявляя функцию подобным образом, мы указываем компилятору, что он должен зарезервировать для функции ячейку в таблице виртуальных функций, но не заносить в нее адрес. Даже если всего одна функция в классе объявлена чисто виртуальной, таблица виртуальных функций остается неполной.

Если таблица виртуальных функций для класса не содержит полной информации, как должен действовать компилятор, когда кто-то пытается создать объект этого класса? Создавать объекты абстрактного класса небезопасно, поэтому компилятор выдает сообщение об ошибке, гарантируя тем самым «чистоту» абстрактного класса. Объявляя свой класс абстрактным, вы предотвращаете потенциальные ошибки при его использовании прикладными программистами.

Ниже приведена программа Instrument4.cpp, измененная для использования чисто виртуальных функций. А поскольку базовый класс не содержит ничего, кроме чисто виртуальных функций, мы назовем его *чисто абстрактным классом*:

```
//: C15:Instrument5.cpp
// Чисто абстрактные базовые классы
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // И т. д.

class Instrument {
public:
  // Чисто виртуальные функции:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  virtual void adjust(int) = 0;
};
// Оставшаяся часть выглядит аналогично ...

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
```

```
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Функция осталась неизменной:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// Новая функция:
void f(Instrument& i) { i.adjust(1); }

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Чисто виртуальные функции полезны прежде всего тем, что они явно сообщают об абстрактности класса и указывают как пользователю, так и компилятору, как этот класс должен использоваться.

Обратите внимание: чисто виртуальные функции предотвращают передачу объекта абстрактного класса функции *по значению*. При этом также предотвращается *расщепление объекта* (см. далее). Объявляя класс абстрактным, вы гарантируете, что повышающее приведение типа к типу этого класса всегда будет осуществляться через ссылки или указатели.

Тот факт, что одна чисто виртуальная функция не позволяет заполнить таблицу виртуальных функций, еще не означает, что тела других функций тоже должны

отсутствовать. В таких случаях часто вызывается версия функции из базового класса, даже если она является виртуальной. Всегда рекомендуется размещать общий код как можно ближе к корневому уровню иерархии. Это не только уменьшает объем кода, но и упрощает распространение изменений.

Определения чисто виртуальных функций

Вы можете предоставить определения чисто виртуальных функций базового класса. При этом компилятор все равно не разрешает создавать объекты абстрактного базового класса, а чтобы создание объектов стало возможным, чисто виртуальные функции должны быть определены в производных классах. Тем не менее может существовать некий общий фрагмент кода, используемый всеми (или некоторыми) определениями производных классов, вместо того чтобы дублировать этот код в каждой функции.

Вот как выглядит определение чисто виртуальной функции:

```
//: C15:PureVirtualDefinitions.cpp
// Определения чисто виртуальных функций в базовом классе
#include <iostream>
using namespace std;

class Pet {
public:
  virtual void speak() const = 0;
  virtual void eat() const = 0;
  // Подстановка определений чисто виртуальных функций запрещена:
  //!  virtual void sleep() const = 0 {}
};

// Можно, функция не является подставляемой
void Pet::eat() const {
  cout << "Pet::eat()" << endl;
}

void Pet::speak() const {
  cout << "Pet::speak()" << endl;
}

class Dog : public Pet {
public:
  // Использование общего кода Pet:
  void speak() const { Pet::speak(); }
  void eat() const { Pet::eat(); }
};

int main() {
  Dog simba;
  simba.speak();
  simba.eat();
} ///:~
```

Ячейка в таблице виртуальных функций класса Pet по-прежнему остается пустой; просто появляется функция, которая может вызываться в производных классах.

Другое преимущество возможности определения чисто виртуальных функций заключается в том, что она позволяет заменить обычные виртуальные функции

чисто виртуальными, не нарушив работы существующего кода (например, это помогает найти классы, в которых виртуальная функция не переопределяется).

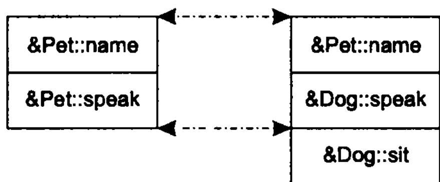
Наследование и таблицы виртуальных функций

Нетрудно представить, что происходит при создании производного класса и переопределении некоторых виртуальных функций. Компилятор создает для нового класса новую таблицу виртуальных функций и вставляет в нее адреса новых функций. Если функция не переопределялась, в соответствующую ячейку таблицы виртуальных функций заносится адрес версии базового класса. Так или иначе, для каждого объекта, который может быть создан (то есть принадлежит к классу, не содержащему чисто виртуальных функций), в таблице виртуальных функций всегда хранится полный набор адресов функций. Тем самым полностью предотвращается вызов функции по несуществующему адресу (что имело бы катастрофические последствия).

Но что произойдет, если новые виртуальные функции добавляются в *производном* классе? Рассмотрим простой пример:

```
//: C15:AddingVirtuals.cpp
// Добавление виртуальных функций при наследовании
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& petName) : pname(petName) {}
  virtual string name() const { return pname; }
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
  string name;
public:
  Dog(const string& petName) : Pet(petName) {}
  // Новая виртуальная функция в классе Dog:
  virtual string sit() const {
    return Pet::name() + " sits";
  }
  string speak() const { // Переопределение
    return Pet::name() + " says 'Bark!'";
  }
};

int main() {
  Pet* p[] = {new Pet("generic"),new Dog("bob")};
  cout << "p[0]->speak() = "
       << p[0]->speak() << endl;
  cout << "p[1]->speak() = "
       << p[1]->speak() << endl;
//! cout << "p[1]->sit() = "
//!      << p[1]->sit() << endl; // Недопустимо
} ///:~
```

Класс Pet содержит две виртуальные функции: speak() и name(). В классе Dog к ним добавляется третья виртуальная функция sit(), а также переопределяется функция speak(). Следующая диаграмма поможет вам лучше разобраться в происходящем. На ней изображены таблицы виртуальных функций, созданные компилятором для Pet и Dog:



Обратите внимание: адрес speak() находится в таблице виртуальных функций Pet на том же месте, что и в классе Dog. Аналогично, если от Dog будет унаследован класс Pug, его версия sit() будет находиться в таблице виртуальных функций на том же месте, что и в классе Dog. Как было показано на примере ассемблерного кода, в сгенерированном фрагменте компилятор выбирает виртуальную функцию простым числовым смещением внутри таблицы виртуальных функций. Независимо от того, к какому из производных типов принадлежит объект, адреса функций следуют в таблице виртуальных функций всегда в одинаковом порядке, поэтому виртуальные функции тоже всегда вызываются одинаковым способом.

Но в нашем случае компилятор работает только с указателем на объект базового класса. В базовом классе определены только функции speak() и name(), и это единственные функции, которые разрешено вызывать компилятору через указатель. Откуда компилятору знать, что вы работаете с объектом Dog, если у него имеется только указатель на объект базового класса? Этот указатель может ссылаться и на другой тип, не имеющий функции sit(). В соответствующей ячейке таблицы виртуальных функций может находиться адрес какой-нибудь другой функции или его может не быть; в любом случае виртуальный вызов по адресу из таблицы виртуальных функций — совсем не то, чего вы добиваетесь. Так что компилятор совершенно правильно делает, защищая вас от вызовов виртуальных функций, существующих только в производных классах.

Также встречаются другие, менее распространенные ситуации, когда вам точно известно, что указатель на самом деле ссылается на объект конкретного производного класса. Если вы хотите вызвать функцию, существующую только в этом производном классе, указатель придется преобразовать. Следующая команда устраняет сообщение об ошибке, выдаваемое при компиляции предыдущей программы:

```
((Dog*)p[1])->sit()
```

Мы знаем, что p[1] ссылается на объект Dog, но в общем случае нам это неизвестно. Если задача приводит к тому, что вам необходимо знать точные типы всех объектов, переформулируйте ее — вероятно, вы неправильно используете виртуальные функции. Впрочем, в некоторых ситуациях архитектура программы оптимально работает, если вам известны точные типы всех объектов в универсальном контейнере (а иногда просто нет выбора). Так возникает проблема *идентификации типов во время выполнения* (RunTime Type Identification, RTTI).

RTTI применяется при *понижающем* приведении типа (указателей на базовый тип к указателям на тип производного класса) — как отмечалось выше, термины «повышающий» и «понижающий» предполагают, что базовый класс находится выше на диаграмме наследования. Повышающее приведение типа выполняется автоматически без потери информации и является абсолютно безопасным. Понижающее приведение типа небезопасно — на стадии компиляции фактические типы объектов неизвестны, поэтому необходимо точно знать, к какому типу относится объект. Если в результате приведения получится неверный тип, у вас будут большие неприятности.

Механизм RTTI рассматривается далее в этой главе, а во втором томе книги этой теме посвящается целая глава.

Расщепление объектов

При использовании полиморфизма существует четкое различие между передачей адресов объектов и передачей объектов по значению. Во всех примерах, приведенных ранее, и практически во всех дальнейших примерах передаются адреса, а не значения. Это объясняется тем, что все адреса имеют одинаковый размер[1], поэтому адрес объекта производного типа (который обычно имеет больший размер) передается точно так же, как объект базового типа. Как уже объяснялось, при использовании полиморфизма мы добиваемся именно этой цели — чтобы код, работающий с базовым типом, мог работать с объектами производных типов прозрачно для программиста.

Если выполнить повышающее приведение типа не со ссылкой или указателем, а с объектом, произойдет нечто неожиданное: объект «расщепляется», и от него остается лишь подобъект, соответствующий целевому типу после приведения. Следующий пример показывает, что происходит при расщеплении объекта:

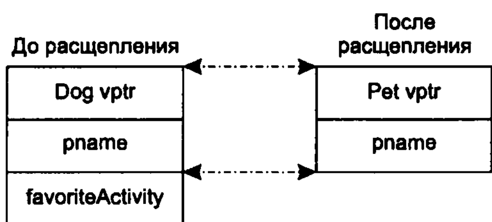
```
//: C15:ObjectSlicing.cpp
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
```

[1] Вообще говоря, не все указатели имеют одинаковые размеры на всех машинах, однако в контексте обсуждаемой темы можно считать их одинаковыми.

```
      favoriteActivity;
  }
};

void describe(Pet p) { // Расщепление объекта
  cout << p.description() << endl;
}

int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
} ///:~
```

Функция describe() получает объект типа Pet *по значению*, после чего она вызывает виртуальную функцию description() для объекта Pet. Можно предположить, что в main() первый вызов выведет строку «This is Alfred», а второй — «Fluffy likes to sleep». На самом деле при обоих вызовах используется версия description() базового класса.

В программе заслуживают внимания два обстоятельства. Во-первых, поскольку функция describe() получает *объект* Pet (а не указатель и не ссылку), при каждом вызове describe() объект размера Pet заносится в стек и уничтожается после вызова. Это означает, что если функции describe() передается объект класса, производного от Pet, то компилятор принимает его, но копирует лишь часть объекта, относящуюся к Pet. Таким образом, объект *расщепляется* с отбрасыванием производной части:



Остается понять, что происходит при вызове виртуальной функции. Вызов Dog::description() использует обе части: Pet, которая продолжает существовать, и Dog, которой не существует, потому что она была утрачена при расщеплении! Как же вызывается виртуальная функция?

Катастрофы не происходит, потому что объект передается по значению. Благодаря этому компилятор знает точный тип объекта, потому что производный объект был насильственно превращен в базовый объект. При передаче по значению используется копирующий конструктор объекта Pet, который инициализирует VPTR адресом таблицы виртуальных функций класса Pet и копирует только те части объекта, которые относятся к Pet. Копирующий конструктор явно не определен, поэтому он автоматически генерируется компилятором. В результате расщепления объект во всех отношениях действительно превращается в Pet.

Расщепление приводит к фактическому удалению части существующего объекта при его копировании в новый объект и не ограничивается простым изменением интерпретации адреса при использовании указателя или ссылки. Из-за этого по-

вышающее приведение типа для объектов выполняется относительно редко; более того, обычно его следует избегать. Обратите внимание: если бы в этом примере функция description() была преобразована в чисто виртуальную функцию базового класса (что вполне разумно, потому что функция ничего не делает в базовом классе), то компилятор предотвратил бы расщепление объекта, запрещая «создавать» объект базового типа, — а именно это происходит при повышающем приведении типа в случае передачи по значению. Возможно, это одно из самых полезных свойств чисто виртуальных функций: они предотвращают расщепление объекта (если кто-нибудь попытается это сделать, компилятор выдает сообщение об ошибке).

Перегрузка и переопределение

В главе 14 вы видели, что переопределение функции, перегруженной в базовом классе, скрывает все остальные версии функции из базового класса. При переопределении виртуальной функции дело обстоит несколько иначе. Рассмотрим слегка измененную версию примера NameHiding.cpp из главы 14:

```
//: C15:NameHiding2.cpp
// Виртуальные функции ограничивают перегрузку
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  virtual int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Переопределение виртуальной функции:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // Невозможно изменить тип возвращаемого значения:
  //! void f() const{ cout << "Derived3::f()\n";}
};

class Derived4 : public Base {
public:
```

```
  // Изменение списка аргументов:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // Строковая версия скрывается
  Derived4 d4;
  x = d4.f(1);
//!  x = d4.f(); // Версия f() скрывается
//!  d4.f(s); // Строковая версия скрывается
  Base& br = d4; // Повышающее приведение типа
//!  br.f(1); // Производная версия недоступна
  br.f(); // Базовая версия доступна
  br.f(s); // Базовая версия доступна
} ///:~
```

Прежде всего следует заметить, что в классе Derived3 компилятор не позволит изменить тип возвращаемого значения переопределенной функции (если бы функция f() не была виртуальной, это было бы возможно). Это важное ограничение — компилятор должен гарантировать возможность полиморфного вызова функции через базовый класс. Если базовый класс ожидает, что f() возвращает int, то версия f() в производном классе должна соблюдать этот контракт, иначе работа программы будет нарушена.

Правило, приведенное в главе 14, по-прежнему работает: если переопределить одну из перегруженных функций в базовом классе, остальные перегруженные версии скрываются в производном классе. В функции main() код для тестирования класса Derived4 показывает, что это происходит даже в том случае, если новая версия f() не переопределяет существующий интерфейс виртуальных функций, — f(int) скрывает обе версии f() базового класса. Тем не менее при повышающем приведении типа d4 к Base остаются доступными только версии базового класса (потому что это гарантировано контрактом базового класса), а версия из производного класса недоступна, поскольку она не входит в контракт базового класса.

Изменение типа возвращаемого значения

Приведенный выше класс Derived3 наводит на мысль, что изменить тип возвращаемого значения виртуальной функции при переопределении в принципе невозможно. Обычно это действительно так, однако существует особый случай, в котором вы можете слегка изменить тип возвращаемого значения. Если функция возвращает указатель или ссылку на объект базового класса, то переопределенная версия функции может вернуть указатель или ссылку на класс, производный от него. Пример:

```
//: C15:VariantReturn.cpp
// Возвращение указателя или ссылки на производный тип
```

```
// при переопределении функции
#include <iostream>
#include <string>
using namespace std;

class PetFood {
public:
  virtual string foodType() const = 0;
};

class Pet {
public:
  virtual string type() const = 0;
  virtual PetFood* eats() = 0;
};

class Bird : public Pet {
public:
  string type() const { return "Bird"; }
  class BirdFood : public PetFood {
  public:
    string foodType() const {
      return "Bird food";
    }
  };
  // Повышающее приведение к базовому типу:
  PetFood* eats() { return &bf; }
private:
  BirdFood bf;
};

class Cat : public Pet {
public:
  string type() const { return "Cat"; }
  class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
  };
  // Возвращение фактического типа:
  CatFood* eats() { return &cf; }
private:
  CatFood cf;
};

int main() {
  Bird b;
  Cat c;
  Pet* p[] = { &b, &c, };
  for(int i = 0; i < sizeof p / sizeof *p; i++)
    cout << p[i]->type() << " eats "
         << p[i]->eats()->foodType() << endl;
  // Может возвращать фактический тип:
  Cat::CatFood* cf = c.eats();
  Bird::BirdFood* bf;
  // Не может возвращать фактический тип:
//!  bf = b.eats();
  // Необходимо понижающее приведение типа:
  bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} ///:~
```

Функция Pet::eats() возвращает указатель на PetFood. В классе Bird эта функция перегружается точно так же, как в базовом классе, включая тип возвращаемого значения. Таким образом, Bird::eats() повышает тип BirdFood до PetFood.

Однако в классе Cat функция eats() возвращает указатель на CatFood — тип, производный от PetFood. Этот код компилируется только потому, что возвращаемый тип является производным от возвращаемого типа функции базового класса. В этом случае контракт по-прежнему выполняется; eats() всегда возвращает указатель на PetFood.

С позиций полиморфизма такой подход не кажется обязательным. Почему бы не выполнить повышающее приведение всех возвращаемых типов к PetFood*, как это сделано в Bird::eats()? Обычно это хорошее решение, но в конце main() мы видим различие: Cat::eats() может вернуть тип PetFood, тогда как возвращаемое значение Bird::eats() приходится приводить к нужному типу «вниз».

Итак, возможность возвращать фактический тип предотвращает потерю специфической информации о типе при автоматическом повышающем приведении типа. Тем не менее в большинстве случаев достаточно возврата базового типа, поэтому такая возможность применяется относительно редко.

Виртуальные функции и конструкторы

При создании объекта, содержащего виртуальные функции, его указатель VPTR должен быть инициализирован адресом таблицы виртуальных функций. Инициализация должна произойти раньше, чем в программе представится возможность вызова виртуальной функции. Как нетрудно догадаться, раз жизненный цикл объекта начинается с вызова конструктора, конструктор также инициализирует VPTR. Код инициализации скрыто вставляется компилятором в начало конструктора. Как отмечалось в главе 14, если конструктор не был явно создан в классе, то он будет сгенерирован компилятором. Если класс содержит виртуальные функции, то сгенерированный конструктор будет включать код инициализации VPTR. Этот факт приводит к ряду следствий.

Первое следствие — эффективность. Подставляемые (inline) функции определяются, чтобы избавиться от издержек, связанных с вызовами небольших функций. Если бы в C++ отсутствовали подставляемые функции, их можно было бы создать при помощи препроцессора. Однако препроцессор не знаком с концепциями уровней доступа или классов, поэтому функции классов не могут создаваться при помощи макросов. Кроме того, препроцессорные макросы вообще не годятся для конструкторов, которые содержат скрытый код, вставляемый компилятором.

Занимаясь оптимизацией, помните, что компилятор включает в конструкторы скрытый код. Компилятор должен не только инициализировать VPTR, но и проверять значение this (на случай, если функция operator new вернет ноль) и вызывать конструкторы базовых классов. По совокупности этот код может сильно изменить то, что вы считали вызовом небольшой подставляемой функции. В частности, размер конструктора может перевесить экономию от затрат на вызов функции. При большом количестве вызовов подставляемых конструкторов программа заметно вырастет без сколько-нибудь заметного выигрыша в быстродействии.

Конечно, это вовсе не означает, что мелкие конструкторы нужно немедленно переделывать в неподставляемые функции, потому что их гораздо удобнее оформ-

лять в виде подставляемых. Но в процессе оптимизации программы обязательно обдумайте возможность отказа от подстановки конструкторов.

Порядок вызова конструкторов

Вторая интересная особенность конструкторов и виртуальных функций связана с порядком вызова конструкторов и виртуальных функций из конструкторов.

Конструктор производного класса всегда вызывает все конструкторы базового класса. Это логично, поскольку конструктор выполняет особое задание: он обеспечивает правильность построения объектов. Производный класс имеет доступ только к своим переменным, но не к переменным базового класса. Лишь конструктор базового класса может правильно инициализировать свои переменные. А значит, очень важно, чтобы были вызваны *все* конструкторы, иначе объект может быть сконструирован неправильно. Становится ясно, почему компилятор гарантирует вызов конструктора для каждой части производного класса. Если конструктор базового класса не указан в списке инициализирующих значений, будет вызван конструктор по умолчанию. Если конструктора по умолчанию не существует, компилятор выдает сообщение об ошибке.

Порядок вызова конструкторов играет важную роль. В производном классе вы знаете все о базовом классе и можете свободно обращаться к его открытым и защищенным членам. Следовательно, предполагается, что с точки зрения производного класса все члены базового класса имеют действительные значения. При вызове обычной функции класса все переменные всех частей объекта заведомо инициализированы. С другой стороны, в конструкторе мы должны быть уверены в том, что инициализированы все переменные класса, используемые этим конструктором. Гарантировать это можно только одним способом: предварительно вызвать конструктор базового класса. Только в этом случае в конструкторе производного класса все переменные, доступные для базового класса, будут инициализированы. Гарантированная инициализация всех переменных в конструкторе также означает, что все внутренние объекты (то есть объекты, включенные в класс посредством композиции) должны по возможности инициализироваться в списке инициализирующих значений конструктора. Если следовать этому правилу, можно быть уверенным в том, что переменные *и* внутренние объекты базового класса всегда инициализированы в текущем объекте.

Поведение виртуальных функций внутри конструкторов

Иерархия вызовов конструкторов создает интересную дилемму. Что произойдет, если внутри конструктора вызывается виртуальная функция? Внутри обычной функции класса предсказать дальнейшее нетрудно — виртуальный вызов проходит связывание во время выполнения, потому что объект не знает, принадлежит он классу, в котором определена функция, или же классу, производному от него. По аналогии можно предположить, что это же произойдет внутри конструкторов.

Однако это предположение ошибочно. При вызове виртуальной функции внутри конструктора используется только локальная версия функции. Иначе говоря, механизм виртуального вызова в конструкторах не работает.

Такое поведение оправданно по двум причинам. С концептуальной точки зрения конструктор должен «сотворить» объект (что вряд ли можно назвать рядовой

задачей). Внутри любого конструктора объект может быть сформирован лишь частично — известно лишь то, что объекты базового класса были инициализированы, но о производных классах ничего не известно. С другой стороны, вызов виртуальной функции направлен «дальше» в иерархии с вызовом функции производного класса. Если бы это было возможно в конструкторе, то вызванная функция могла бы выполнять операции с неинициализированными переменными, а это наверняка бы плохо кончилось.

Существует и другая, чисто механическая причина. При вызове конструктора одной из первых выполняемых операций является инициализация указателя VPTR. Но при этом конструктор знает лишь свой «текущий» тип — тот, для которого он был написан. Код конструктора ничего не знает о том, входит ли объект в другой класс. Компилятор генерирует код конструктора для конкретного класса, а не для базового или производного класса (поскольку класс не знает, какие классы являются производными от него). Следовательно, используемый указатель VPTR должен ссылаться на таблицу виртуальных функций *этого* класса. VPTR сохраняет свое значение на протяжении всего жизненного цикла объекта, *если только* после него не вызваны другие конструкторы. При последующем вызове конструктора производного класса этот конструктор переводит VPTR на *свою* таблицу виртуальных функций, и это продолжается до последнего конструктора в цепочке. Состояние VPTR определяется последним вызванным конструктором. В этом заключается вторая причина, по которой конструкторы вызываются по порядку — от базовых классов к производным.

Но в процессе вызова серии конструкторов каждый конструктор настраивает VPTR на свою таблицу виртуальных функций. Если бы для вызовов функций использовался виртуальный механизм, то вызовы осуществлялись бы через таблицу виртуальных функций текущего, а не последнего из производных классов (как это происходит после вызова *всех* конструкторов). Кроме того, многие компиляторы распознают вызовы виртуальных функций в конструкторах и выполняют раннее связывание, потому что они учитывают, что позднее связывание приведет лишь к вызову локальной функции. В любом случае вызов виртуальной функции в конструкторе не приведет к тем результатам, на которые вы могли бы рассчитывать.

Деструкторы и виртуальные деструкторы

Ключевое слово virtual не может использоваться для конструкторов, однако деструкторы могут и часто должны быть виртуальными.

Конструктор выполняет особую работу: он собирает объект по частям, последовательно вызывая сначала базовый конструктор, а затем конструкторы производных классов (попутно также должны быть вызваны конструкторы внутренних объектов). Деструктор тоже выполняет особую работу: он должен «разобрать» объект, который может входить в иерархию классов. Для этого компилятор генерирует код вызова всех деструкторов, но в порядке, *обратном* порядку вызова конструкторов. Иначе говоря, деструктор начинает с последнего производного класса и продвигается к базовому классу. Такая последовательность вызова безопасна, поскольку текущий деструктор знает, что члены базового класса продолжают существовать. Если в деструкторе потребуется вызвать функцию базо-

вого класса, ничто не помешает вам сделать это. Таким образом, деструктор выполняет собственную зачистку, вызывает следующий деструктор, который тоже выполняет собственную зачистку, и т. д. Каждый деструктор знает, *от какого* класса он происходит, но не знает, какие классы являются производными от него.

Учтите, что подобная иерархия вызовов необходима только в конструкторах и деструкторах (и поэтому соответствующий код автоматически генерируется компилятором). Во всех остальных случаях будет вызываться только *заданная* функция (но не ее версии базового класса), независимо от того, является она виртуальной или нет. Версия базового класса может быть вызвана из обычной функции (виртуальной или нет) только в одном случае — если вы *явно* вызываете эту функцию в программе.

Но что произойдет, если вы захотите работать с объектом через указатель на его базовый класс (то есть через общий интерфейс)? Подобные операции играют важную роль в объектно-ориентированном программировании. Если вызвать оператор delete с таким указателем для объекта, созданного в куче оператором new, возникают проблемы. Если указатель ссылается на объект базового класса, то при вызове delete компилятор сможет вызвать только версию деструктора из базового класса. Звучит знакомо? Перед вами та же проблема, для решения которой в общем случае создавались виртуальные функции. К счастью, механизм виртуального вызова работает с деструкторами точно так же, как и со всеми остальными функциями, исключая конструкторы.

```
//: C15:VirtualDestructors.cpp
// Поведение виртуального и невиртуального деструкторов
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1; // Повышающее приведение типа
  delete bp;
  Base2* b2p = new Derived2; // Повышающее приведение типа
  delete b2p;
} ///:~
```

Если запустить программу, вы увидите, что команда delete bp ведет к вызову только деструктора базового класса, тогда как delete b2p вызывает сначала деструктор производного класса, а затем деструктор базового класса; это именно то, что требовалось. Ошибка, когда программист забывает объявить деструктор виртуальным, весьма коварна — часто она не оказывает прямого влияния на поведение программы, но создает незаметную утечку памяти. А тот факт, что *кое-какое* уничтожение объектов все же происходит, лишь маскирует проблему.

Деструктор, как и конструктор, считается «особой» функцией, однако деструкторы могут быть виртуальными, потому что объект уже знает, к какому типу он относится (а в процессе конструирования такой информации нет). Указатель VPTR сконструированного объекта уже инициализирован, что делает возможным вызов виртуальных функций.

Чисто виртуальные деструкторы

Стандарт C++ разрешает использовать чисто виртуальные деструкторы, но на них накладывается дополнительное ограничение: программист обязан предоставить тело функции для чисто виртуального деструктора. Поначалу это выглядит противоестественно: как виртуальная функция может быть «чистой», если ей нужно тело? Но с учетом того, что конструкторы и деструкторы выполняют особые операции, такой подход выглядит более логично, особенно если вспомнить, что в процессе уничтожения всегда вызываются все деструкторы в иерархии класса. Если бы вы *могли* опустить определение для чисто виртуального деструктора, тело какой функции было бы вызвано при уничтожении объекта? Следовательно, компилятор и компоновщик *обязаны* обеспечить существование тела функции для чисто виртуального деструктора.

Зачем же нужны «чистые» деструкторы, обладающие телом? Единственное различие между чисто виртуальным и просто виртуальным деструктором заключается в том, что базовый класс с чисто виртуальным деструктором становится абстрактным (то есть вы не сможете создавать объекты базового класса). Впрочем, этим свойством обладают любые чисто виртуальные функции базовых классов, а не только деструкторы.

Но если класс наследует от другого класса, содержащего чисто виртуальный деструктор, ситуация усложняется. В отличие от остальных чисто виртуальных функций, вы *не обязаны* предоставлять определение чисто виртуального деструктора в производном классе. Тот факт, что следующая программа компилируется и компонуется, доказывает это утверждение:

```
//: C15:UnAbstract.cpp
// Странное поведение чисто виртуальных деструкторов

class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// Переопределять деструктор не обязательно?

int main() { Derived d; } ///:~
```

Обычно при наличии чисто виртуальной функции в базовом классе производный класс становится абстрактным, если в нем не определена эта функция (а также все остальные чисто виртуальные функции). Может показаться, что в данном случае это правило не выполняется. Но вспомните, о чем говорилось выше: компилятор *автоматически* создает определение деструктора для каждого класса, не имеющего такого определения. Именно это происходит в нашем случае — деструктор базового класса скрыто переопределяется. Определение предоставляется компилятором, поэтому класс Derived абстрактным не является.

Тогда возникает интересный вопрос: а для чего вообще нужны чисто виртуальные деструкторы? В отличие от обычных чисто виртуальных функций, вы *должны* предоставить тело функции в базовом классе. С другой стороны, функцию не обязательно переопределять в производном классе, поскольку компилятор сгенерирует деструктор за вас. Так чем же обычный виртуальный деструктор отличается от чисто виртуального деструктора?

Единственное различие проявляется в классах, содержащих только одну-единственную чисто виртуальную функцию, которой в данном случае является деструктор. В этом случае чистота деструктора только предотвращает создание экземпляров базового класса, и ничего более. Создание экземпляров базового класса могло бы быть предотвращено другими чисто виртуальными функциями, а если их нет, учитывается чисто виртуальный деструктор. Итак, хотя наличие виртуального деструктора имеет важные последствия, его чистота не столь существенна.

При запуске следующего примера вы увидите, что тело чисто виртуального деструктора, как и любого деструктора базового класса, вызывается после версии производного класса:

```
//: C15:PureVirtualDestructors.cpp
// Чисто виртуальные деструкторы требуют
// наличия тела функции
#include <iostream>
using namespace std;

class Pet {
public:
  virtual ~Pet() = 0;
};

Pet::~Pet() {
  cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
  ~Dog() {
    cout << "~Dog()" << endl;
  }
};

int main() {
  Pet* p = new Dog; // Повышающее приведение типа
  delete p; // Вызов виртуального деструктора
} ///:~
```

Остается дать общую рекомендацию: если класс содержит хотя бы одну виртуальную функцию, обязательно включите в него виртуальный деструктор (даже если он ничего не делает). Тем самым вы застрахуетесь от сюрпризов в будущем.

Виртуальные функции в деструкторах

В процессе уничтожения объекта происходит нечто такое, что может показаться неожиданным. При вызове виртуальной функции из обычной функции класса используется механизм позднего связывания. К деструкторам (как виртуальным, так и обычным) это не относится. Внутри деструктора вызывается только «локальная» версия функции класса; механизм виртуального вызова игнорируется.

```
//: C15:VirtualsInDestructors.cpp
// Вызов виртуальных функций из деструктора
#include <iostream>
using namespace std;

class Base {
public:
  virtual ~Base() {
    cout << "Base1()\n";
    f();
  }
  virtual void f() { cout << "Base::f()\n"; }
};

class Derived : public Base {
public:
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};

int main() {
  Base* bp = new Derived; // Повышающее приведение типа
  delete bp;
} ///:~
```

Во время выполнения деструктора версия Derived::f() *не вызывается*, хотя функция f() является виртуальной.

Почему? Давайте предположим, что механизм виртуального вызова может использоваться внутри деструкторов. Тогда процесс разрешения виртуального вызова мог бы привести к функции класса, находящегося в иерархии наследования дальше класса с текущим деструктором (то есть в «более производном» классе). Но деструкторы вызываются от производных классов к базовым, поэтому вызванная функция может обратиться к тем частям объекта, которые *уже были уничтожены*! Вместо этого компилятор рассматривает вызов на стадии компиляции и вызывает только «локальную» версию функции. То же самое относится к конструкторам (см. выше), но по другим причинам: в конструкторе информация о типе недоступна, тогда как в деструкторе эта информация (проще говоря, VPTR) доступна, но ненадежна.

Создание однокоренной иерархии

На страницах книги неоднократно встречались контейнерные классы Stack и Stash, и каждый раз неизменно возникал вопрос о «принадлежности» объектов. «Владель-

цем» объекта называется тот, кто отвечает за вызов оператора delete для динамически (с использованием оператора new) созданных объектов. Контейнеры должны быть достаточно гибкими, чтобы в них можно было хранить объекты разных типов. Для этого в них содержались указатели на void, то есть контейнер не знал типа хранящихся в нем объектов. Удаление указателя на void не приводит к вызову деструктора, поэтому контейнеры не могли отвечать за уничтожение своих элементов.

В примере C14:InheritStack.cpp было представлено одно из возможных решений: на базе класса Stack был создан новый производный класс, который получал и возвращал только указатели на string. Поскольку класс знал, что в нем могут храниться только указатели на объекты string, он мог правильно удалить их. Это хорошее решение, однако вам придется создавать новый контейнерный класс для каждого типа, который может храниться в контейнере (в главе 16 будет показано, как эта задача решается при помощи шаблонов).

Проблема состоит в следующем: мы хотим, чтобы в контейнере можно было хранить объекты разных типов, но при этом обойтись без указателей на void. Другое решение основано на полиморфизме: все объекты, хранящиеся в контейнере, являются производными от одного базового класса. Другими словами, в контейнере хранятся объекты базового класса, для которых вызываются виртуальные функции, в частности виртуальные деструкторы для решения проблемы принадлежности.

В этом решении используется так называемая *однокоренная иерархия*. Оказывается, у однокоренных иерархий имеется много других достоинств; все остальные объектно-ориентированные языки, кроме C++, заставляют программиста ограничиваться только такими иерархиями — любой создаваемый класс явно или косвенно наследует от единого базового класса, определенного создателями языка. При разработке C++ было решено, что обязательное применение единого базового класса породит слишком много непроизводительных затрат, поэтому от этой идеи отказались. Тем не менее вы можете использовать единый базовый класс в своих проектах; эта тема будет рассматриваться во втором томе книги.

Чтобы решить проблему принадлежности объектов, мы создаем чрезвычайно простой базовый класс Object, содержащий только виртуальный деструктор. Далее в контейнере Stack сохраняются объекты классов, производных от Object:

```
//: C15:OStack.h
// Использование однокоренной иерархии
#ifndef OSTACK_H
#define OSTACK_H

class Object {
public:
  virtual ~Object() = 0;
};

// Обязательное определение:
inline Object::~Object() {}

class Stack {
  struct Link {
    Object* data;
    Link* next;
    Link(Object* dat, Link* nxt) :
      data(dat), next(nxt) {}
```

```
    }* head;
  public:
    Stack() : head(0) {}
    ~Stack(){
      while(head)
        delete pop();
    }
    void push(Object* dat) {
      head = new Link(dat, head);
    }
    Object* peek() const {
      return head ? head->data : 0;
    }
    Object* pop() {
      if(head == 0) return 0;
      Object* result = head->data;
      Link* oldHead = head;
      head = head->next;
      delete oldHead;
      return result;
    }
  };
  #endif // OSTACK_H ///:~
```

Чтобы упростить программу и хранить как можно больше кода в заголовочном файле, определение чисто виртуального деструктора(обязательное) оформлено как подставляемое в заголовочном файле; функция pop() (которая может оказаться слишком большой для подстановки) также является подставляемой.

Вместо указателей на void объекты Link теперь содержат указатели на Object, а контейнер Stack принимает и возвращает только указатели на Object. Класс Stack стал более гибким, поскольку в нем теперь могут храниться объекты разных типов, а все оставшиеся объекты уничтожаются автоматически. Впрочем, появилось новое ограничение: все объекты, помещаемые в Stack, должны наследовать от Object (это ограничение устраняется с помощью шаблонов — см. главу 16). Хорошо, если классы для хранения в контейнере пишутся «с нуля», но что делать, если у вас уже имеется готовый класс вроде string, объекты которого нужно хранить в стеке? В этом случае новый класс нужно сделать производным как от string, так и от Object, то есть он должен наследовать от обоих классов. Такая ситуация называется *множественным наследованием*, и во втором томе книги ей будет посвящена целая глава, в которой будет показано, что множественное наследование нередко создает массу сложностей и пользоваться им нужно крайне осторожно. Впрочем, наша ситуация достаточно проста, чтобы избежать всех ловушек множественного наследования:

```
//: C15:OStackTest.cpp
//{T} OStackTest.cpp
#include "OStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// Множественное наследование. Класс должен быть производным
// как от string, так и от Object:
class MyString: public string, public Object {
```

```
public:
  ~MyString() {
    cout << "deleting string: " << *this << endl;
  }
  MyString(string s) : string(s) {}
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Аргумент - имя файла
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Чтение файла и сохранение строк в стеке:
  while(getline(in, line))
    textlines.push(new MyString(line));
  // Извлечение строк из стека:
  MyString* s;
  for(int i = 0; i < 10; i++) {
    if((s=(MyString*)textlines.pop())==0) break;
      cout << *s << endl;
      delete s;
  }
  cout << "Letting the destructor do the rest:"
    << endl;
} ///:~
```

Программа похожа на предыдущую версию тестовой программы для Stack, но обратите внимание на то, что из стека извлекается всего 10 элементов; вероятно, в стеке останутся неизвлеченные объекты. Контейнер Stack знает, что в нем хранятся объекты Object, и это позволяет деструктору правильно уничтожать объекты. В этом можно убедиться по результатам работы программы, поскольку объекты MyString выводят сообщения по мере уничтожения.

Вариант с контейнерами для хранения потомков Object достаточно разумен, *если* вы работаете с однокоренной иерархией (что обеспечивается либо самим языком, либо дополнительным требованием, согласно которому все классы должны наследовать от Object). При этом любой объект гарантированно является производным от Object, а операции с контейнером относительно просты. Однако в C++ нельзя ожидать, что все классы будут удовлетворять этому условию, поэтому выбор данного решения неизбежно приводит к множественному наследованию. В главе 16 будет показано, что эта задача гораздо проще и элегантнее решается при помощи шаблонов.

Перегрузка операторов

Операторные функции, как и все остальные функции классов, могут объявляться виртуальными. Однако реализация виртуальных операторов часто усложняется тем, что оператор может работать с двумя объектами неизвестных типов. В частности, эта ситуация типична для математических операторов, которые часто перегружаются. Для примера рассмотрим систему для работы с матрицами, векторами и скалярными величинами, в которой все три класса являются производными от класса Math:

```
//: C15:OperatorPolymorphism.cpp
// Полиморфизм при перегрузке операторов
#include <iostream>
```

```
using namespace std:

class Matrix:
class Scalar:
class Vector:

class Math {
public.
  virtual Math& operator*(Math& rv) = 0:
  virtual Math& multiply(Matrix*) = 0:
  virtual Math& multiply(Scalar*) = 0:
  virtual Math& multiply(Vector*) = 0:
  virtual ~Math() {}
}.

class Matrix : public Math {
public·
  Math& operator*(Math& rv) {
    return rv.multiply(this): // Вторая диспетчеризация
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Matrix" << endl:
    return *this:
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Matrix" << endl:
    return *this:
  }
  Math& multiply(Vector*) {
    cout << "Vector * Matrix" << endl:
    return *this.
  }
}.

class Scalar : public Math {
public.
  Math& operator*(Math& rv) {
    return rv.multiply(this). // Вторая диспетчеризация
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Scalar" << endl:
    return *this:
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Scalar" << endl:
    return *this.
  }
  Math& multiply(Vector*) {
    cout << "Vector * Scalar" << endl:
    return *this:
  }
}:

class Vector : public Math {
public.
  Math& operator*(Math& rv) {
    return rv.multiply(this): // Вторая диспетчеризация
  }
  Math& multiply(Matrix*) {
```

```
    cout << "Matrix * Vector" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Vector" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Vector" << endl;
    return *this;
  }
};

int main() {
  Matrix m; Vector v; Scalar s;
  Math* math[] = { &m, &v, &s };
  for(int i = 0; i < 3; i++)
    for(int j = 0; j < 3; j++) {
      Math& m1 = *math[i];
      Math& m2 = *math[j];
      m1 * m2;
    }
} ///:~
```

Для простоты в этом примере перегружается только функция operator*. Мы хотим, чтобы любые два объекта Math можно было умножить друг на друга и получить желаемый результат. Обратите внимание: умножение матрицы на вектор является совершенно иной операцией, чем умножение вектора на матрицу.

Проблема заключается в том, что в выражении m1*m2 из функции main() задействованы две операции повышающего приведения к типу Math, а следовательно, два объекта неизвестного типа. Виртуальная функция способна выполнить только *одиночную диспетчеризацию*, то есть определить тип одного неизвестного объекта. Для определения обоих типов в нашем примере используется прием *множественной диспетчеризации*, при котором то, что выглядит как один вызов виртуальной функции, приводит к вызову второй виртуальной функции. К моменту второго вызова мы уже определили оба типа объектов и можем выполнить необходимые действия. В происходящем трудно разобраться сразу, но если вы как следует поразмыслите над этим примером, картина начнет проясняться. Данная тема более подробно рассматривается во втором томе книги в главе, посвященной эталонам проектирования.

Понижающее приведение типа

Как нетрудно догадаться, наряду с повышающим приведением типа (перемещением вверх по иерархии наследования) должно существовать *понижающее* приведение типа, то есть перемещение вниз по иерархии. Но повышающее приведение никогда не вызывает проблем, поскольку классы всегда преобразуются в более общую форму. При повышающем приведении типа всегда четко выделяется класс-предок (обычно один, если не считать случая множественного наследования), а при понижающем приведении типа обычно возможны разные варианты. Так, класс Circle (круг) является специализацией базового класса Shape (геометрическая фигура) и может быть приведен к нему повышающим преобразованием, однако

понижающее преобразование Shape может привести к разным классам — Circle (круг), Square (квадрат), Triangle (треугольник) и т. д. Спрашивается, как безопасно выполнить понижающее приведение типа? (Хотя существует еще более важный вопрос: зачем вообще выполнять понижающее преобразование, когда можно автоматически определить нужный тип при помощи полиморфизма? Тема предотвращения операций понижающего приведения типа рассматривается во втором томе книги.)

В C++ предусмотрена особая разновидность операций *явного приведения типа*, представленных в главе 3, которые выполняются с помощью оператора dynamic_cast. Этот оператор реализует понижающее преобразование, безопасное по отношению к типам. Когда dynamic_cast используется для приведения к конкретному типу, возвращаемое значение представляет собой указатель на этот тип только в том случае, если такое преобразование допустимо и успешно выполнено. В противном случае возвращается ноль — признак того, что целевой тип был задан неправильно. Простейший пример:

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Повышающее приведение типа
  // Попытка приведения к типу Dog*:
  Dog* d1 = dynamic_cast<Dog*>(b);
  // Попытка приведения к типу Cat*:
  Cat* d2 = dynamic_cast<Cat*>(b);
  cout << "d1 = " << (long)d1 << endl;
  cout << "d2 = " << (long)d2 << endl;
} ///:~
```

Оператор dynamic_cast работает только в полноценных полиморфных иерархиях (то есть иерархиях, содержащих виртуальные функции), поскольку для определения фактического типа dynamic_cast использует информацию, хранящуюся в таблице виртуальных функций. В нашем примере базовый класс содержит виртуальный деструктор, и этого достаточно. В функции main() указатель на Cat повышается до Pet, после чего делаются попытки преобразовать его в указатель на Dog и Cat. Значения обоих указателей выводятся. При запуске программы становится видно, что неверное понижающее приведение типа приводит к нулевому результату. Конечно, при выполнении понижающих преобразований в программе вы должны убедиться в том, что результат преобразования отличен от нуля. Также нельзя предполагать, что преобразованный указатель сохранит свое значение, потому что операции повышающего и понижающего приведения типа иногда сопровождаются модификацией указателя (особенно при множественном наследовании).

Вызов оператора dynamic_cast требует определенных затрат. Эти затраты невелики, однако большое количество вызовов dynamic_cast в программе может отразиться на ее производительности (в таких случаях стоит серьезно пересмотреть архитектуру программы). Иногда при понижающем приведении типа имеется дополнительная информация, благодаря которой вы точно знаете, с каким типом

имеете дело. В таких случаях преобразование оператором dynamic_cast с его дополнительными затратами становится ненужным и вместо него удается обойтись оператором static_cast. Вот как это может происходить:

```
//: C15:StaticHierarchyNavigation.cpp
// Перемещение по иерархии классов с применением оператора static_cast
#include <iostream>
#include <typeinfo>
using namespace std;

class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
  Circle c;
  Shape* s = &c; // Повышающее приведение типа: все нормально
  // Более очевидная, но не обязательная форма:
  s = static_cast<Shape*>(&c);
  // (Поскольку повышающее приведение типа является безопасной
  // и часто выполняемой операцией, static_cast лишь загромождает программу)
  Circle* cp = 0;
  Square* sp = 0;
  // Для статического перемещения по иерархии классов
  // необходима дополнительная информация о типе:
  if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // Статическое перемещение используется ТОЛЬКО
  // для повышения эффективности; dynamic_cast всегда безопаснее.
  // Команда
  // Other* op = static_cast<Other*>(s);
  // выдает полезное сообщение об ошибке, тогда как для
  Other* op2 = (Other*)s;
  // такое сообщение не выдается.
} ///:~
```

В этой программе задействована новая возможность, полное описание которой будет приведено лишь во втором томе книги, в главе, посвященной механизму RTTI. Этот механизм позволяет восстановить информацию о типе, потерянную в результате повышающего преобразования. Оператор dynamic_cast в действительности реализует одну из форм RTTI. В приведенном примере ключевое слово typeid (объявленное в заголовочном файле <typeinfo>) используется для идентификации типов указателей. Мы сравниваем тип указателя на Shape с типами указателей на Circle и Square и проверяем, совпадают ли они. Определение информации о типе не исчерпывает всех возможностей RTTI; вы можете относительно легко реализовать собственную систему проверки информации о типе с использованием виртуальных функций.

Мы создаем объект Circle и выполняем повышающее приведение его адреса до указателя на Shape; вторая версия выражения показывает, как более понятно

провести повышающее преобразование при помощи оператора static_cast. Тем не менее повышающее приведение типа всегда безопасно и является достаточно стандартной операцией, поэтому, по мнению автора, оператор static_cast лишь загромождает программу, а следовательно, не является необходимым.

Механизм RTTI используется для определения типа, после чего оператор static_cast выполняет понижающее преобразование. В этой архитектуре процесс не отличается от случая с оператором dynamic_cast. Чтобы узнать, успешно ли прошло преобразование, прикладному программисту придется выполнить дополнительную проверку. Использование static_cast вместо dynamic_cast (еще раз напомним, что перед использованием оператора dynamic_cast стоит внимательно проанализировать архитектуру приложения) обычно происходит в ситуациях, более детерминированных по сравнению с этим примером.

Если иерархия классов не содержит виртуальных функций (что обычно является признаком сомнительной архитектуры) или вы располагаете другой информацией, позволяющей безопасно выполнять понижающее приведение типа, static_cast обычно работает чуть быстрее dynamic_cast. Кроме того, в отличие от традиционного преобразования типа, оператор static_cast не позволит выйти за пределы иерархии, поэтому он безопаснее. С другой стороны, статическое перемещение по иерархии всегда сопряжено с определенным риском, поэтому на практике (кроме особых случаев) лучше использовать dynamic_cast.

Итоги

Полиморфизм, реализованный в C++ в виде виртуальных функций, буквально означает «различные формы». В объектно-ориентированном программировании имеется единый интерфейс (общий интерфейс базового класса) и различные формы использования этого интерфейса, то есть разные версии виртуальных функций.

Как было показано в этой главе, невозможно понять (и даже просто придумать) пример полиморфизма без абстракции и наследования. Полиморфизм относится к числу тех аспектов языка, которые не могут рассматриваться сами по себе (как, например, const или switch), а работают лишь в совокупности, как часть «большой картины» отношений между классами. Некоторых программистов сбивают с толку другие, не являющиеся объектно-ориентированными возможности C++ (такие, как перегрузка и аргументы по умолчанию), которые иногда выдаются за объектно-ориентированные. Не позволяйте себя обмануть; там, где нет позднего связывания, нет и полиморфизма.

Для эффективного использования полиморфизма (а следовательно, и объектно-ориентированных методов) в программах вы должны расширить свои представления о программировании так, чтобы понимать не только переменные и сообщения отдельных классов, но и концепции общности классов и их отношений друг с другом. Хотя это потребует значительных усилий, ваши старания не пропадут напрасно — их результатом станет ускорение разработки программ, улучшение логической организации и расширяемости программ, а также упрощение их сопровождения.

Полиморфизм завершает знакомство с объектно-ориентированными средствами языка, однако в C++ имеется еще два принципиально важных механизма: шаб-

лоны (кратко представленные в главе 16 и гораздо более подробно описанные во втором томе) и обработка исключений (рассматривается во втором томе). Они расширяют возможности программиста в такой же степени, как и любой из объектно-ориентированных механизмов (абстрактные типы, наследование и полиморфизм).

Упражнения

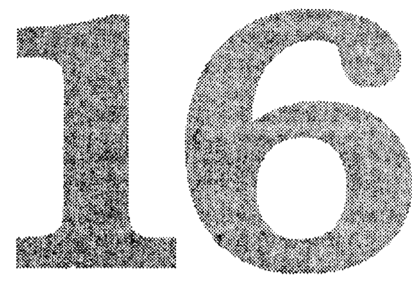
1. Создайте простую иерархию геометрических фигур, состоящую из базового класса Shape и производных классов Circle, Shape и Triangle. Включите в базовый класс виртуальную функцию draw() и переопределите ее в производных классах. Создайте массив указателей на объекты, созданные в куче (с повышающим преобразованием), и вызовите draw() через указатели базового класса, чтобы проверить поведение виртуальных функций. Выполните программу в пошаговом режиме, если ваш отладчик позволяет это сделать.
2. Измените упражнение 1 так, чтобы функция draw() была чисто виртуальной. Попробуйте создать объект типа Shape. Попробуйте вызвать чисто виртуальную функцию в конструкторе и посмотрите, что произойдет. Оставив функцию draw() чисто виртуальной, предоставьте для нее определение.
3. В упражнении 2 создайте функцию, которая получает объект Shape *по значению*, и попробуйте передать в аргументе объект производного класса. Посмотрите, что произойдет. Исправьте функцию, передав ссылку на объект Shape.
4. Измените пример C14:Combined.cpp так, чтобы функция f() была объявлена виртуальной в базовом классе. Измените функцию main() так, чтобы в ней выполнялось повышающее приведение типа и виртуальный вызов.
5. Измените пример Instrument3.cpp, добавив в него виртуальную функцию prepare(). Вызовите prepare() из tune().
6. Создайте иерархию наследования на базе класса Rodent («грызун») с производными классами Mouse («мышь»), Gerbil («песчанка»), Hamster («хомяк») и т. д. В базовом классе определите методы, общие для всех объектов семейства Rodent, и переопределите их в производных классах, чтобы они выполняли разные операции в зависимости от фактического типа. Создайте массив указателей на Rodent, заполните его разными объектами классов, производных от Rodent, вызовите методы базового класса и понаблюдайте за происходящим.
7. Измените упражнение 6 так, чтобы вместо массива указателей использовался класс vector<Rodent*>. Убедитесь в том, что память освобождается правильно.
8. В описанной выше иерархии Rodent объявите производный от Hamster класс BlueHamster («голубой хомяк» — да, они действительно существуют в природе; когда автор был маленьким, у него жил голубой хомячок). Переопределите методы базового класса и покажите, что программа, работающая с методами базового класса, не изменяется при добавлении нового типа.

9. В описанной выше иерархии Rodent объявите невиртуальный деструктор, создайте объект класса Hamster при помощи оператора new, повысьте указатель на него до указателя Rodent* и вызовите для указателя оператор delete. Убедитесь в том, что это не приводит к вызову всех деструкторов в иерархии. Сделайте деструктор виртуальным и продемонстрируйте, что в новом варианте программа работает правильно.
10. В описанной выше иерархии Rodent измените класс Rodent так, чтобы он стал чисто абстрактным базовым классом.
11. Смоделируйте систему управления воздушным движением, включающую базовый класс Aircraft («самолет») и различные производные типы. Создайте класс Tower («диспетчерская вышка») с контейнером vector<Aircraft*>, который посылает сообщения самолетам, находящимся под его управлением.
12. Создайте модель оранжереи — объявите различные типы, производные от Plant («растение»), и определите механизмы для ухода за «растениями».
13. В примере Early.cpp сделайте Pet чисто абстрактным базовым классом.
14. В примере AddingVirtuals.cpp объявите все функции класса Pet чисто виртуальными, но предоставьте определение для name(). Внесите необходимые исправления в Dog, используя определение name() из базового класса.
15. Напишите небольшую программу, демонстрирующую различия между вызовами виртуальной функции из обычной функции класса и из конструктора. Программа должна показывать, что эти два вызова приводят к разным результатам.
16. В примере VirtualsInDestructors.cpp создайте класс, производный от Derived, и переопределите в нем f() и деструктор. В функции main() создайте и повысьте объект нового типа, затем вызовите для него оператор delete.
17. В упражнении 16 включите в каждый деструктор вызов f(). Объясните происходящее.
18. Создайте базовый класс, содержащий переменную, и производный класс, в котором добавляется другая переменная. Напишите глобальную функцию, которая получает объект базового класса *по значению* и выводит размер этого объекта при помощи функции sizeof. В функции main() создайте объект производного класса, выведите его размер и вызовите вашу функцию. Объясните происходящее.
19. Напишите простую программу с вызовом виртуальной функции, сгенерируйте для нее ассемблерный код. Найдите фрагмент вызова виртуальной функции и проанализируйте его.
20. Создайте класс, содержащий две функции: виртуальную и невиртуальную. Создайте новый класс, производный от первого, создайте объект этого класса и выполните повышающее приведение к указателю на тип базового класса. Воспользуйтесь функцией clock() из библиотеки <ctime> (найдите ее описание в руководстве по библиотеке С) и сравните время виртуального и невиртуального вызовов. Чтобы различия были заметными, многократно вызовите каждую функцию в цикле хронометража.

21. В примере C14:Order.cpp добавьте виртуальную функцию в базовый класс макроса CLASS (пусть эта функция выводит какое-нибудь сообщение) и объявите деструктор виртуальным. Создайте объекты различных производных классов и повысьте их до базового класса. Убедитесь в том, что механизм виртуального вызова работает, а объекты правильно конструируются и уничтожаются.
22. Напишите класс с тремя перегруженными виртуальными функциями. Объявите новый класс производным от него и переопределите одну из функций. Создайте объект производного класса. Все ли функции базового класса могут вызываться через объект производного класса? Повысьте адрес объекта до базового класса. Удастся ли вам вызвать все три функции через базовый класс? А если удалить переопределение из производного класса?
23. Измените пример VariantReturn.cpp так, чтобы он работал как со ссылками, так и с указателями.
24. Как в примере Early.cpp узнать, какое связывание используется компилятором при вызове, раннее или позднее? Определите тип связывания для своего компилятора.
25. Создайте базовый класс с функцией clone(), которая возвращает указатель на *копию* текущего объекта. Создайте два производных класса, переопределяющих clone() для возвращения копий своих типов. В функции main() создайте объекты двух производных типов, повысьте их, вызовите clone() для каждого объекта и убедитесь в том, что копии относятся к правильным типам. Поэкспериментируйте с функцией clone() так, чтобы она возвращала базовый тип, затем попробуйте вернуть фактический производный тип. Можете ли вы представить ситуацию, в которой понадобится второй подход?
26. В примере OStackTest.cpp создайте собственный класс и используйте его вместе с классом Object для множественного наследования. Создайте объекты для сохранения в контейнере Stack. Протестируйте свой класс в main().
27. Добавьте в пример OperatorPolymorphism.cpp тип Tensor.
28. (Задача средней трудности) Создайте базовый класс X, не содержащий данных и конструкторов, но содержащий виртуальную функцию. Создайте класс Y, производный от X, но не содержащий явно определяемого конструктора. Сгенерируйте ассемблерный код, проанализируйте его и определите, создается ли и вызывается ли конструктор для X, и если это происходит, — что делает этот конструктор. Объясните полученный результат. X не имеет конструктора по умолчанию, тогда почему же компилятор не сообщает об ошибке?
29. (Задача средней трудности) Измените упражнение 28, написав для обоих классов конструкторы, чтобы каждый конструктор вызывал виртуальную функцию. Сгенерируйте ассемблерный код. Определите, где в каждом конструкторе присваивается значение VPTR. Использует ли ваш компилятор механизм виртуального вызова в конструкторе? Выясните, почему и в этом случае вызывается локальная версия функции.

30. (Задача повышенной трудности) Если бы для вызовов функции с передачей объекта по значению *не использовалось* раннее связывание, то виртуальный вызов мог бы привести к обращению к несуществующим частям класса. Возможно ли такое? Напишите программу с виртуальным вызовом и посмотрите, приведет ли это к сбою. Чтобы объяснить происходящее, внимательно проанализируйте, что происходит при передаче объекта по значению.
31. (Задача повышенной трудности) Точно определите, насколько больше времени тратится на виртуальный вызов функции. Для этого обратитесь к описанию ассемблера для вашего процессора или другому техническому руководству и сравните количество тактов процессора, затрачиваемых на простой и виртуальный вызовы.
32. Определите размер указателя VPTR для вашей реализации. Создайте новый класс множественным наследованием от двух классов, содержащих виртуальные функции. Сколько указателей VPTR содержит производный класс, один или два?
33. Создайте класс, содержащий переменные и виртуальные функции. Напишите функцию, которая ищет в памяти объект вашего класса и выводит его различные фрагменты. Для этого вам придется поэкспериментировать и определить, где в объекте хранятся указатели VPTR.
34. Представьте, что виртуальных функций не существует, и измените пример Instrument4.cpp так, чтобы виртуальные вызовы имитировались с применением оператора dynamic_cast. Объясните, чем плохо такое решение.
35. В примере StaticHierarchyNavigation.cpp вместо механизма RTTI языка C++ используйте свою собственную модель RTTI. Включите в базовый класс виртуальную функцию whatAmI() и тип enum type {Circles, Squares};.
36. Возьмите за основу пример PointerToMemberOperator.cpp из главы 12 и покажите, что полиморфизм работает и с указателями на члены классов, даже если функция operator->* была перегружена.

Знакомство с шаблонами



Наследование и композиция предоставляют средства для многократного использования объектного кода. *Шаблоны* C++ позволяют заново использовать *исходные тексты*.

Шаблоны C++ принадлежат к числу инструментов программирования общего назначения, но после их появления в языке программисты почти перестали применять однокоренные иерархии контейнерных классов (см. конец главы 15). Например, стандартные контейнеры и алгоритмы C++, рассматриваемые в первых двух главах второго тома книги, построены исключительно на базе шаблонов и относительно просты в использовании.

Эта глава не только представит читателю основные принципы использования шаблонов, но и познакомит его с контейнерами — основными компонентами объектно-ориентированного программирования, практически полностью реализованными в стандартной библиотеке C++. На страницах книги вы уже неоднократно встречали контейнеры Stash и Stack. Они были введены специально, чтобы вы привыкали к работе с контейнерами. В этой главе также будет представлена новая концепция *итераторов*. Хотя шаблоны идеально подходят для реализации контейнеров, во втором томе (в котором целая глава посвящена нетривиальному использованию контейнеров) будет показано, что у шаблонов находится множество других применений.

Контейнеры

Допустим, вы хотите создать стек, как мы уже неоднократно делали в книге. Ради простоты в стеке будут храниться только данные типа int:

```
//: C16:IntStack.cpp
// Простой стек для целых чисел
//{L} fibonacci
#include "fibonacci.h"
```

```
#include "../require.h"
#include <iostream>
using namespace std:

class IntStack {
  enum { ssize = 100 }:
  int stack[ssize]:
  int top:
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize. "Too many push()es"):
    stack[top++] = i:
  }
  int pop() {
    require(top > 0. "Too many pop()s"):
    return stack[--top]:
  }
}:

int main() {
  IntStack is:
  // Сохранение в стеке чисел Фибоначчи (просто ради интереса):
  for(int i = 0: i < 20: i++)
    is.push(fibonacci(i)):
  // Извлечение и вывод:
  for(int k = 0: k < 20: k++)
    cout << is.pop() << endl:
} ///:~
```

Класс IntStack является тривиальным примером стека с выталкиванием данных из стека вниз. Ради простоты стек создается с фиксированным размером, но вы также можете изменить его и организовать автоматическое расширение с выделением памяти из кучи (как это сделано в классе Stack, неоднократно рассматривавшемся в книге).

Функция main() заносит в стек несколько целых чисел, а затем снова извлекает их. Чтобы пример был более интересным, целые числа создаются функцией fibonacci(), генерирующей последовательность чисел Фибоначчи. Ниже приводится заголовочный файл с объявлением функции:

```
//: C16:fibonacci.h
// Генератор чисел Фибоначчи
int fibonacci(int n): ///:~
```

Реализация функции выглядит так:

```
//: C16:fibonacci.cpp {O}
#include "../require.h"

int fibonacci(int n) {
  const int sz = 100:
  require(n < sz):
  static int f[sz]: // Инициализируется нулями
  f[0] = f[1] = 1:
  // Поиск незаполненных элементов массива:
  int i:
  for(i = 0: i < sz: i++)
    if(f[i] == 0) break:
  while(i <= n) {
```

```
      f[i] = f[i-1] + f[i-2];
      i++;
    }
    return f[n];
} ///:~
```

Приведенная реализация достаточно эффективна, потому что она не генерирует числа более одного раза. В ней используется статический массив int и то обстоятельство, что компилятор инициализирует статический массив нулями. Первый цикл for перемещает индекс i в позицию первого элемента, равного нулю, а затем цикл while добавляет числа Фибоначчи в массив до тех пор, пока не будет достигнут нужный элемент. Если числа Фибоначчи вплоть до элемента n уже инициализированы, цикл while пропускается.

Потребность в контейнерах

Разумеется, стек целых чисел нельзя назвать жизненно важным инструментом. Потребность в контейнерах по-настоящему возникает при создании объектов в куче оператором new и их уничтожении оператором delete. В общем случае во время написания программы неизвестно, сколько объектов вам потребуется. Например, в системе управления воздушным движением было бы нежелательно ограничивать максимальное количество обслуживаемых самолетов; программа не должна завершаться аварийно только потому, что вы превысили некоторое пороговое значение. Система автоматизированного проектирования может поддерживать множество объектов, но только пользователь (во время выполнения программы) определяет, сколько именно объектов ему понадобится. При желании вы обнаружите множество подобных примеров в своих собственных задачах.

Программисты C, полагающиеся на виртуальную память для организации собственного «управления памятью», часто настороженно относятся к операторам new и delete и контейнерным классам. В C в подобных случаях иногда создавался огромный глобальный массив — заведомо больший, чем когда-либо может понадобиться в программе. Конечно, такие решения избавляют программиста от необходимости думать (или хотя бы знать о существовании функций malloc() и free()), но полученные программы плохо переносятся на другие платформы и в них кроются хитроумные ошибки.

Кроме того, работа с огромным глобальным массивом объектов в C++ может существенно замедлиться из-за издержек, связанных с вызовами конструктора и деструктора. Подход C++ работает гораздо лучше: когда вам потребуется объект, вы создаете его при помощи оператора new и сохраняете указатель в контейнере. Затем указатель извлекается из контейнера, и программа делает с ним что-нибудь полезное. В этом случае создаются только те объекты, которые действительно используются в программе. Кроме того, все данные инициализации часто недоступны при запуске программы. Оператор new позволяет отложить создание объектов до создания некоторых внешних условий.

Итак, в типичной ситуации контейнер содержит указатели на объекты. Программа создает объекты при помощи оператора new, сохраняет полученные указатели в контейнере (возможно — с повышающим приведением типа) и извлекает их позднее, когда с объектом нужно выполнить операцию. Этот подход позволяет создавать наиболее гибкие и универсальные программы.

Общие сведения о шаблонах

А теперь о проблеме. Мы имеем контейнер IntStack, в котором хранятся целые числа; но вам может потребоваться стек для хранения геометрических фигур, самолетов, растений или еще чего-нибудь. Каждый раз переделывать исходные тексты? Вряд ли это можно назвать разумным решением, тем более в языке, ориентированном на многократное использование кода. Нужен другой, лучший способ.

В данной ситуации существуют три варианта многократного использования исходных текстов: подход С, представленный для контраста; подход Smalltalk, значительно повлиявший на С++, и подход С++, в котором используются шаблоны.

Многократное использование кода в С и Smalltalk

Конечно, мы стараемся уйти от решения, принятого в С, — оно хлопотно, чревато ошибками и крайне неэлегантно. В этом варианте вы копируете исходный текст Stack и вручную вносите изменения, а вместе с ними — и новые ошибки. Такая методика недостаточно эффективна.

В языке Smalltalk (а по его примеру и в Java) принято простое и прямолинейное решение: чтобы многократно использовать код, требуется наследование. Для этого в каждом контейнерном классе хранятся элементы, производные от единого базового класса Object (как в примере, приведенном в конце главы 15). А поскольку библиотеке в Smalltalk отводится исключительно важная, основополагающая, роль, класс даже не приходится создавать «с нуля». Вы просто находите класс, по возможности близкий к нужному, создаете от него производный класс и вносите небольшие изменения. Конечно, такой подход чрезвычайно удобен — он избавляет программиста от лишней работы (вот почему для эффективного программирования на Smalltalk приходится тратить много времени на изучение библиотеки классов).

Но из этого также следует, что все классы Smalltalk в конечном счете оказываются связанными в единое иерархическое дерево, а значит, при создании нового класса вам неизбежно придется наследовать от одной из ветвей этого дерева. Большая часть дерева уже существует (это библиотека классов Smalltalk), а в корне дерева находится класс с именем Object — тот самый класс, который хранится во всех контейнерах Smalltalk.

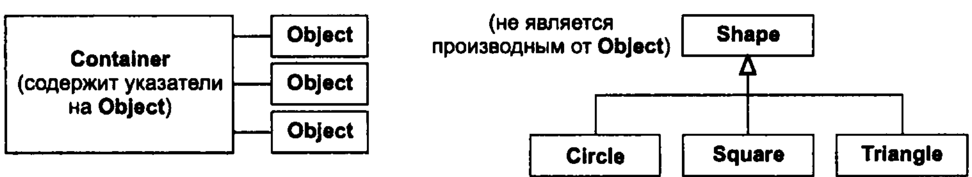
Решение получается довольно изящным, потому что оно означает, что любой класс в иерархии Smalltalk (и Java[1]) является производным от Object, поэтому любой контейнер может содержать объекты любых классов (включая класс самого контейнера). Такой тип однокоренной иерархии, основанной на фундаментальном типе-предке (часто называемом Object, как в Java), называется *объектно-базированной иерархией.* Возможно, вы уже слышали термин и решили, что он относится к какой-нибудь новой концепции ООП вроде полиморфизма. На самом деле он просто обозначает иерархию классов, в корне которой лежит класс Object (или другой аналогичный класс); объекты корневого класса хранятся в контейнерах.

Библиотека классов Smalltalk появилась гораздо раньше библиотеки С++ и широко использовалась на практике. Поскольку исходные компиляторы С++ *не име-*

[1] Кроме примитивных типов данных Java, которые для повышения эффективности не оформлялись как производные от Object.

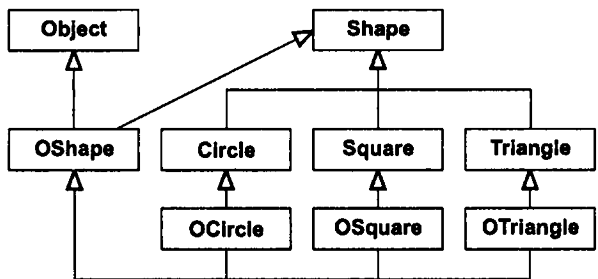
ли библиотеки контейнерных классов, казалось, что библиотеку Smalltalk стоит продублировать на C++. Такая попытка была предпринята на экспериментальном уровне в одной из ранних реализаций C++[1]. Библиотека содержала большой объем полезного кода, и многие программисты стали работать с ней. Однако при попытке использования контейнерных классов неожиданно обнаружилась проблема.

Дело в том, что в Smalltalk (и в большинстве других известных автору объектно-ориентированных языков) все классы автоматически наследуют от единой иерархии, но в C++ это правило не действует. Вы создаете замечательную объектно-базированную иерархию с контейнерными классами, а потом приобретаете набор классов, представляющих геометрические фигуры, самолеты и т. д. у другого производителя, который этой иерархией не пользовался (и, если уж на то пошло, само использование иерархии связано с непроизводительными затратами, от которых избавлены программисты C). Как связать отдельное дерево классов с контейнерным классом объектно-базированной иерархии? В наглядном представлении эта проблема выглядит так:



Из-за того, что C++ поддерживает множество независимых иерархий, объектно-базированная иерархия Smalltalk работает недостаточно хорошо.

Решение кажется очевидным: раз мы имеем дело с несколькими независимыми иерархиями, значит, нужно предусмотреть возможность наследования от нескольких классов. Множественное наследование должно решить проблему. Итак, мы делаем следующее (похожий пример приведен в конце главы 15):



Теперь класс OShape обладает характеристиками и поведением Shape, а так как он также является производным от Object, его можно хранить в контейнере Container. Дополнительные производные классы OCircle, OSquare и т. д. нужны для того, чтобы их можно было повысить до OShape с сохранением поведения. Как видите, иерархия быстро усложняется.

Разработчики компиляторов создавали и распространяли свои собственные объектно-базированные иерархии контейнерных классов, многие из которых были

[1] Библиотека OOPS автора Кейта Горлена (Keith Gorlen) во время его работы в NIH.

заменены шаблонными версиями. Некоторые полагают, что множественное наследование необходимо для решения общих задач программирования. Но, как будет показано во втором томе книги, оно порождает лишние сложности, которых лучше избегать (за исключением особых случаев).

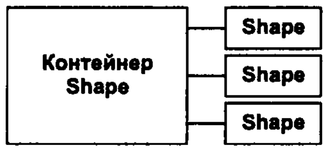
Решение с применением шаблонов

Хотя на концептуальном уровне объектно-базированные иерархии с множественным наследованием достаточно просты, работать с ними неудобно. В исходной версии своей книги[1] Страуструп привел решение, которое он считал предпочтительной альтернативой объектно-базированным иерархиям. Контейнерные классы создавались в виде больших препроцессорных макросов с аргументами, вместо которых подставлялся нужный тип. Контейнер для хранения определенного типа создавался парой макровызовов.

К сожалению, такой подход не соответствовал всему опыту программирования и существующей литературе по Smalltalk, к тому же он был неудобен. Короче говоря, он не прижился.

Тем временем Страуструп и группа проектировщиков C++ из лаборатории Bell изменили исходное решение с макросами, упростили его и передали его из зоны ответственности препроцессора компилятору. Новый механизм подстановки, названный *шаблоном*[2] (template), представляет принципиально новый способ многократного использования кода. Вместо многократного использования *объектного* кода, как при наследовании и композиции, шаблоны многократно используют *исходные тексты*. В контейнерах хранятся уже не объекты обобщенного базового класса Object, а некий тип, определяемый неизвестным параметром. При применении шаблона *компилятор* подставляет на место параметра фактический тип; происходит примерно то же самое, как в решении с макросами, но такой вариант понятнее и проще.

Вместо того чтобы «мучиться» с наследованием и композицией при применении контейнерного класса, вы создаете шаблонный контейнер и определяете его специализированную версию для своей задачи:



Компилятор выполняет всю работу, в результате вместо громоздкой иерархии наследования вы получаете именно тот контейнер, который необходим для решения вашей задачи. В C++ шаблоны реализуют концепцию *параметризованных типов*. У шаблонного решения есть и другое преимущество: даже программист, незнакомый с наследованием или не имеющий опыта работы с ним, сможет немедленно приступить к работе с контейнерными классами (так мы немедленно стали использовать контейнер vector в этой книге).

[1] «The C++ Programming Language», Bjarne Stroustrup (1st edition, Addison-Wesley, 1986).
[2] Вероятно, шаблоны создавались по образцу обобщенных конструкций языка ADA.

Синтаксис шаблонов

Ключевое слово template сообщает компилятору, что в следующем определении класса используется один или несколько неизвестных типов. В тот момент, когда компилятор генерирует код класса на основании шаблона, эти типы должны быть известны, чтобы компилятор мог подставить их на место параметров шаблона.

Для демонстрации синтаксиса рассмотрим небольшой пример, в котором создается массив с автоматической проверкой границ:

```
//: C16:Array.cpp
#include "../require.h"
#include <iostream>
using namespace std;

template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return A[index];
  }
};

int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
} ///:~
```

Класс Array выглядит вполне обычно, если не считать строки

```
template<class T>
```

Эта строка сообщает, что T является подставляемым параметром, который отражает имя типа. Внутри класса идентификатор T используется везде, где обычно указывается конкретный тип, хранимый в контейнере.

В Array присваивание *и* выборка элементов осуществляются одной функцией — перегруженной операторной функцией operator[ ]. Оператор возвращает ссылку, поэтому результат может находиться по обе стороны от оператора присваивания (то есть как l- и как r-значение). Если индекс выходит за границы массива, класс выводит сообщение об ошибке при помощи функции require(). Функция operator[ ] оформлена в виде подставляемой (inline), поэтому мы можем сначала убедиться в том, что границы массива не нарушаются, а затем удалить вызов require() из окончательной версии программы.

Функция main() показывает, как легко создаются контейнеры Array для хранения различных типов объектов. Встречая следующие определения, компилятор дважды расширяет шаблон (также говорят *создает экземпляр* шаблона) Array и соз-

дает два новых *сгенерированных класса*, которые можно условно обозначить Array_int и Array_float (разные компиляторы используют разные схемы формирования имен):

```
Array<int> ia;
Array<float> fa;
```

Эти классы работают точно так же, как при ручной подстановке типов, если не считать того, что они автоматически создаются компилятором при определении объектов ia и fa. Повторяющиеся определения классов либо обходятся компилятором, либо объединяются компоновщиком.

Определения неподставляемых функций

Конечно, многие определения функций классов должны оформляться как неподставляемые. В таких случаях компилятор должен получить объявление template до определения функции класса. Ниже приведен предыдущий пример, но на этот раз функция класса определяется как неподставляемая:

```
//: C16:Array2.cpp
// Определение неподставляемых функций в шаблонах
#include "../require.h"

template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
  require(index >= 0 && index < size,
    "Index out of range");
  return A[index];
}

int main() {
  Array<float> fa;
  fa[0] = 1.414;
} ///:~
```

За любыми ссылками на имя шаблона класса должен следовать список аргументов, как в вызове Array<T>::operator[ ]. Представьте, что во внутренней реализации к имени класса присоединяются аргументы из списка, в результате чего формируется уникальный идентификатор класса для каждого экземпляра шаблона.

Заголовочные файлы

Все объявления *и* определения, относящиеся к шаблону, лучше разместить в заголовочном файле. На первый взгляд кажется, что это противоречит стандартному правилу «Не размещай в заголовочных файлах ничего, что связано с выделением памяти» (из-за ошибок повторного определения на стадии компиляции), однако определения шаблонов занимают особое место. Любая конструкция с префиксом template<...> означает, что компилятор не выделяет память немедленно, а ждет специального распоряжения (то есть создания экземпляра шаблона), при этом в компиляторе и компоновщике реализован механизм исключения множественных оп-

ределений идентичного шаблона. По этой причине в заголовочный файл почти всегда включаются полные объявление и определение шаблона (это делается для простоты использования).

В отдельных случаях определения шаблонов приходится размещать в отдельном cpp-файле (например, чтобы все экземпляры шаблонов оказались в одном dll-файле Windows). В большинстве компиляторов предусмотрены соответствующие средства; если вы захотите ими воспользоваться, обращайтесь к документации по вашему компилятору.

Некоторые программисты считают, что вынесение всего исходного текста реализации в заголовочный файл нежелательно, поскольку дает возможность изменять (или просто красть) их программный код после покупки библиотеки. Такие опасения имеют основания, но здесь стоит спросить себя: что именно вы продаете, продукт или услугу? Если продукт, то вам следует приложить все усилия к его защите; вероятно, не стоит распространять исходные тексты, а лишь откомпилированный код. Но многие клиенты рассматривают программы как услугу, даже более того, как подписку на услугу. Клиент хочет пользоваться вашим опытом; он предпочитает, чтобы вы сами сопровождали свою программу, а он будет заниматься *своей* работой. По мнению автора, большинство клиентов рассматривают разработчика как ценный ресурс и не захотят рисковать хорошими отношениями с ним. А что до тех, кто предпочитает украсть, вместо того чтобы купить или сделать самостоятельно, — все равно они за вами не угонятся.

Класс IntStack в виде шаблона

Ниже приводятся контейнер и итератор из примера IntStack.cpp, реализованные в виде обобщенного контейнерного класса с применением шаблонов:

```
//: C16:StackTemplate.h
// Шаблон простого стека
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H
#include "../require.h"

template<class T>
class StackTemplate {
  enum { ssize = 100 };
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  int size() { return top; }
};
#endif // STACKTEMPLATE_H ///:~
```

Обратите внимание: шаблон делает некоторые предположения относительно хранящихся в нем объектов. Например, объект StackTemplate в функции push() пред-

полагает, что для T определен некий оператор присваивания. Можно сказать, что шаблон *подразумевает интерфейс* хранимых в нем типов.

То же самое можно сказать иначе: шаблон обеспечивает своего рода механизм *слабой типизации* в C++, который обычно считается языком с сильной типизацией. Если при сильной типизации считаются приемлемыми только объекты, относящиеся к конкретному типу, слабая типизация требует лишь *доступности* вызываемых функций для конкретного объекта. А значит, код со слабой типизацией может применяться к любым объектам, поддерживающим вызовы этих функций, и потому обладает гораздо большей гибкостью[1].

Ниже приведена обновленная версия программы для тестирования шаблона:

```
//: C16:StackTemplateTest.cpp
// Тестирование шаблона простого стека
//{L} fibonacci
#include "fibonacci.h"
#include "StackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
  ifstream in("StackTemplateTest.cpp");
  assure(in, "StackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  while(strings.size() > 0)
    cout << strings.pop() << endl;
} ///:~
```

Единственное различие связано с созданием объекта is. В списке аргументов шаблона указывается тип объектов, хранящихся в стеке и обслуживаемых итератором. Чтобы продемонстрировать универсальность шаблона, мы также создаем контейнер StackTemplate для хранения объектов string и тестируем его, читая строки из файла с исходным текстом программы.

Константы в шаблонах

Аргументы шаблонов не ограничиваются классами; в них также можно передавать встроенные типы. Значения этих аргументов превращаются в константы времени компиляции для указанного экземпляра шаблона. Таким аргументам даже можно присваивать значения по умолчанию. В следующем примере размер класса Array задается при создании экземпляра, но также предусмотрено значение по умолчанию:

[1] Все методы в Smalltalk и Python слабо типизованы, поэтому в этих языках шаблоны не нужны. В сущности, функциональность шаблонов предоставляется в них автоматически.

```
//: C16:Array3.cpp
// Встроенные типы как аргументы шаблонов
#include "../require.h"
#include <iostream>
using namespace std;

template<class T, int size = 100>
class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Index out of range");
    return array[index];
  }
  int length() const { return size; }
};

class Number {
  float f;
public:
  Number(float ff = 0.0f) : f(ff) {}
  Number& operator=(const Number& n) {
    f = n.f;
    return *this;
  }
  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const Number& x) {
      return os << x.f;
  }
};

template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};

int main() {
  Holder<Number> h;
  for(int i = 0; i < 20; i++)
    h[i] = i;
  for(int j = 0; j < 20; j++)
    cout << h[j] << endl;
} ///:~
```

Как и прежде, класс Array представляет массив, предотвращающий выход индекса за границы допустимых значений. Класс Holder очень похож на Array, но вместо внутреннего объекта типа Array в нем хранится указатель на Array. Этот указатель не инициализируется конструктором; инициализация откладывается до первого

обращения. Этот прием называется *отложенной инициализацией*; он применяется в случае, когда вы создаете много объектов, но работаете лишь с небольшой их частью и хотите сэкономить память.

Обратите внимание: значение size в обоих шаблонах не хранится внутри класса, но все функции класса используют его как обычную переменную.

Шаблоны для классов Stack и Stash

Проблема «принадлежности объектов» контейнерных классов Stash и Stack, часто встречающихся в книге, обусловлена тем фактом, что этим контейнерам не был известен точный тип хранящихся в них элементов. Лучшее, что у нас получилось, — это «стек объектов Object», приведенный в конце главы 15 (пример OStackTest.cpp).

Если прикладной программист не извлечет из контейнера все указатели на объекты, то контейнер должен суметь правильно удалить эти указатели. Другими словами, контейнеру «принадлежат» все объекты, которые не были удалены из него, и он отвечает за их уничтожение. Загвоздка в том, что для уничтожения нужно знать тип объекта, а для создания обобщенного контейнерного класса его знать *не нужно*. Тем не менее шаблоны позволяют писать программы, в которых, не зная типа объекта, можно легко создавать специализированные версии контейнера для любых типов. Специализации *знают* тип хранящихся в них объектов и могут вызвать правильный деструктор (в типичном случае с использованием полиморфизма предполагается наличие виртуального деструктора).

Для класса Stack эта задача решается относительно просто, поскольку все его функции логично оформить как подставляемые:

```
//: C16:TStack.h
// Контейнер Stack в виде шаблона
#ifndef TSTACK_H
#define TSTACK_H

template<class T>
class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop(){
    if(head == 0) return 0;
    T* result = head->data;
```

```
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // TSTACK_H ///:~
```

Сравните этот пример с примером OStack.h, и вы увидите, что класс Stack почти не изменился, а класс Object заменился аргументом T. Тестовая программа тоже осталась прежней, если не считать ликвидации множественного наследования от string и Object (да и самой необходимости в объекте Object). Исчез класс MyString, объявлявший о своем уничтожении, поэтому в программу включен маленький новый класс, который показывает, что контейнер Stack уничтожает свои объекты:

```
//: C16:TStackTest.cpp
//{T} TStackTest.cpp
#include "TStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class X {
public:
  virtual ~X() { cout << "~X " << endl; }
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Аргумент - имя файла
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack<string> textlines;
  string line;
  // Чтение файла и сохранение строк в стеке:
  while(getline(in, line))
    textlines.push(new string(line));
  // Извлечение строк из стека:
  string* s;
  for(int i = 0; i < 10; i++) {
    if((s = (string*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  } // Деструктор уничтожает оставшиеся строки.
  // Показать, что уничтожение выполнено правильно:
  Stack<X> xx;
  for(int j = 0; j < 10; j++)
    xx.push(new X);
} ///:~
```

Деструктор X объявлен виртуальным не потому, что так нужно в программе, а чтобы контейнер xx позднее мог использоваться для хранения объектов, производных от X.

Обратите внимание на то, как просто создаются разные специализации класса Stack для string и X. Шаблон предоставляет в ваше распоряжение все лучшее: простоту использования класса Stack в сочетании с правильной зачисткой.

Шаблон класса PStash для указателей

Переработать код класса PStash в шаблон сложнее, поскольку некоторые функции класса не могут оформляться как подставляемые. Тем не менее, как и в других шаблонах, определения этих функций все равно находятся в заголовочном файле (компилятор и компоновщик решают возможные проблемы с повторяющимися определениями). В целом программа похожа на знакомый класс PStash, но размер приращения (используемый функцией inflate()) превратился в параметр шаблона со значением по умолчанию, что позволяет выбрать размер приращения при создании экземпляра. С другой стороны, это означает, что приращение является фиксированной величиной, хотя, возможно, кто-то сочтет, что приращение должно изменяться во время жизненного цикла объекта:

```
//: C16:TPStash.h
#ifndef TPSTASH_H
#define TPSTASH_H

template<class T, int incr = 10>
class PStash {
  int quantity; // Количество элементов
  int next; // Следующий пустой элемент
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), next(0), storage(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const; // Выборка
  // Удаление ссылки из PStash:
  T* remove(int index);
  // Количество элементов в Stash:
  int count() const { return next; }
};

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if(next >= quantity)
    inflate(incr);
  storage[next++] = element;
  return(next - 1); // Индекс
}

// Принадлежность оставшихся указателей:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for(int i = 0; i < next; i++) {
    delete storage[i]; // Допустимо для null-указателей
    storage[i] = 0;    // Для надежности
  }
  delete []storage;
}

template<class T, int incr>
T* PStash<T, incr>::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] index negative");
  if(index >= next)
```

```
    return 0; // Чтобы показать на конец
  require(storage[index] != 0,
    "PStash::operator[] returned null pointer");
  // Получение указателя на нужный элемент:
  return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
  // operator[] проверяет действительность индекса:
  T* v = operator[](index);
  // "Удаление" указателя:
  if(v != 0) storage[index] = 0;
  return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int psz = sizeof(T*);
  T** st = new T*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Освобождение старого буфера
  storage = st; // Перевод указателя на новый буфер
}
#endif // TPSTASH_H ///:~
```

Приращение по умолчанию выбрано достаточно малым, чтобы гарантировать вызов функции inflate() и проверить ее работу.

Для проверки системы управления принадлежащими объектами в шаблонной версии PStash мы создаем следующий класс, который сообщает о создании и уничтожении своих экземпляров, а также гарантирует уничтожение всех созданных объектов. Класс AutoCounter разрешает только непосредственное создание своих объектов в стеке.

```
//: C16:AutoCounter.h
#ifndef AUTOCOUNTER_H
#define AUTOCOUNTER_H
#include "../require.h"
#include <iostream>
#include <set> // Контейнер из стандартной библиотеки C++
#include <string>

class AutoCounter {
  static int count;
  int id;
  class CleanupCheck {
    std::set<AutoCounter*> trace;
  public:
    void add(AutoCounter* ap) {
      trace.insert(ap);
    }
    void remove(AutoCounter* ap) {
      require(trace.erase(ap) == 1,
        "Attempt to delete AutoCounter twice");
    }
    ~CleanupCheck() {
```

```
      std::cout << "~CleanupCheck()"<< std::endl;
      require(trace.size() == 0,
       "All AutoCounter objects not cleaned up");
    }
  };
  static CleanupCheck verifier;
  AutoCounter() : id(count++) {
    verifier.add(this); // Register itself
    std::cout << "created[" << id << "]"
              << std::endl;
  }
  // Запрет присваивания и конструирования копий:
  AutoCounter(const AutoCounter&);
  void operator=(const AutoCounter&);
public:
  // Объекты могут создаваться только так:
  static AutoCounter* create() {
    return new AutoCounter();
  }
  ~AutoCounter() {
    std::cout << "destroying[" << id
              << "]" << std::endl;
    verifier.remove(this);
  }
  // Вывод объектов и указателей:
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter& ac){
    return os << "AutoCounter " << ac.id;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter* ac){
    return os << "AutoCounter " << ac->id;
  }
};
#endif // AUTOCOUNTER_H ///:~
```

Класс AutoCounter решает две задачи. Во-первых, он последовательно нумерует каждый экземпляр AutoCounter: номер генерируется при помощи статической переменной count и сохраняется в переменной id.

Во-вторых, статический экземпляр verifier вложенного класса CleanupCheck отслеживает все создаваемые и уничтожаемые объекты AutoCounter и выдает сообщение о неполной зачистке (то есть о возникновении утечек памяти). При этом используется класс set из стандартной библиотеки C++; перед нами прекрасный пример того, как хорошо спроектированный шаблон упрощает работу программиста (контейнеры стандартной библиотеки C++ будут описаны во втором томе книги).

Класс set параметризуется по типу содержащихся в нем объектов; в нашем случае создается экземпляр для хранения указателей на AutoCounter. Контейнер set («множество») может содержать не более одной копии каждого конкретного объекта; в функции add() это обстоятельство используется при вызове set::insert(). Возвращаемое значение функции insert() сообщает о попытках добавления ранее добавленного объекта. Впрочем, в данном случае в контейнере сохраняются адреса объектов, уникальность которых гарантируется языком C++.

В функции remove() указатель на AutoCounter исключается из контейнера set функцией set::erase(). Возвращаемое значение этой функции определяет количе-

ство удаленных элементов; в нашем случае оно должно быть равно либо 0, либо 1. Если значение равно 0, это означает, что объект уже был удален из контейнера set и мы пытаемся удалить его повторно (такая ситуация является ошибкой программирования, о которой следует сообщить при помощи функции require()).

Деструктор CleanupCheck напоследок проверяет, что размер set равен нулю (то есть что все объекты были удалены из контейнера). Если он отличен от нуля, значит, возникла утечка памяти, о которой также следует сообщить при помощи функции require().

Конструктор и деструктор класса AutoCounter регистрируют и разрывают связь своих объектов с объектом verifier. Обратите внимание: конструктор, копирующий конструктор и оператор присваивания объявлены закрытыми, поэтому объекты могут создаваться только статической функцией класса create(). Это гарантирует, что все объекты будут создаваться в куче, а объекту verifier не придется разбираться в присваиваниях и конструировании копий.

Поскольку все функции класса оформлены как подставляемые, файл реализации нужен только для хранения определений статических переменных:

```
//: C16:AutoCounter.cpp {O}
// Определение статических переменных класса
#include "AutoCounter.h"
AutoCounter::CleanupCheck AutoCounter::verifier;
int AutoCounter::count = 0;
///:~
```

«Вооружившись» классом AutoCounter, мы можем проверить, как работает класс PStash. Следующий пример не только демонстрирует, что деструктор класса PStash уничтожает все принадлежащие ему объекты, но и показывает, как класс AutoCounter обнаруживает неуничтоженные объекты:

```
//: C16:TPStashTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "TPStash.h"
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  PStash<AutoCounter> acStash;
  for(int i = 0; i < 10; i++)
    acStash.add(AutoCounter::create());
  cout << "Removing 5 manually:" << endl;
  for(int j = 0; j < 5; j++)
    delete acStash.remove(j);
  cout << "Remove two without deleting them:"
       << endl;
  // ... Чтобы программа сообщила об ошибке при уничтожении объектов
  cout << acStash.remove(5) << endl;
  cout << acStash.remove(6) << endl;
  cout << "The destructor cleans up the rest:"
       << endl;
  // Повторение теста из предыдущих глав:
  ifstream in("TPStashTest.cpp");
  assure(in, "TPStashTest.cpp");
  PStash<string> stringStash;
  string line;
```

```
  while(getline(in, line))
    stringStash.add(new string(line));
  // Вывод строк:
  for(int u = 0; stringStash[u]; u++)
    cout << "stringStash[" << u << "] = "
         << *stringStash[u] << endl;
} ///:~
```

После удаления из PStash элементов AutoCounter 5 и 6 за них отвечает вызывающая сторона. А так как вызывающая сторона не уничтожает эти объекты, возникает утечка памяти, которая обнаруживается объектом AutoCounter во время выполнения.

После запуска программы становится очевидно, что сообщение об ошибке содержит недостаточно полную информацию. Применяя методику, представленную в AutoCounter, для обнаружения утечки памяти в системе, можно организовать вывод более подробной информации о «бесхозных» объектах. Во втором томе будут представлены более совершенные средства решения этой задачи.

Управление принадлежностью

Но вернемся к проблеме принадлежности. Для контейнеров, хранящих объекты по значению, эта проблема несущественна, потому что хранящиеся в контейнере объекты явно принадлежат ему. Но если в контейнере содержатся указатели (что более характерно для C++, особенно в полиморфных иерархиях), вполне вероятно, что эти указатели используются в других местах. В таких случаях удалять извлекаемые объекты нельзя, потому что другие указатели в программе будут ссылаться на уничтоженный объект. Чтобы этого не произошло, нужно учитывать проблему принадлежности при проектировании и использовании контейнера.

С другой стороны, многие программы устроены гораздо проще, и проблема принадлежности для них неактуальна: в контейнере хранятся указатели на объекты, используемые только этим контейнером. В этом случае ситуация предельно проста: контейнеру принадлежат хранящиеся в нем объекты.

В оптимальном варианте решения проблемы принадлежности прикладному программисту предоставляется право выбора. Для этого часто используется аргумент конструктора, по умолчанию указывающий на принадлежность (простейший случай). Кроме того, в интерфейсе контейнера могут присутствовать функции get и set для получения и изменения режима принадлежности объектов. Режим принадлежности обычно влияет на процесс удаления, поэтому если у контейнера имеются функции удаления объектов, у них, как правило, тоже имеются флаги, управляющие уничтожением объектов. Возможно и другое решение — с каждым элементом контейнера связывается дополнительный флаг принадлежности, чтобы решение об уничтожении принималось для каждого элемента по отдельности. Получается некая разновидность подсчета ссылок, если не считать того, что количество ссылок на объект хранится в контейнере, а не в объекте.

```
//: C16:OwnerStack.h
// Стек с управлением принадлежностью объектов
#ifndef OWNERSTACK_H
#define OWNERSTACK_H

template<class T> class Stack {
```

```
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
  bool own;
public:
  Stack(bool own = true) : head(0), own(own) {}
  ~Stack();
  void push(T* dat) {
    head = new Link(dat,head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop();
  bool owns() const { return own; }
  void owns(bool newownership) {
    own = newownership;
  }
  // Автоматическое приведение типа: true, если стек не пуст.
  operator bool() const { return head != 0; }
};

template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

template<class T> Stack<T>::~Stack() {
  if(!own) return;
  while(head)
    delete pop();
}
#endif // OWNERSTACK_H ///:~
```

По умолчанию контейнер уничтожает свои объекты, но на его поведение можно повлиять либо изменением аргумента конструктора, либо при помощи функций класса owns().

Как и для большинства шаблонов, с которыми вам придется иметь дело, вся реализация хранится в заголовочном файле. Небольшая тестовая программа показывает, как организуется управление объектами:

```
//: C16:OwnerStackTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "OwnerStack.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
```

```
    Stack<AutoCounter> ac; // Управление принадлежностью объектов
    Stack<AutoCounter> ac2(false); // Отключение
    AutoCounter* ap;
    for(int i = 0; i < 10; i++) {
      ap = AutoCounter::create();
      ac.push(ap);
      if(i % 2 == 0)
        ac2.push(ap);
    }
    while(ac2)
      cout << ac2.pop() << endl;
    // Уничтожать объекты не нужно,
    // потому что все они "принадлежат" контейнеру ac
  } ///:~
```

Объекты, хранящиеся в контейнере ac2, ему не принадлежат. Таким образом, ac является «главным» контейнером, на который возлагается ответственность за уничтожение объектов. Если где-то в середине жизненного цикла контейнера вам захочется изменить режим принадлежности объектов контейнеру, воспользуйтесь функцией owns().

В принципе, такие отношения принадлежности можно изменить, чтобы решения принимались на уровне отдельных объектов, но тогда решение проблемы принадлежности становится сложнее самой проблемы.

Хранение объектов по значению

Без шаблонов копирование объектов внутри универсальных контейнеров является весьма сложной задачей. При наличии шаблонов все относительно несложно — просто укажите, что в контейнере хранятся объекты, а не указатели:

```
//: C16:ValueStack.h
// Хранение объектов по значению в стеке
#ifndef VALUESTACK_H
#define VALUESTACK_H
#include "../require.h"

template<class T, int ssize = 100>
class Stack {
  // Конструктор по умолчанию выполняет инициализацию
  // для каждого элемента в массиве:
  T stack[ssize];
  int top;
public:
  Stack() : top(0) {}
  // Копирующий конструктор копирует объект в массив:
  void push(const T& x) {
    require(top < ssize, "Too many push()es");
    stack[top++] = x;
  }
  T peek() const { return stack[top]; }
  // Извлеченный объект продолжает существовать;
  // просто он становится недоступным:
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
```

```
  }
};
#endif // VALUESTACK_H ///:~
```

Вся основная работа выполняется копирующим конструктором при передаче и возврате объектов по значению. В функции push() объекты сохраняются в массиве Stack функцией T::operator=. С целью проверки класс SelfCounter отслеживает создание объектов и конструирование копий:

```
//: C16:SelfCounter.h
#ifndef SELFCOUNTER_H
#define SELFCOUNTER_H
#include "ValueStack.h"
#include <iostream>

class SelfCounter {
  static int counter;
  int id;
public:
  SelfCounter() : id(counter++) {
    std::cout << "Created: " << id << std::endl;
  }
  SelfCounter(const SelfCounter& rv) : id(rv.id){
    std::cout << "Copied: " << id << std::endl;
  }
  SelfCounter operator=(const SelfCounter& rv) {
    std::cout << "Assigned " << rv.id << " to "
              << id << std::endl;
    return *this;
  }
  ~SelfCounter() {
    std::cout << "Destroyed: "<< id << std::endl;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const SelfCounter& sc){
    return os << "SelfCounter: " << sc.id;
  }
};
#endif // SELFCOUNTER_H ///:~

//: C16:SelfCounter.cpp {O}
#include "SelfCounter.h"
int SelfCounter::counter = 0; ///:~

//: C16:ValueStackTest.cpp
//{L} SelfCounter
#include "ValueStack.h"
#include "SelfCounter.h"
#include <iostream>
using namespace std;

int main() {
  Stack<SelfCounter> sc;
  for(int i = 0; i < 10; i++)
    sc.push(SelfCounter());
  // Выполнять peek() можно, результатом является временный обьект:
  cout << sc.peek() << endl;
  for(int k = 0; k < 10; k++)
    cout << sc.pop() << endl;
} ///:~
```

При создании контейнера **Stack** для каждого объекта в массиве вызывается конструктор по умолчанию класса элемента. Сначала без видимых причин создаются 100 объектов **SelfCounter**, но это всего лишь инициализация массива. Операция обходится недешево, но в таких простых архитектурах без обходных путей иначе нельзя. А если обобщить **Stack** и разрешить динамическое увеличение размеров, возникнет еще более сложная ситуация, потому что в приведенной выше реализации для этого потребуется создавать новый массив большего размера, копировать старый массив в новый и уничтожать старый массив (кстати, именно это делает класс **vector** из стандартной библиотеки C++).

Знакомство с итераторами

Итератором называется объект, который перебирает содержимое контейнера и в любой момент времени ассоциируется с одним элементом, не предоставляя прямого доступа к реализации контейнера. Итераторы обеспечивают стандартный механизм обращения к элементам независимо от того, поддерживает ли контейнер прямой доступ к элементам. Итераторы чаще всего используются с контейнерными классами, они занимают важное место в архитектуре стандартных контейнеров C++ и их практическом применении (эта тема будет рассматриваться во втором томе книги). Итераторы также относятся к числу *архитектурных идиом*, которым во втором томе посвящена целая глава.

Во многих отношениях итератор может рассматриваться как «умный» указатель. И действительно, итераторы поддерживают многие операции, характерные для указателей. Но в отличие от указателей, при проектировании итераторов особое значение придавалось вопросам безопасности, поэтому неприятности типа выхода за пределы массива с ними случаются гораздо реже (а если и случаются, то выявляются гораздо быстрее).

Вернемся к первому примеру этой главы. Вот как он выглядит при добавлении простого итератора:

```
//: C16:IterIntStack.cpp
// Простой целочисленный стек с итераторами.
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
```

```
  friend class IntStackIter;
};

// Итераторы похожи на "умные" указатели:
class IntStackIter {
  IntStack& s;
  int index;
public:
  IntStackIter(IntStack& is) : s(is), index(0) {}
  int operator++() { // Префиксная версия
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[++index];
  }
  int operator++(int) { // Постфиксная версия
    require(index < s.top,
      "iterator moved out of range");
    return s.stack[index++];
  }
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Перебор с использованием итератора:
  IntStackIter it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
} ///:~
```

Итератор IntStackIter рассчитан на работу только с классом IntStack. Обратите внимание: класс IntStackIter является дружественным (friend) по отношению к IntStack и поэтому получает доступ ко всем закрытым членам IntStack.

Итератор IntStackIter, по аналогии с указателями, перебирает содержимое IntStack и производит выборку отдельных элементов. В нашем простом примере IntStackIter может перемещаться только вперед (для чего используются префиксная и постфиксная формы оператора ++). Тем не менее возможности определения операторов не ограничиваются ничем, кроме естественных требований контейнера, с которым он работает. Ничто не мешает итератору (в рамках базового контейнера) произвольным образом перемещаться по своему контейнеру и изменять содержащиеся в нем элементы.

Обычно итератор создается конструктором, который ассоциирует его с одним контейнерным объектом, и этот итератор не переводится на другой контейнер на протяжении своего жизненного цикла (итераторы обычно компактны и быстры, поэтому при необходимости вы можете легко создать другой итератор).

При помощи итератора можно перебирать элементы, не извлекая их из стека, подобно тому как указатель может использоваться для перебора элементов массива. Однако итератор знает базовую структуру стека и умеет перебирать элементы. Хотя на первый взгляд перебор сводится к простому «увеличению указателя», на самом деле выполняются более сложные действия. В этом заключается ключевая особенность итераторов: они абстрагируют сложный процесс перемещения между элементами контейнера в нечто, напоминающее простой указатель.

Это делается для того, чтобы *все* итераторы в программе имели единый интерфейс, а для программного кода, работающего с итератором, было бы неважно, с каким типом контейнера связан итератор, — операции с итераторами должны выполняться одинаково, поэтому конкретный контейнер не важен. При таком подходе появляется возможность написания в большей степени обобщенного кода. Все контейнеры и алгоритмы стандартной библиотеки C++ используют это свойство итераторов.

Для дальнейшей унификации было бы удобно связать с каждым контейнером свой класс с именем iterator, но это вызовет проблемы с именами. Задача решается включением в каждый контейнер вложенного класса iterator (обратите внимание: в данном случае iterator начинается со строчной буквы в соответствии со стилем стандартной библиотеки C++). Ниже приведен класс IterIntStack.cpp с вложенным классом iterator:

```
//: C16:NestedIterator.cpp
// Вложенное определение итератора в контейнере
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  class iterator;
  friend class iterator;
  class iterator {
    IntStack& s;
    int index;
  public:
    iterator(IntStack& is) : s(is), index(0) {}
    // Для создания "конечного" итератора:
    iterator(IntStack& is, bool)
      : s(is), index(s.top) {}
    int current() const { return s.stack[index]; }
    int operator++() { // Префиксная версия
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[++index];
    }
    int operator++(int) { // Постфиксная версия
      require(index < s.top,
        "iterator moved out of range");
```

```
      return s.stack[index++];
    }
    // Смещение итератора вперед
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        "IntStack::iterator::operator+=() "
        "tried to move out of bounds");
      index += amount;
      return *this;
    }
    // Проверка выхода в конец контейнера:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
    friend ostream&
    operator<<(ostream& os, const iterator& it) {
      return os << it.current();
    }
  };
  iterator begin() { return iterator(*this); }
  // Создание "конечного итератора":
  iterator end() { return iterator(*this, true);}
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  cout << "Traverse the whole IntStack\n";
  IntStack::iterator it = is.begin();
  while(it != is.end())
    cout << it++ << endl;
  cout << "Traverse a portion of the IntStack\n";
  IntStack::iterator
    start = is.begin(), end = is.begin();
  start += 5, end += 15;
  cout << "start = " << start << endl;
  cout << "end = " << end << endl;
  while(start != end)
    cout << start++ << endl;
} ///:~
```

Вложенные дружественные классы создаются по определенным правилам: сначала вы объявляете имя класса, затем объявляете его дружественным и только потом определяете класс, иначе компилятор запутается в объявлениях.

В итератор из нашего примера были добавлены новые возможности. Функция current() возвращает текущий элемент контейнера, связанный с итератором. Итератор можно «перевести» на произвольное количество элементов вперед оператором +=. Также в класс включены два перегруженных оператора: == и !=, сравнивающих один итератор с другим. Операторы могут сравнить два любых итератора IntStack::iterator, но прежде всего они предназначены для проверки того, не вышел ли итератор за конец последовательности элементов (как это делают все «настоящие» итераторы стандартной библиотеки C++). Дело в том, что два итератора

определяют интервал, в который входит элемент, определяемый первым итератором, но *не входит* последний элемент, определяемый вторым итератором. Следовательно, перебор интервала, определяемого двумя итераторами, осуществляется примерно так:

```
while(start != end)
  cout << start++ << endl;
```

Здесь start и end — два итератора, образующие интервал. Обратите внимание: конечный итератор end не разыменовывается и используется лишь для проверки факта достижения конца последовательности. Таким образом, он представляет «следующую позицию за последним элементом».

На практике обычно перебирается полная последовательность элементов контейнера, поэтому контейнеру необходим способ получения итераторов, обозначающих границы этой последовательности. В стандартной библиотеке C++ эти итераторы возвращаются функциями класса контейнера begin() и end(). Функция begin() использует первый конструктор iterator, который по умолчанию ассоциирует итератор с началом контейнера (для стека это будет первый элемент). Но для создания конечного итератора необходим второй конструктор, задействованный функцией end(). «Конец последовательности» означает вершину стека, поскольку top всегда указывает на следующую доступную, но не используемую позицию в стеке. Второй конструктор iterator получает второй, фиктивный, аргумент типа bool, по которому различаются два конструктора.

Для заполнения контейнера IntStack в функции main() снова применяются числа Фибоначчи, затем итераторы используются для перемещения по всему контейнеру IntStack и ограниченному подынтервалу.

Естественно, следующим шагом должно быть обобщение кода и его параметризация по типу элементов контейнера. В новом варианте стек не ограничивается типом int и может содержать элементы произвольного типа:

```
//: C16:IterStackTemplate.h
// Простой шаблон стека с вложенным итератором
#ifndef ITERSTACKTEMPLATE_H
#define ITERSTACKTEMPLATE_H
#include "../require.h"
#include <iostream>

template<class T, int ssize = 100>
class StackTemplate {
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  class iterator; // Необходимо объявление
  friend class iterator; // Дружественный класс
  class iterator { // Теперь можно определять
    StackTemplate& s;
```

```
    int index:
  public:
    iterator(StackTemplate& st): s(st).index(0){}
    // Для создания "конечного" итератора:
    iterator(StackTemplate& st, bool)
      : s(st), index(s.top) {}
    T operator*() const { return s.stack[index];}
    T operator++() { // Префиксная версия
      require(index < s.top,
        "iterator moved out of range"):
      return s.stack[++index]:
    }
    T operator++(int) { // Постфиксная версия
      require(index < s.top,
        "iterator moved out of range"):
      return s.stack[index++]:
    }
    // Смещение итератора вперед
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        " StackTemplate::iterator::operator+=() "
        "tried to move out of bounds"):
      index += amount:
      return *this:
    }
    // Проверка выхода в конец контейнера:
    bool operator==(const iterator& rv) const {
      return index == rv.index:
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index:
    }
    friend std::ostream& operator<<(
      std::ostream& os, const iterator& it) {
      return os << *it:
    }
  }:
  iterator begin() { return iterator(*this): }
  // Создание "конечного итератора":
  iterator end() { return iterator(*this, true):}
}:
#endif // ITERSTACKTEMPLATE_H ///:~
```

Как видите, переход от обычного класса к шаблону довольно прозрачен. Такой подход — создание и отладка обычного класса с последующим преобразованием в шаблон — обычно проще, чем создание шаблона «с нуля».

Обратите внимание на следующую строку:

```
friend iterator: // Дружественный класс
```

Вместо этой простой строки в программу включается строка

```
friend class iterator: // Дружественный класс
```

Это важно, потому что имя iterator уже попало в область видимости при включении заголовочного файла.

Вместо функции current() выборка текущего элемента осуществляется оператором *, что усиливает сходство итератора с указателем. Такая практика считается общепринятой.

Ниже приведен обновленный вариант тестовой программы для шаблона:

```
//: C16:IterStackTemplateTest.cpp
//{L} fibonacci
#include "fibonacci.h"
#include "IterStackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Перебор элементов с использованием итератора:
  cout << "Traverse the whole StackTemplate\n";
  StackTemplate<int>::iterator it = is.begin();
  while(it != is.end())
    cout << it++ << endl;
  cout << "Traverse a portion\n";
  StackTemplate<int>::iterator
    start = is.begin(), end = is.begin();
  start += 5, end += 15;
  cout << "start = " << start << endl;
  cout << "end = " << end << endl;
  while(start != end)
    cout << start++ << endl;
  ifstream in("IterStackTemplateTest.cpp");
  assure(in, "IterStackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  StackTemplate<string>::iterator
    sb = strings.begin(), se = strings.end();
  while(sb != se)
    cout << sb++ << endl;
} ///:~
```

Сначала итератор просто перебирает все содержимое контейнера от начала до конца (демонстрируя правильную работу конечного итератора). Второй тест показывает, что итераторы позволяют легко определять интервалы элементов (эта концепция почти постоянно используется контейнерами и итераторами стандартной библиотеки C++). Перегруженный оператор += перемещает итераторы start и end в середину интервала элементов is, после чего эти элементы выводятся в выходной поток. При выводе конечный итератор *не включается* в интервал, поэтому он может находиться в следующей позиции после фактического конца интервала, однако конечный итератор не должен разыменовываться в программе, поскольку это приведет к разыменованию null-указателя (StackTemplate::iterator содержит код проверки границ контейнера, но в контейнерах и итераторах стандартной библиотеки C++ такой код отсутствует по соображениям эффективности, поэтому вы должны быть внимательны).

Чтобы доказать, что StackTemplate работает с объектами классов, мы создаем экземпляр шаблона для класса string, заполняем его строками исходного текста программы и выводим их.

Класс Stack с итераторами

Давайте повторим процедуру параметризации для динамически наращиваемого класса **Stack**, который использовался в примерах книги. Вот как выглядит класс **Stack** с добавлением вложенного итератора:

```
//: C16:TStack2.h
// Параметризованный класс Stack с вложенным итератором
#ifndef TSTACK2_H
#define TSTACK2_H

template<class T> class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack();
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop();
  // Вложенный класс итератора:
  class iterator; // Необходимо объявление
  friend class iterator; // Дружественный класс
  class iterator { // Теперь можно определять
    Stack::Link* p;
  public:
    iterator(const Stack<T>& tl) : p(tl.head) {}
    // Копирующий конструктор:
    iterator(const iterator& tl) : p(tl.p) {}
    // Конечный итератор:
    iterator() : p(0) {}
    // operator++ возвращает логический признак конца последовательности:
    bool operator++() {
      if(p->next)
        p = p->next;
      else p = 0; // Признак конца списка
      return bool(p);
    }
    bool operator++(int) { return operator++(); }
    T* current() const {
      if(!p) return 0;
      return p->data;
    }
    // Оператор разыменования указателя:
    T* operator->() const {
      require(p != 0,
        "PStack::iterator::operator->returns 0");
      return current();
    }
    T* operator*() const { return current(); }
```

```
    // Преобразование к bool для условной проверки:
    operator bool() const { return bool(p); }
    // Сравнение для проверки выхода в конец контейнера:
    bool operator==(const iterator&) const {
      return p == 0;
    }
    bool operator!=(const iterator&) const {
      return p != 0;
    }
  };
  iterator begin() const {
    return iterator(*this);
  }
  iterator end() const { return iterator(); }
};

template<class T> Stack<T>::~Stack() {
  while(head)
    delete pop();
}

template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}
#endif // TSTACK2_H ///:~
```

Также обратите внимание, что объекты теперь принадлежат контейнеру, потому что классу известен их точный тип (или, по крайней мере, базовый тип; при использовании виртуальных деструкторов достаточно и этого). По умолчанию контейнер уничтожает свои объекты, но за указатели, извлеченные функцией **pop()**, отвечает вызывающая сторона.

Итератор прост и занимает минимальный объем физической памяти, равный размеру одного указателя. При создании итератор устанавливается в начало связанного списка и в дальнейшем может перемещаться только вперед. Если потребуется снова вернуться к началу списка, вы создаете новый итератор, а для перехода к определенной позиции списка новый итератор создается на базе существующего, установленного в нужную позицию (с использованием копирующего конструктора).

Для вызова функций объекта, на который ссылается итератор, можно воспользоваться функцией **current()**, оператором ***** или оператором **->** (стандартная картина при работе с итераторами). Реализация последнего оператора *выглядит* идентичной **current()**, потому что она возвращает указатель на текущий объект, но на самом деле отличается, поскольку оператор **->** выполняет дополнительное разыменование на дополнительных уровнях (см. главу 12).

Класс **iterator** построен по схеме, представленной выше; он создается вложенным внутри контейнерного класса, содержит конструкторы для создания итератора, указывающего на элемент контейнера, и «конечного» итератора, для получения которых в классе контейнера определяются методы **begin()** и **end()**. После знакомства со стандартной библиотекой C++ вы убедитесь, что задействованные имена **iterator**, **begin()** и **end()** позаимствованы из стандартных контейнерных клас-

сов. А в конце этой главы показано, что такой выбор имен позволяет работать с пользовательскими контейнерными классами так, словно они входят в стандартную библиотеку C++.

Вся реализация содержится в заголовочном файле, поэтому отдельный срр-файл не требуется. Ниже приведена небольшая тестовая программа для экспериментов с итератором:

```
//: C16:TStack2Test.cpp
#include "TStack2.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  ifstream file("TStack2Test.cpp");
  assure(file, "TStack2Test.cpp");
  Stack<string> textlines;
  // Чтение файла и загрузка строк в Stack:
  string line;
  while(getline(file, line))
    textlines.push(new string(line));
  int i = 0;
  // Использование итератора для вывода строк из списка:
  Stack<string>::iterator it = textlines.begin();
  Stack<string>::iterator* it2 = 0;
  while(it != textlines.end()) {
    cout << it->c_str() << endl;
    it++;
    if(++i == 10) // Создание итератора для 10-й строки
      it2 = new Stack<string>::iterator(it);
  }
  cout << (*it2)->c_str() << endl;
  delete it2;
} ///:~
```

Шаблон Stack специализируется для объектов string и заполняется строками файла. Затем программа создает итератор и использует его для перебора элементов. Чтобы запомнить десятую строку, мы создаем новый итератор на базе существующего, применяя копирующий конструктор; затем эта строка выводится, а динамически созданный итератор уничтожается. Здесь механизм динамического создания объекта определяет его жизненный цикл.

Класс PStash с итераторами

В большинстве контейнерных классов имеет смысл определить свои итераторы. Вот как выглядит класс PStash после добавления итератора:

```
//: C16:TPStash2.h
// Параметризованный класс PStash с вложенным итератором
#ifndef TPSTASH2_H
#define TPSTASH2_H
#include "../require.h"
#include <cstdlib>

template<class T, int incr = 20>
```

```
class PStash {
  int quantity;
  int next;
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), storage(0), next(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const;
  T* remove(int index);
  int count() const { return next; }
  // Вложенный класс итератора:
  class iterator; // Необходимо объявление
  friend class iterator; // Дружественный класс
  class iterator { // Теперь можно определять
    PStash& ps;
    int index;
  public:
    iterator(PStash& pStash)
      : ps(pStash), index(0) {}
    // Для создания "конечного итератора"
    iterator(PStash& pStash, bool)
      : ps(pStash), index(ps.next) {}
    // Копирующий конструктор:
    iterator(const iterator& rv)
      : ps(rv.ps), index(rv.index) {}
    iterator& operator=(const iterator& rv) {
      ps = rv.ps;
      index = rv.index;
      return *this;
    }
    iterator& operator++() {
      require(++index <= ps.next,
        "PStash::iterator::operator++ "
        "moves index out of bounds");
      return *this;
    }
    iterator& operator++(int) {
      return operator++();
    }
    iterator& operator--() {
      require(--index >= 0,
        "PStash::iterator::operator-- "
        "moves index out of bounds");
      return *this;
    }
    iterator& operator--(int) {
      return operator--();
    }
    // Смещение итератора в прямом и обратном направлениях:
    iterator& operator+=(int amount) {
      require(index + amount < ps.next &&
        index + amount >= 0,
        "PStash::iterator::operator+= "
        "attempt to index out of bounds");
      index += amount;
      return *this;
    }
```

```
    iterator& operator-=(int amount) {
      require(index - amount < ps.next &&
        index - amount >= 0,
        "PStash::iterator::operator-= "
        "attempt to index out of bounds");
      index -= amount;
      return *this;
    }
    // Создание нового итератора с перемещением вперед
    iterator operator+(int amount) const {
      iterator ret(*this);
      ret += amount; // op+= обеспечивает проверку границ
      return ret;
    }
    T* current() const {
      return ps.storage[index];
    }
    T* operator*() const { return current(); }
    T* operator->() const {
      require(ps.storage[index] != 0,
        "PStash::iterator::operator->returns 0");
      return current();
    }
    // Удаление текущего элемента:
    T* remove(){
      return ps.remove(index);
    }
    // Сравнение для проверки выхода в конец контейнера:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
  };
  iterator begin() { return iterator(*this); }
  iterator end() { return iterator(*this, true);}
};

// Уничтожение хранящихся объектов:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for(int i = 0; i < next; i++) {
    delete storage[i]; // Допустимо для null-указателей
    storage[i] = 0; // Для надежности
  }
  delete []storage;
}

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if(next >= quantity)
    inflate();
  storage[next++] = element;
  return(next - 1); // Индекс
}

template<class T, int incr> inline
T* PStash<T, incr>::operator[](int index) const {
```

```
    require(index >= 0,
      "PStash::operator[] index negative");
    if(index >= next)
      return 0; // To indicate the end
    require(storage[index] != 0,
      "PStash::operator[] returned null pointer");
    return storage[index];
  }

  template<class T, int incr>
  T* PStash<T, incr>::remove(int index) {
    // operator[] проверяет действительность индекса:
    T* v = operator[](index);
    // "Удаление" указателя:
    storage[index] = 0;
    return v;
  }

  template<class T, int incr>
  void PStash<T, incr>::inflate(int increase) {
    const int tsz = sizeof(T*);
    T** st = new T*[quantity + increase];
    memset(st, 0, (quantity + increase) * tsz);
    memcpy(st, storage, quantity * tsz);
    quantity += increase;
    delete []storage; // Освобождение старого буфера
    storage = st; // Перевод указателя на новый буфер
  }
  #endif // TPSTASH2_H ///:~
```

Большая часть файла представляет собой довольно прямолинейное преобразование класса PStash и вложенного итератора в шаблоны. Впрочем, в этой версии операторы возвращают ссылки на текущие итераторы; такой подход получается более типичным и гибким.

Деструктор вызывает delete для всех хранящихся в контейнере указателей, а поскольку тип хранится в шаблоне, элементы будут должным образом уничтожены. Если в контейнере хранятся указатели на тип базового класса, то в этом типе должен быть определен виртуальный деструктор, обеспечивающий правильное уничтожение объектов производных классов, адреса которых прошли повышающее приведение типа перед сохранением в контейнере.

Поведение итератора PStash::iterator соответствует базовой модели итераторов, в которой итератор на протяжении всего жизненного цикла остается связанным только с одним контейнером. Кроме того, копирующий конструктор позволяет создать новый итератор, установленный в ту же позицию, что и существующий итератор, на базе которого он создавался; фактически это позволяет помечать отдельные позиции контейнера. Операторные функции operator+= и operator-= перемещают итератор на заданное число позиций, проверяя целостность границ контейнера. Перегруженные операторы ++ и -- перемещают итератор на одну позицию. Оператор + создает новый итератор, перемещенный вперед на количество позиций, определяемое вторым слагаемым. Как и в предыдущем примере, операторы разыменования используются для работы с элементом, на который ссылается итератор, а текущий объект уничтожается вызовом контейнерной версии функции remove().

Для создания конечных итераторов используются те же функции, что и в предыдущем примере (в стиле контейнеров стандартной библиотеки C++): второй конструктор, функция **end()** класса контейнера, а также операторные функции **operator==** и **operator!=**.

Следующая программа создает и тестирует две специализации класса **Stash**: для нового класса **Int**, который выводит сообщения о своем конструировании и уничтожении, и для стандартного библиотечного класса **string**.

```
//: C16:TPStash2Test.cpp
#include "TPStash2.h"
#include "../require.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Int {
  int i;
public:
  Int(int ii = 0) : i(ii) {
    cout << ">" << i << ' ';
  }
  ~Int() { cout << "~" << i << ' '; }
  operator int() const { return i; }
  friend ostream&
    operator<<(ostream& os, const Int& x) {
      return os << "Int: " << x.i;
  }
  friend ostream&
    operator<<(ostream& os, const Int* x) {
      return os << "Int: " << x->i;
  }
};

int main() {
  { // Для обеспечения вызова деструктора
    PStash<Int> ints;
    for(int i = 0; i < 30; i++)
      ints.add(new Int(i));
    cout << endl;
    PStash<Int>::iterator it = ints.begin();
    it += 5;
    PStash<Int>::iterator it2 = it + 10;
    for(; it != it2; it++)
      delete it.remove(); // Стандартное удаление
    cout << endl;
    for(it = ints.begin();it != ints.end();it++)
      if(*it) // Remove() создает "дыры"
        cout << *it << endl;
  } // Здесь вызывается деструктор "ints"
  cout << "\n-------------------\n";
  ifstream in("TPStash2Test.cpp");
  assure(in, "TPStash2Test.cpp");
  // Специализация для string:
  PStash<string> strings;
  string line;
  while(getline(in, line))
    strings.add(new string(line));
```

```
  PStash<string>::iterator sit = strings.begin();
  for(; sit != strings.end(); sit++)
    cout << **sit << endl;
  sit = strings.begin();
  int n = 26;
  sit += n;
  for(; sit != strings.end(); sit++)
    cout << n++ << ": " << **sit << endl;
} ///:~
```

Для удобства в Int включены потоковые операторы << для Int& и Int*.

Первый фрагмент кода main() заключен в фигурные скобки, чтобы при выходе из блока экземпляр PStash<Int> был уничтожен с автоматическим уничтожением объектов. Мы вручную удаляем и уничтожаем интервал элементов, чтобы показать, что PStash удалит все остальное.

Для обоих экземпляров PStash программа создает итератор и использует его для перебора элементов. Обратите внимание на элегантность этих конструкций; нам совершенно не приходится беспокоиться о подробностях реализации при работе с массивом. Объектам контейнера и итератора нужно указать, *что* следует сделать, а не *как* это сделать. Это упрощает концептуализацию, построение и последующую модификацию решений.

Ценность итераторов

Итак, мы рассмотрели механизм работы итераторов. Но чтобы понять, почему они так важны, потребуется более сложный пример.

В настоящих объектно-ориентированных программах полиморфизм, динамическое создание объектов и контейнеры часто объединяются. Контейнеры и механизм динамического создания объектов решают проблему неизвестности типов создаваемых объектов. А если контейнер рассчитан на хранение указателей на объекты базового класса, при каждом сохранении в контейнере указателя на производный класс выполняется повышающее приведение типа (с соответствующими выгодами в плане организации кода и расширяемости). В завершение первого тома книги будет представлен пример, в котором собраны воедино различные аспекты всего, о чем рассказывалось ранее. Если вы сможете разобраться в работе этого примера, то вы готовы к чтению второго тома.

Допустим, наша программа предназначена для редактирования и создания графических изображений. Каждое изображение представляет собой объект, содержащий коллекцию объектов Shape:

```
//: C16:Shape.h
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>
#include <string>

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
```

```
public:
  Circle() {}
  ~Circle() { std::cout << "Circle::~Circle\n"; }
  void draw() { std::cout << "Circle::draw\n";}
  void erase() { std::cout << "Circle::erase\n";}
};

class Square : public Shape {
public:
  Square() {}
  ~Square() { std::cout << "Square::~Square\n"; }
  void draw() { std::cout << "Square::draw\n";}
  void erase() { std::cout << "Square::erase\n";}
};

class Line : public Shape {
public:
  Line() {}
  ~Line() { std::cout << "Line::~Line\n"; }
  void draw() { std::cout << "Line::draw\n";}
  void erase() { std::cout << "Line::erase\n";}
};
#endif // SHAPE_H ///:~
```

В этом примере используется классическая архитектура с виртуальными функциями в базовом классе и их переопределением в производном классе. Обратите внимание: класс Shape содержит виртуальный деструктор, который следует автоматически включать в любой класс с виртуальными функциями. Если контейнер содержит указатели или ссылки на объекты Shape, то при вызове виртуальных деструкторов для этих объектов уничтожение пройдет правильно.

Для разных типов изображений в следующем примере используются разные параметризованные контейнерные классы: PStash и Stack, определения которых приводились в этой главе, а также класс vector из стандартной библиотеки C++. «Использование» контейнеров предельно упрощено; в общем случае наследование может оказаться не лучшим решением (возможно, правильнее воспользоваться композицией), но в нашем случае наследование работает достаточно просто и не отвлекает от основной темы примера.

```
//: C16:Drawing.cpp
#include <vector> // Используется стандартный контейнер vector!
#include "TPStash2.h"
#include "TStack2.h"
#include "Shape.h"
using namespace std;

// Drawing --  контейнер для хранения объектов Shape:
class Drawing : public PStash<Shape> {
public:
  ~Drawing() { cout << "~Drawing" << endl; }
};

// Plan -- другой контейнер для хранения объектов Shape:
class Plan : public Stack<Shape> {
public:
  ~Plan() { cout << "~Plan" << endl; }
};

// Schematic -- еще один контейнер с объектами Shape:
```

```
class Schematic : public vector<Shape*> {
public:
  ~Schematic() { cout << "~Schematic" << endl; }
};

// Шаблон функции:
template<class Iter>
void drawAll(Iter start, Iter end) {
  while(start != end) {
    (*start)->draw();
    start++;
  }
}

int main() {
  // Каждый тип контейнера обладает своим интерфейсом:
  Drawing d;
  d.add(new Circle);
  d.add(new Square);
  d.add(new Line);
  Plan p;
  p.push(new Line);
  p.push(new Square);
  p.push(new Circle);
  Schematic s;
  s.push_back(new Square);
  s.push_back(new Circle);
  s.push_back(new Line);
  Shape* sarray[] = {
    new Circle, new Square, new Line
  };
  // Итераторы и шаблонные функции позволяют выполнять с ними
  // обобщенные операции:
  cout << "Drawing d:" << endl;
  drawAll(d.begin(), d.end());
  cout << "Plan p:" << endl;
  drawAll(p.begin(), p.end());
  cout << "Schematic s:" << endl;
  drawAll(s.begin(), s.end());
  cout << "Array sarray:" << endl;
  // Even works with array pointers:
  drawAll(sarray,
    sarray + sizeof(sarray)/sizeof(*sarray));
  cout << "End of main" << endl;
} ///:~
```

Во всех типах контейнеров хранятся указатели на Shape, а также указатели на объекты классов, производных от Shape, прошедшие повышающее приведение типа. Но благодаря полиморфизму вызовы виртуальных функций все равно работают так, как нужно.

Учтите, что sarray (массив Shape*) тоже может рассматриваться как своего рода контейнер.

Шаблоны функций

В функции drawAll() встречается нечто новое. До настоящего момента мы использовали только шаблоны классов, которые создают экземпляры новых классов, па-

раметризованные по одному или нескольким типам. Однако с таким же успехом можно использовать *шаблоны функций* для создания новых функций, параметризованных по типам. Шаблоны функций создаются по тем же причинам, что и шаблоны классов: вы пишете обобщенный код и хотите отложить задание одного или нескольких типов. Требуется указать лишь то, что эти параметры-типы поддерживают некоторые операции (а не то, к какому конкретному типу они относятся).

Шаблон функции drawAll() можно рассматривать как *алгоритм* (так называется большинство шаблонных функций в стандартной библиотеке C++). Шаблонная функция всего лишь объясняет, как некоторая операция выполняется с итераторами, описывающими интервал элементов (при условии, что для итераторов поддерживается разыменование, инкремент и сравнение). Именно такие итераторы разрабатывались нами в этой главе; они же используются контейнерами стандартной библиотеки C++, что доказывает класс vector в нашем примере (причем это совпадение далеко не случайно).

Шаблонная функция drawAll() должна представлять собой *универсальный алгоритм*, чтобы можно было работать с контейнерами абсолютно любого типа и не приходилось писать новую версию алгоритма для каждого типа контейнера. Для достижения этой цели очень важны шаблоны функций, автоматически генерирующие специализированный код для каждого типа контейнера. Но без дополнительного уровня логической абстракции, обеспечиваемого применением итераторов, такая универсальность была бы невозможна. Вот почему так важны итераторы; они позволяют писать обобщенный код, который использует контейнеры, не обладая информацией о фактической структуре контейнера. (Учтите, в C++ для работы итераторов и универсальных алгоритмов необходима поддержка шаблонов функций).

Функция main() демонстрирует универсальность полученных шаблонов: drawAll() без изменений работает с разными типами контейнеров. И что еще интереснее, drawAll() также работает с указателями на начало и конец массива sarray. Возможность интерпретации массивов как контейнеров является неотъемлемой частью стандартной библиотеки C++, алгоритмы которой во многом похожи на функцию drawAll().

Поскольку шаблоны контейнерных классов участвуют в наследовании и повышающих преобразованиях реже, чем «обычные» классы, классы контейнеров почти никогда не содержат виртуальных функций. Многократное использование контейнерного класса реализуется посредством шаблонов, а не наследования.

Итоги

Контейнерные классы относятся к числу важнейших элементов объектно-ориентированного программирования. Они предоставляют в распоряжение программиста еще один механизм для упрощения программ и скрытия подробностей, реализации, а также ускорения разработки. Кроме того, контейнеры заменяют примитивные массивы и относительно тривиальные средства работы со структурами данных языка C, обладая при этом прекрасными показателями в плане безопасности и гибкости.

С контейнерами часто работают прикладные программисты, поэтому очень важно, чтобы контейнеры были просты в использовании. Здесь на помощь програм-

мисту приходят шаблоны. При наличии шаблонов синтаксис многократного использования исходных текстов (в отличие от многократного использования объектного кода, предоставляемого наследованием и композицией) тривиален даже для неопытных пользователей. На самом деле многократно использовать код с помощью шаблонов гораздо проще, чем с применением наследования и композиции.

В этой главе рассматривалась методика создания классов контейнеров и итераторов, но на практике намного важнее изучить контейнеры и итераторы стандартной библиотеки C++, поскольку они поддерживаются всеми компиляторами. Как будет показано во втором томе книги, контейнеры и алгоритмы стандартной библиотеки C++ удовлетворяют практически любые потребности, поэтому создавать новые контейнеры вам, скорее всего, не придется.

Мы кратко рассмотрели некоторые вопросы проектирования контейнерных классов. Возможно, вы поняли, что эта тема гораздо обширнее и сложнее. Качественная библиотека контейнерных классов может поддерживать массу дополнительных возможностей, в том числе многопоточность, сохранение и загрузку данных, уборку мусора.

Упражнения

1. Реализуйте иерархию наследования класса OShape, изображенную на диаграмме в начале главы.
2. Измените результат упражнения 1 из главы 15 так, чтобы вместо массива указателей на Shape в нем использовались классы Stack и iterator из файла TStack2.h. Включите в иерархию классов деструкторы и убедитесь в том, что объекты Shape уничтожаются при выходе Stack из области видимости.
3. Измените пример TPStash.h так, чтобы приращение, используемое функцией inflate(), могло изменяться на протяжении жизненного цикла отдельного объекта контейнера.
4. Измените пример TPStash.h так, чтобы приращение, используемое функцией inflate(), автоматически изменялось для уменьшения количества вызовов. Например, при каждом вызове приращение для следующего вызова может увеличиваться вдвое. Чтобы продемонстрировать новые возможности программы, выводите сообщение при каждом вызове inflate() и включите тестовый код в функцию main().
5. Создайте шаблон функции fibonacci() с параметризацией по типу создаваемых значений (чтобы функция могла генерировать не только int, но и long, float и т. д.).
6. На базе контейнера vector из стандартной библиотеки C++ создайте шаблон класса Set, в котором каждый объект может храниться только в одной копии. Создайте вложенный класс iterator с поддержкой «конечных итераторов», описанных в этой главе. Включите в main() код для тестирования шаблона Set, затем замените его шаблоном set из стандартной библиотеки C++ и убедитесь в том, что он работает правильно.
7. Измените файл AutoCounter.h так, чтобы класс AutoCounter мог использоваться как внутренний объект внутри любого класса, создание и уничтожение экзем-

пляров которого требуется отслеживать. Добавьте переменную string для хранения имени класса. Протестируйте AutoCounter для написанного вами класса.

8. Создайте версию OwnerStack.h, в базовой реализации которой используется контейнер vector стандартной библиотеки C++. Вероятно, для этого придется изучить описания некоторых функций vector (или просто просмотреть заголовочный файл <vector>).
9. Измените пример ValueStack.h так, чтобы исчерпание свободного места при занесении объектов функцией push() приводило к динамическому расширению контейнера. Измените тестовую программу ValueStackTest.cpp и протестируйте новые возможности.
10. Повторите упражнение 9, используя во внутренней реализации ValueStack контейнер vector стандартной библиотеки C++. Обратите внимание, насколько это упрощает программу.
11. Измените тестовую программу ValueStackTest.cpp так, чтобы вместо Stack в функции main() использовался класс стандартной библиотеки C++. Проследите за поведением программы во время выполнения: создаются ли объекты по умолчанию при создании контейнера vector?
12. Измените пример TStack2.h так, чтобы в его внутренней реализации использовался контейнер vector стандартной библиотеки C++. Проследите за тем, чтобы интерфейс остался неизменным, а программа TStack2Test.cpp работала без изменений.
13. Повторите упражнение 12, используя вместо vector контейнер stack из стандартной библиотеки C++. Обратите внимание, насколько это упрощает программу. Возможно, вам придется поискать дополнительную информацию о классе stack или просмотреть заголовочный файл <stack>.
14. Измените пример TPStash2.h так, чтобы во внутренней реализации использовался контейнер vector стандартной библиотеки C++. Проследите за тем, чтобы интерфейс остался неизменным, а программа TStash2Test.cpp работала без изменений.
15. В примере IterIntStack.cpp измените итератор IntStackIter; включите в него конструктор «конечного итератора» и добавьте операторы == и !=. В функции main() перебирайте элементы контейнера при помощи итератора до тех пор, пока не будет достигнут конечный итератор.
16. Используя файлы TStack2.h, TPStash2.h и Shape.h, создайте специализации контейнеров Stack и PStash для Shape*, заполните их указателями на Shape, полученными в результате различных повышающих преобразований, переберите элементы каждого контейнера при помощи итератора и вызовите для каждого объекта функцию draw().
17. Преобразуйте класс Int из программы TPStash2Test.cpp в шаблон, чтобы в нем можно было хранить объекты произвольного типа (придумайте ему более подходящее имя).
18. Преобразуйте класс IntArray из программы IostreamOperatorOverloading.cpp главы 12 в шаблон, параметризованный как по типу хранящихся в нем объектов, так и по размеру внутреннего массива.

19. Преобразуйте класс ObjContainer из программы NestedSmartPointer.cpp главы 12 в шаблон. Протестируйте его с двумя разными классами.
20. Измените примеры C15:Ostack.h и C15:OStackTest.cpp и преобразуйте класс Stack в шаблон, который бы автоматически создавался путем множественного наследования от класса элементов и от Object. Сгенерированный класс Stack должен принимать и возвращать только указатели на тип элементов.
21. Повторите упражнение 20, используя контейнер vector вместо Stack.
22. Создайте класс StringVector, производный от vector<void*>; переопределите функции push_back() и operator[ ] так, чтобы они принимали и возвращали только string* (организуйте соответствующее приведение типов). Затем создайте шаблон, который бы заставлял контейнерный класс делать то же самое с указателями на произвольный тип. Такая методика часто используется для уменьшения объема программного кода при слишком большом количестве экземпляров шаблонов.
23. В примере TPStash2.h создайте и протестируйте оператор - в классе PStash::iterator. Новый оператор должен работать по тем же принципам, что и оператор +.
24. В примере Drawing.cpp создайте и протестируйте шаблон функции для вызова erase().
25. (Задача повышенной трудности) Измените класс Stack в примере TStack2.h и реализуйте управление принадлежностью на уровне отдельных объектов. Для каждого указателя должен храниться флаг, указывающий, принадлежит ли объект контейнеру. Обеспечьте обработку новой информации в функции push() и деструкторе. Добавьте функции для чтения и модификации флага принадлежности на уровне отдельных элементов.
26. (Задача повышенной трудности) Измените программу PointerToMemberOperator.cpp из главы 12 так, чтобы класс FunctionObject и функция operator->* были параметризованы для работы с произвольным возвращаемым типом (для оператора ->* придется использовать шаблонные функции классов, описанные во втором томе книги). Добавьте поддержку функций класса Dog без аргументов, а также с одним и двумя аргументами. Протестируйте полученные функции.

Приложение А
Стиль программирования

Данное приложение не о том, как расставлять отступы, круглые и фигурные скобки (хотя речь пойдет и о них). В нем описываются общие правила организации программного кода, приведенного в книге.

Все решения относительно оформления программ в книге принимались абсолютно сознательно и были тщательно продуманы (иногда в течение нескольких лет). Конечно, у каждого программиста могут найтись свои причины, по которым он оформляет свои программы именно так, а не иначе. Здесь лишь делается попытка объяснить, как сформировался личный стиль программирования автора и какие ограничения и внешние факторы способствовали принятию решений.

Общие правила

В примерах книги используется особый стиль оформления программ, на который отчасти повлиял стиль, использованный Бьярном Страуструпом в первом издании книги «The C++ Programming Language». Обсуждение стиля форматирования часто переходит в многочасовые жаркие дебаты, поэтому хочется особо подчеркнуть, что автор не пытается навязать свои понятия о стиле. Поскольку C++ относится к числу языков со свободным форматированием, вы можете продолжать использовать тот стиль, который покажется вам наиболее привычным.

Однако при этом очень важно, чтобы в рамках проекта стиль оформления кода оставался единым. В Интернете можно найти немало утилит, позволяющих переформатировать весь код проекта для соблюдения заветного единства стиля.

Файлы в архиве примеров были автоматически извлечены из текста книги, что позволило протестировать примеры и убедиться в их работоспособности. Следовательно, примеры программ должны работать без ошибок компиляции в любой среде, соответствующей требованиям стандарта C++ (учтите, что некоторые компиляторы могут не поддерживать те или иные возможности языка). Строки, которые *должны* вызывать ошибки компиляции, закомментированы в формате //!, что упрощает автоматизацию поиска и тестирования.

При подготовке книги было решено, что все приводимые программы должны компилироваться и компоноваться без ошибок (хотя иногда возможны предупреждения). Для этого некоторые примеры, лишь демонстрирующие те или иные концепцию и не являющиеся самостоятельными программами, имеют пустую функции main():

```
int main() {}
```

Это позволяет компоновщику завершить свою работу без ошибок.

По стандарту функция main() возвращает int, однако в стандарте C++ сказано, что при отсутствии команды return в функции main() компилятор автоматически генерирует команду return 0. Эта возможность — отсутствие явной команды return в функции main() — часто используется в книге (некоторые компиляторы выдают предупреждения, но такое поведение не соответствует стандарту C++).

Имена файлов

В C заголовочные файлы (содержащие объявления) обычно имеют расширение .h, а для файлов реализации (в которых выделялась память и генерировался код) — расширение .c. В языке C++ система расширений прошла определенную эволюцию. На первом этапе C++ разрабатывался под управлением операционной системы Unix, различавшей символы верхнего и нижнего регистров в именах файлов. Поэтому первоначально в именах файлов C++ использовались те же расширения, что и в C, но записанные в верхнем регистре: .H и .C. Конечно, этот вариант не подходил для операционных систем, не различавших регистр символов (например, DOS). Разработчики компиляторов под DOS использовали для заголовочных файлов расширения .hxx или .cxx, а для файлов реализации — расширения .hpp или .cpp. Позднее кто-то сообразил, что разные расширения нужны только для того, чтобы компилятор мог выбрать режим компиляции файла — C или C++. Поскольку компилятор никогда не компилирует заголовочные файлы напрямую, достаточно изменить расширение только для заголовочных файлов. Теперь практически во всех системах принято расширение .cpp для файлов реализации и расширение .h для заголовочных файлов. Обратите внимание: при включении заголовочных файлов в стандарте C++ используется синтаксис без расширения:

```
#include <iostream>
```

Начальные и конечные комментарии

При написании книги приходилось учитывать еще один очень важный аспект: весь приводимый код обязательно проверялся хотя бы на одном компиляторе, а это требовало автоматического извлечения всех файлов с программным кодом из текста книги. Для упрощения этой задачи все листинги, предназначенные для компиляции (в отличие от немногочисленных фрагментов), помечались специальными начальными и конечными комментариями. Эти комментарии использовались утилитой ExtractCode.cpp, описанной во втором томе, для извлечения листингов из текстовой версии книги.

Конечный комментарий просто сообщает утилите ExtractCode.cpp о завершении комментария, но за начальным комментарием указывается подкаталог, в котором

находится файл (каждый подкаталог соответствует отдельной главе, поэтому файл, например, из главы 8 снабжается пометкой C08), за ним следуют двоеточие и имя файла.

Поскольку утилита ExtractCode.cpp также генерирует make-файл для каждого подкаталога, в листинги включена информация о построении программы и о командной строке, используемой для ее тестирования. Полностью автономные программы (которые не требуют компоновки с другими программами) не содержат дополнительной информации. То же самое относится к заголовочным файлам. Но если программа не содержит функции main() и предназначена для компоновки с другими модулями, после имени файла следует маркер {O}. Если листинг содержит основную программу, компонующуюся с другими модулями, комментарий содержит дополнительную строку, которая начинается с символов //{L} и содержит имена всех компонуемых файлов (без расширений, которые могут зависеть от платформы).

Примеры часто встречаются в книге.

Если файл должен извлекаться из текста книги без начальных и конечных комментариев (например, если он содержит тестовые данные), то за начальным тегом сразу же следует символ восклицательного знака (!).

Отступы, круглые и фигурные скобки

Возможно, вы заметили, что стиль форматирования программ в книге отличается от многих традиционных стилей языка C. Разумеется, любой программист убежден в том, что его стиль является самым рациональным. Тем не менее использованный в книге стиль основан на простой логике, о которой мы поговорим чуть позже. Заодно мы попытаемся разобраться, почему появляются другие стили.

Стиль форматирования выбирается только для одной цели: для внешнего представления программы (в печатном виде, в презентации на семинаре и т. д.). Возможно, ваш код не приходится часто представлять другим, и вы решите, что к вам это не относится. Однако рабочие программы читаются гораздо чаще, чем пишутся, поэтому они должны хорошо восприниматься читателем. Двумя основными критериями автор считает «сканируемость» (то есть простоту восприятия одной строки читателем) и количество строк на странице. Второй критерий может показаться странным, но при интерактивной подаче материала частые переходы между слайдами отвлекают аудиторию, а их причиной как раз могут стать несколько лишних строк.

Все согласны с тем, что код в фигурных скобках должен снабжаться отступами. Разногласия начинаются по другому вопросу: где должна находиться открывающая фигурная скобка? Этот вопрос породил множество разнообразных стилей форматирования. Здесь нужно учитывать, что многие применяемые стили обусловлены ограничениями, действовавшими до появления стандарта C (а вернее, до появления прототипов функций), и поэтому в наши дни они неактуальны.

Прежде всего, автор считает, что открывающая фигурная скобка всегда должна находиться в одной строке с ее «предшественником» (будь то класс, функция, определение объекта, команда if и т. д.). Это универсальное единое правило применя-

ется во всех программах книги, и оно заметно упрощает форматирование. Программа проще воспринимается. Например, при взгляде на следующую строку по символу точки с запятой (;) в конце строки сразу понятно, что это объявление, и не более того:

```
int func(int a);
```

Теперь представьте, что вы встречаете такую строку:

```
int func(int a) {
```

В этом случае вы немедленно понимаете, что это определение, поскольку строка завершается открывающей фигурной скобкой, а не точной с запятой. При соблюдении этого правила открывающая фигурная скобка всегда находится в одной позиции как в многострочных определениях, так и в однострочных определениях. Пример многострочного определения:

```
int func(int a) {
  int b = a + 1;
  return b * 2;
}
```

Пример однострочного определения:

```
int func(int a) { return (a + 1) * 2; }
```

Такого рода определения часто используются в качестве определений для подставляемых функций Аналогично, следующая строка объявляет имя класса:

```
class Thing;
```

В то же время показанная ниже строка начинает определение класса:

```
class Thing {
```

Взглянув всего на одну строку, вы всегда можете определить, что перед вами — определение или объявление. И конечно, написание открывающей фигурной скобки в той же, а не в отдельной строке позволяет разместить на странице больше информации.

Так почему же существует так много других стилей? В частности, многие программисты при созданий классов применяют этот стиль (который использовался Страуструпом во всех изданиях книги «The C++ Programming Language»), но при создании функций открывающая фигурная скобка оказывается в отдельной строке (что также порождает множество разных стилей расстановки отступов). Кстати, Страуструп тоже поступает так во всех случаях, кроме коротких подставляемых функций. В том варианте, который применяется в этой книге, все абсолютно последовательно: вы указываете нечто (класс, функцию, перечисляемый тип и т. д.) и в этой же строке ставите открывающую фигурную скобку, сообщая, что дальше следует тело этого «нечто». Наконец, в определениях коротких подставляемых и обычных функций открывающая фигурная скобка располагается в одном и том же месте.

По мнению автора, стиль определения функций, применяемый многими программистами, берет свое начало в первых версиях С (до появления прототипов), где аргументы объявлялись не в круглых скобках, а между закрывающей круглой и открывающей фигурной скобками (в этом проявляются ассемблерные корни С):

```
void bar()
  int x;
```

```
  float y;
{
  /* Тело */
}
```

В таком варианте размещать открывающую фигурную скобку за последним аргументом неудобно, поэтому так никто не поступал. С другой стороны, программисты принимали разные решения относительно того, должно ли тело функции снабжаться отступами по отношению к фигурным скобкам или же находиться на уровне «предшественника». Так появились разные стили форматирования.

Существуют и другие доводы в пользу размещения фигурной скобки в строке, следующей за объявлением (класса, структуры, функции и т. д.). Вот мнение одного читателя, чтобы вы знали и об этих доводах.

Опытные пользователи редактора vi (vim) знают, что двукратное нажатие клавиши] осуществляет переход к следующему вхождению символа { в столбце 0. Эта возможность чрезвычайно удобна для перемещения в программном коде (переход к следующему определению функции или класса)[1].

Размещение символа { в следующей строке упрощает структуру кода в некоторых сложных условных конструкциях и упрощает чтение программы. Пример:

```
if(cond1
   && cond2
   && cond3) {
   команда;
}
```

Предыдущий фрагмент, по мнению читателя, плохо воспринимается. С другой стороны, следующая запись отделяет if от исполняемой части и упрощает чтение (вероятно, ваше мнение по этому вопросу будет зависеть от того, к чему вы привыкли):

```
if(cond1
   && cond2
   && cond3)
{
команда;
}
```

Наконец, парные скобки гораздо проще искать, когда они находятся в одном столбце, поскольку они визуально выделяются в программе.

Вероятно, вопрос о положении открывающей фигурной скобки вызывает больше всего споров. Автор научился спокойно воспринимать программы в обоих форматах, так что в конечном счете все сводится к выбору более привычного способа. Однако здесь надо учитывать, что официальный стандарт программирования на Java (познакомиться с ним можно на веб-сайте Sun) фактически не отличается от представленного автором. Так как многие программисты одновременно начинают работать на обоих языках, единство стиля программирования может пригодиться.

---

[1] Начиная работать в Unix, автор увлекся только что появившимся редактором GNU Emacs и не ощутил «прелести» vi. В результате автор так и не привык думать в контексте «позиции 0 столбца». И все же количество пользователей vi достаточно велико, и для них этот фактор имеет значение.

Выбранный автором стиль не имеет исключений или особых случаев и логически приводит к единому стилю расстановки отступов. Это единство сохраняется даже внутри тела функции:

```
for(int i = 0; i < 100; i++) {
  cout << i << endl;
  cout << x * i << endl;
}
```

Этот стиль легко изучается и запоминается — все форматирование подчиняется единым логически последовательным правилам без деления на классы и функции (или многострочные и однострочные подставляемые функции) и без выделения особых случаев для циклов for, команд if и т. д. Одно это единообразие уже заслуживает внимания. Не забывайте, что язык C++ появился позднее C; хотя он во многом связан с C, не стоит искусственно переносить в него старые привычки, которые могут обернуться проблемами в будущем. Мелкие проблемы, повторяясь во множестве строк программного кода, становятся крупными.

Другое ограничение, которое автору также приходилось учитывать, — максимальная длина строки. В книге максимальная длина строка листинга ограничивалась пятьюдесятью символами. Что делать, если какая-нибудь конструкция не помещается в одной строке? Автор снова постарался выработать единый стиль разбиения строк, чтобы упростить их чтение. В логических частях одного определения, списка аргументов и т. д. в строках продолжения отступ увеличивался на один уровень по сравнению с началом определения, списка аргументов и т. д.

Имена идентификаторов

Читатели, знакомые с языком Java, могли заметить, что автор перешел на схему выбора имен, которая считается стандартной для языка Java. Впрочем, добиться полного единства не удалось, потому что идентификаторы стандартных библиотек C и C++ не соответствуют этой схеме.

Схема весьма прямолинейна. Идентификатор начинается с прописной буквы только в том случае, если определяет имя класса. Идентификаторы функций и переменных записываются со строчной буквы. Оставшаяся часть идентификатора состоит из одного или нескольких объединенных слов, каждое из которых начинается с прописной буквы. Например, идентификаторы классов выглядят так:

```
class FrenchVanilla : public IceCream {
```

Пример идентификатора объекта:

```
FrenchVanilla myIceCreamCone(3);
```

Пример идентификатора функции (причем это может быть как обычная функция, так и функция класса):

```
void eatIceCreamCone();
```

Единственное исключение из этой схемы составляют константы времени компиляции (определяемые ключевым словом const или директивой #define), полностью записываемые в верхнем регистре.

В этой схеме регистр первого символа несет полезную информацию — по первой букве идентификатора сразу видно, идет речь о классе или об объекте/методе. Это особенно полезно при обращении к статическим членам классов.

Порядок включения заголовочных файлов

Заголовки включаются в порядке «от специализированных к общим». Иначе говоря, сначала включаются все заголовочные файлы в текущем каталоге, затем заголовочные файлы личного инструментария автора (например, require.h), затем заголовки стандартной библиотеки C++ и, наконец, — заголовочные файлы библиотеки C.

Джон Лакос (John Lacos) в своей книге «Large-Scale C++ Software Design» (Addison-Wesley, 1996) так обосновывает эту последовательность.

Автономная обработка h-файла компонента (без внешних объявлений и определений) помогает предотвратить скрытые ошибки. Включение h-файла в самой первой строке c-файла гарантирует присутствие всей критически важной информации, относящейся к физическому интерфейсу компонента (или ее отсутствие обнаружится немедленно при попытке откомпилировать c-файл).

Если заголовочные файлы включаются в порядке «от специализированных к общим», вы быстрее обнаружите те из них, которые не обрабатываются автономно, и избавите себя от дальнейших огорчений.

Защита заголовков

Механизм *защиты заголовков* всегда используется в заголовочных файлах для предотвращения их повторного включения во время компиляции cpp-файла. Для этого при первом включении выполняется препроцессорная директива #define, а далее препроцессор проверяет, определено ли соответствующее символическое имя. Это имя создается из имени заголовочного файла: все символы преобразуются к верхнему регистру, а точка заменяется символом подчеркивания. Пример:

```
// IncludeGuard.h
#ifndef INCLUDEGUARD_H
#define INCLUDEGUARD_H
// Тело заголовочного файла...
# endif // INCLUDEGUARD_H
```

Идентификатор в последней строке добавлен для ясности. Хотя некоторые препроцессоры игнорируют любые символы после #endif, это поведение не оговорено в стандарте, поэтому идентификатор закомментирован.

Использование пространств имен

В заголовочных файлах следует тщательно избегать «загрязнения» пространства имен, в которое включается заголовочный файл. Иначе говоря, модификация пространства имен вне функции или класса распространяется на все файлы, включающие этот заголовочный файл, что приводит к всевозможным проблемам. Использо-

вание любых объявлений using вне определений функций запрещено, и заголовочные файлы не могут содержать глобальные директивы using.

В cpp-файлах глобальные директивы using действуют только в границах файла, поэтому они часто используются в книге для получения более компактного кода (особенно в небольших программах).

Функции require() и assure()

Функции require() и assure(), определенные в файле require.h, часто используются в книге для получения диагностической информации. Если вы знакомы с концепциями *предусловий* и *постусловий*, предложенными Бертраном Мейером (Bertrand Meyer), то вы поймете, что require() и assure() более или менее соответствуют предусловиям (обычно) и постусловиям (в отдельных случаях). В начале функции (перед выполнением ее «основного» кода) мы проверяем предусловия и убеждаемся в том, что все необходимые требования для вызова соблюдены. После выполнения «основного» кода функции иногда проверяются постусловия; с их помощью можно убедиться в том, что новое состояние данных лежит в допустимых пределах. Постусловия в книге проверяются редко, а функция assure() чаще всего применяется для проверки успешного открытия файлов.

Приложение Б
Рекомендации по программированию

В этом приложении содержится подборка рекомендаций по программированию на C++. Эти рекомендации были составлены на основании личного авторского опыта программирования и обучения, а также советов некоторых его друзей, в числе которых Том Плам (Tom Plum), Скотт Мейерс (Scott Meyers) и Роб Мюррей (Rob Murray). Во многих полезных советах просто обобщается то, о чем говорится на страницах книги.

1. Сначала заставьте программу работать, а потом занимайтесь усовершенствованиями. Это правило справедливо даже в том случае, если вы работаете над очень важным фрагментом, эффективность которого заведомо отразится на эффективности системы в целом. Не увлекайтесь ранней оптимизацией. Сначала добейтесь, чтобы программа заработала в самой простой архитектуре. Если она окажется недостаточно быстрой, беритесь за профайлер. Почти всегда выясняется, что проблемы кроются совсем не там, где вы ожидаете. Поберегите время для решения действительно важных задач.
2. Элегантность всегда окупается. Не стоит считать ее бесполезным эстетствованием. Элегантное программирование упрощает не только разработку и отладку, но также чтение и сопровождение программы, а с этим связана прямая финансовая выгода. Возможно, вы не сразу поверите этой рекомендации. На первый взгляд кажется, что время, потраченное на доработку программы, тратится неэффективно. Однако эффективность проявляется тогда, когда ваш код гладко интегрируется в работу системы, и еще более наглядно — при модификации этого кода или всей системы.
3. Помните принцип «разделяй и властвуй». Если вам попалась слишком сложная задача, попробуйте представить, как должен выглядеть базовый принцип действия программы, если все сложные операции будут выполняться неким «волшебным компонентом». Этим волшебным компонентом должен быть объект — напишите код, работающий с объектом, затем проанализируйте объект и инкапсулируйте *его* сложные части в других объектах и т. д.

4. Не переписывайте весь существующий код C на C++, если только не нужно существенно менять его функциональность (а проще говоря, не чините то, что не ломалось). *Перекомпиляция* кода C на C++ приносит пользу, потому что она помогает обнаружить скрытые ошибки. Но переписывание на C++ нормально работающих программ C обычно является малопродуктивным занятием — разве что версия C++ обеспечит возможность многократного использования кода в виде класса.

5. Если у вас имеется большой объем кода C, требующий внесения изменений, для начала выделите те части кода, которые должны остаться неизменными. Возможно, их стоит выделить в «класс API» в виде статических функций. Затем сосредоточьте внимание на изменяемом коде и переработайте его в систему классов, чтобы упростить внесение изменений в процессе дальнейшего сопровождения программы.

6. Отделите создателя класса от пользователя класса (*прикладного программиста*). Пользователю класса не нужно знать, как устроен класс и как он работает. Создатель класса должен быть специалистом в области проектирования классов, чтобы его классы могли использоваться даже неопытными программистами, но при этом надежно работали в приложениях. С библиотеками удобно работать только в том случае, если все операции абсолютно прозрачны.

7. При создании класса постарайтесь сделать имена как можно более информативными. Ваша цель — по возможности упростить интерфейс, с которым работает прикладной программист, на концептуальном уровне. Постарайтесь сделать имена предельно понятными, чтобы любые комментарии были излишними. С этой же целью используйте перегрузку функций и аргументы по умолчанию для создания интуитивно понятного интерфейса.

8. Ограничение доступа позволит вам (создателю класса) в будущем изменять внутреннюю реализацию класса, не нарушая работы клиентского кода, в котором этот класс используется. В свете сказанного как можно большая часть класса должна быть закрытой, а открытым должен быть только интерфейс класса, причем в нем всегда должны использоваться функции вместо данных. Открытый доступ к данным предоставляется только при крайней необходимости. Если какая-либо функция не нужна пользователям класса, объявите ее закрытой. Если некоторая часть класса должна предоставляться производным классам как защищенная, реализуйте интерфейс на базе функций, не предоставляя доступа к данным. При таком подходе изменения в реализации окажут минимальное воздействие на работу производных классов.

9. Не допускайте «аналитического паралича». Некоторые обстоятельства просто невозможно узнать до того, как вы приступите к программированию и создадите некий прототип рабочей системы. В C++ имеются встроенные средства защиты; воспользуйтесь ими. Ваши ошибки в классах или группах классов не должны нарушать целостность всей системы.

10. В результате анализа и проектирования как минимум должна быть сформирована структура классов системы, их открытых интерфейсов и связей с дру-

гими классами, особенно базовыми. Если ваша методология проектирования дает нечто большее, спросите себя, все ли компоненты, полученные при помощи этой методологии, будут иметь ценность на протяжении жизненного цикла программы. Если нет, то их сопровождение повлечет за собой лишние затраты. Разработчики обычно не склонны поддерживать то, что не влияет на продуктивность их работы; это житейское обстоятельство не учитывается во многих методах проектирования.

11. Напишите тестовый код заранее (еще до написания класса) и храните его вместе с классом. Автоматизируйте запуск теста при помощи make-файла или другой аналогичной утилиты. В этом случае любые изменения будут автоматически проверяться запуском тестовой программы, и вы немедленно обнаружите допущенные ошибки. Если вы знаете, что система тестов обеспечит дополнительную страховку, вам будет проще внести серьезные изменения, если это потребуется. Помните, что важнейшие усовершенствования в языках программирования обусловлены встроенными средствами тестирования: проверкой типов, обработкой исключений и т. д., однако их возможности небезграничны. Остаток пути к построению надежной системы придется пройти самостоятельно, для чего необходимо спроектировать систему тестов, проверяющих специфические возможности вашего класса или программы.
12. Напишите тестовый код заранее (еще до написания класса), чтобы убедиться в полноте архитектуры класса. Если написать тестовый код не удается, значит, вы плохо представляете, каким должен быть ваш класс. Кроме того, в процессе написания тестового кода часто выявляются дополнительные возможности или ограничения, необходимые для данного класса, которые не всегда удается выявить во время анализа и проектирования.
13. Помните основное правило разработки программного обеспечения[1]: *любые проблемы проектирования упрощаются введением нового уровня концептуальной косвенности.* Этот принцип заложен в основу абстракции — одной из основных концепций объектно-ориентированного программирования.
14. Старайтесь делать классы атомарными; иначе говоря, каждый класс создается для единственной четко выраженной цели. Если классы в вашей системной архитектуре становятся слишком сложными, разбейте сложные классы на несколько простых. Наиболее очевидным признаком можно считать размер; если класс слишком велик, скорее всего, на него возложено слишком много функций, и его следует разделить.
15. Избегайте длинных определений функций классов. Сопровождение длинных, запутанных функций требует больших усилий и дорого обходится; вероятно, на такую функцию взвалено слишком много работы. Если вы обнаружите такую функцию, по меньшей мере, ее стоит разбить на несколько меньших функций. Еще имеет смысл подумать о создании нового класса.
16. Избегайте длинных списков аргументов, потому что они усложняют написание программы, ее чтение и последующее сопровождение. Вместо этого

[1] Это правило разъяснил Эндрю Кёниг (Andrew Koenig).

попробуйте переместить такую функцию в класс, где она окажется более уместной, или организуйте передачу объектов в аргументах.

17. Не повторяйтесь. Если некоторый фрагмент кода часто встречается во многих функциях производных классов, выделите его в одну функцию базового класса и вызывайте ее из функций производных классов. Тем самым вы не только уменьшите объем программы, но и упростите ее модификацию. Для повышения эффективности можно воспользоваться подставляемой функцией. Иногда выделение общего кода добавляет новые полезные возможности в интерфейс.

18. Избегайте команд switch и длинных цепочек команд if-else. Обычно они применяются при *программирования с проверкой типов*, когда выполняемый код выбирается в зависимости от некоторого признака типа (фактический тип может быть неочевиден). Как правило, подобный код удается заменить, применяя наследование и полиморфизм; полиморфный вызов функции автоматически определяет тип и обеспечивает более надежное и простое расширение программы.

19. С позиций проектирования найдите и отделите изменяемые части от тех, которые остаются постоянными. Другими словами, найдите в системе элементы, изменение которых не потребует переработки общей архитектуры, и инкапсулируйте их в классах. Данная концепция гораздо подробнее рассматривается в главе второго тома, посвященной эталонам проектирования.

20. Два семантически разных объекта могут выполнять одинаковые операции или решать одинаковые задачи; возникает естественное искушение объявить один объект производным от другого, чтобы извлечь пользу от наследования. На практике определение искусственных наследных связей там, где их быть не должно, не оправданно. Более удачное решение заключается в создании общего базового класса, который бы предоставлял интерфейс для обоих производных классов. Объем программного кода при этом увеличится, но вы по-прежнему воспользуетесь выгодами наследования... а возможно, по-новому взглянете на архитектуру своей системы.

21. Избегайте *ограничений* при наследовании. В грамотно построенной архитектуре унаследованные возможности дополняются новыми возможностями. Если при наследовании старые возможности исключаются без добавления новых, такая архитектура выглядит подозрительно. Впрочем, правила создаются для того, чтобы их нарушали. Если вы работаете со старой библиотекой классов, иногда бывает эффективнее ограничить существующий класс в производных классах, чем реструктурировать иерархию, размещая новый класс там, где он должен находиться, — над старым классом.

22. Не расширяйте базовую функциональность при наследовании. Если некоторый элемент интерфейса особенно важен для класса, он должен находиться в базовом классе, а не добавляться при наследовании.

23. При проектировании класса начинайте с минимального предельно компактного и простого интерфейса, необходимого для решения текущей задачи. Не пытайтесь заранее прогнозировать все способы, которыми *может* использоваться ваш класс. Во время использования класса может выясниться, что

интерфейс необходимо расширить. Но если класс уже функционирует, любые *сокращения* интерфейса приведут к нарушению работы клиентского кода. Добавление новых функций не отразится на клиентском коде (разве что потребует перекомпиляции). Но даже если новые функции замещают функциональность старых функций, не изменяйте существующий интерфейс (при желании объедините функциональность во внутренней реализации). Если потребуется расширить интерфейс существующей функции и включить в него новые аргументы, оставьте исходный порядок старых аргументов и назначьте всем новым аргументам значения по умолчанию. Тем самым вы защитите все существующие вызовы этой функции.

24. Прочитайте объявления своих классов вслух, чтобы убедиться в их логичности. Для обозначения связи между базовым и производным классом используйте термин «является частным случаем», а для внутренних объектов — «содержит».
25. Выбирая между наследованием и композицией, спросите себя, нужно ли выполнять повышающее приведение к базовому типу. Если нет, предпочтение следует отдать композиции (то есть включению внутренних объектов). Тем самым будет предотвращена кажущаяся необходимость в множественном наследовании. Если выбрать наследование, то пользователи будут предполагать, что оно подразумевает повышающее приведение типа.
26. Иногда приходится наследовать для обращения к защищенным членам базового класса. На первый взгляд может показаться, что ситуация требует множественного наследования. Если вы не собираетесь выполнять повышающее приведение типа, сначала создайте новый производный класс для обращения к защищенным членам. Затем включите объект этого класса в другой класс, который должен использовать его (то есть примените композицию вместо наследования).
27. Как правило, базовый класс требуется в основном для формирования интерфейса классов, производных от него. Из-за этого при создании базового класса его функции часто объявляются чисто виртуальными. Деструктор тоже может быть чисто виртуальным (чтобы заставить его переопределяться в производных классах), но не забудьте предоставить тело деструктора, потому что при уничтожении объекта всегда вызываются все деструкторы в иерархии.
28. При включении в класс виртуальной функции объявите все функции этого класса виртуальными и включите виртуальный деструктор. Такой подход предотвращает возможные сюрпризы в поведении интерфейса. Ключевые слова `virtual` убираются только в процессе оптимизации, если профайлер покажет на необходимость действовать в этом направлении.
29. Переменные класса используйте для модификации содержания, а виртуальные функции — для модификации поведения. Иначе говоря, если в некотором классе задействованы переменные состояния, на основании которых изменяется поведение класса, вероятно, лучше перепроектировать этот класс и выразить различия в поведении при помощи производных классов и переопределяемых виртуальных функций.

30. Если потребуется выполнить операцию, которая не работает на других платформах, создайте для нее абстракцию и локализуйте в классе. Тем самым будет предотвращено распространение непереносимости в программе.
31. Избегайте множественного наследования. Оно применяется в специфических ситуациях, особенно при восстановлении нарушенных интерфейсов, когда класс-нарушитель находится вне вашего контроля (см. второй том). Только опытные программисты могут использовать множественное наследование при проектировании своих систем.
32. Не используйте закрытое наследование. Хотя оно поддерживается в C++ и иногда находит практические применения, закрытое наследование создает существенные неоднозначности в сочетании с механизмом RTTI. Замените закрытое наследование созданием закрытого внутреннего объекта.
33. Если два класса связываются друг с другом на функциональном уровне (как, например, контейнеры и итераторы), попробуйте сделать один класс открытым вложенным дружественным классом внутри другого, как итераторы по отношению к контейнерам в стандартной библиотеке C++ (примеры приводятся в конце главы 16). Тем самым вы не только подчеркнете связь между классами, но и сможете многократно использовать имя вложенного класса в другом классе. В стандартной библиотеке C++ в каждом контейнерном классе определяется вложенный класс iterator, что обеспечивает общий интерфейс для всех контейнеров. Вложенные классы также создаются как часть закрытой реализации. В этом случае вложение выполняется для скрытия реализации.
34. Перегрузка операторов — всего лишь «синтаксический сахар», другой способ вызова функций. Если в результате перегрузки оператора интерфейс класса не становится проще и понятнее, откажитесь от перегрузки. Создайте только один оператор автоматического приведения типа для класса. В общем случае при перегрузке операторов следует руководствоваться советами и правилами, приведенными в главе 12.
35. Не становитесь жертвой преждевременной оптимизации. Этот путь приводит к безумию. В частности, не пытайтесь на начальном этапе разработки системы превращать обычные функции в подставляемые (или наоборот), объявлять невиртуальные функции или дорабатывать код с целью добиться максимальной эффективности. На этой стадии вашей главной целью должна быть проверка архитектуры (хотя может оказаться, что эта архитектура требует определенного уровня эффективности).
36. Обычно не стоит разрешать компилятору создавать конструкторы, деструкторы и оператор присваивания за вас. Проектировщик класса всегда должен точно указывать, какие операции выполняются классом, и держать класс под своим контролем. Если вам не нужен копирующий конструктор или оператор присваивания, объявите их закрытыми. Помните, что создание любого конструктора предотвращает автоматическое построение компилятором конструктора по умолчанию.
37. Если класс содержит указатели, то для нормальной работы класса следует определить копирующий конструктор, оператор присваивания и деструктор.

38. При написании копирующего конструктора для производного класса помните о необходимости явного вызова копирующего конструктора базового класса, а также копирующих конструкторов внутренних объектов (см. главу 14). Если этого не сделать, то для базового класса (или внутренних объектов) будет вызван конструктор по умолчанию; вероятно, это не то, чего вы добивались. При вызове копирующего конструктора базового класса передайте ему копируемый объект производного класса:

```
Derived(const Derived& d) : Base(d) { // ...
```

39. При написании оператора присваивания для производного класса помните о необходимости явного вызова версии оператора присваивания из базового класса (см. главу 14). Если вы этого не сделаете, соответствующий вызов не будет сгенерирован автоматически (это относится и к внутренним объектам класса). Чтобы вызвать оператор присваивания базового класса, воспользуйтесь именем базового класса и оператором уточнения области видимости (::):

```
Derived& operator=(const Derived& d) {
  Base::operator=(d);
```

40. Для сокращения количества перекомпиляций при разработке большого проекта воспользуйтесь классами-манипуляторами, или «чеширскими котами» (см. главу 5). Удалите их только в том случае, если у вас возникнут проблемы с эффективностью программы на стадии выполнения.
41. Постарайтесь не прибегать к помощи препроцессора. Всегда используйте ключевое слово const для подстановки значений и спецификатор inline вместо макросов.
42. Области видимости в программе должны быть как можно меньше, чтобы жизненный цикл объектов имел минимальную продолжительность. Тем самым снижается вероятность использования объекта в неверном контексте и появления трудноуловимых ошибок. Допустим, имеются контейнер и фрагмент кода, перебирающий его элементы. Если скопировать этот код для нового контейнера, размер старого контейнера может быть случайно задействован при определении верхней границы нового контейнера. Однако если старый контейнер выйдет из области видимости, ошибка будет обнаружена на стадии компиляции.
43. Избегайте глобальных переменных. Всегда старайтесь размещать свои данные внутри классов. Глобальные функции чаще находят оправданное применение, чем глобальные переменные, хотя позднее может выясниться, что и глобальную функцию лучше оформить в виде статической функции класса.
44. Если потребуется объявить класс или функцию из библиотеки, всегда делайте это с включением заголовочного файла. Например, если вы захотите создать функцию для записи в ostream, никогда не объявляйте ostream с неполными спецификациями вида:

```
class ostream;
```

При таком подходе ваша программа начинает зависеть от изменений в реализации (например, тип ostream в действительности может быть получен

с помощью оператора typedef). Вместо этого всегда используйте заголовочный файл:

```
#include <iostream>
```

Если вы занимаетесь созданием классов для очень большой библиотеки, предоставьте в распоряжение пользователей сокращенную форму заголовочного файла с неполными спецификациями типа (то есть объявлениями имен классов) для тех ситуаций, когда пользователи ограничиваются указателями; это может ускорить компиляцию.

45. Выбирая тип значения, возвращаемого перегруженным оператором, подумайте, что произойдет при объединении выражений. Возвращайте копию или ссылку на l-значение (return *this), чтобы возвращаемое значение могло использоваться в цепочках вида A=B=C. При определении оператора присваивания помните о выражении x=x.

46. При написании функции прежде всего стоит рассмотреть вариант передачи аргументов по ссылке на константу. Если вам не нужно изменять передаваемый объект, этот способ оптимален, поскольку он обладает простым синтаксисом передачи по значению, не требуя дорогостоящих операций конструирования и уничтожения локального объекта, характерных для передачи по значению. Обычно на стадии проектирования и построения системы не стоит чрезмерно беспокоиться о проблемах эффективности, но эта привычка беспроигрышная.

47. Помните о временных объектах. Во время оптимизации системы обращайте внимание на создание временных объектов, особенно при перегрузке операторов. Если конструкторы и деструкторы сложны, то затраты на создание и уничтожение временных объектов могут быть значительными. При возврате значения из функции всегда старайтесь строить объект «на месте» вызовом конструктора в команде return:

```
return MyType(i. j):
```

Эта строка смотрится лучше пары строк:

```
MyType x(i. j):
return x:
```

В первом случае пропадает необходимость в вызовах копирующего конструктора и деструктора (это называется *оптимизацией возвращаемого значения*).

48. При создании конструкторов учитывайте возможность исключений. В идеальном случае конструктор не должен выполнять никаких операций, которые могли бы породить исключения. Если это невозможно, при композиции или наследовании класса следует использовать только надежные классы, которые при возникновении исключения автоматически «убирали бы за собой». При работе непосредственно с указателями вы отвечаете за перехват исключений и освобождение всех ресурсов, на которые ссылается указатель, перед выдачей исключения в конструкторе. Если конструктору не удается завершить свою работу, он должен генерировать исключение.

49. Конструкторы должны делать только то, что действительно необходимо. Это приводит к снижению затрат на вызовы конструкторов (многие из которых находятся вне вашего контроля), и конструкторы реже являются источником проблем и генерируют исключения.

50. Деструктор отвечает за освобождение ресурсов, выделенных на протяжении всего жизненного цикла объекта, не только во время конструирования.

51. Используйте иерархии исключений, желательно — производные от стандартной иерархии исключений C++ и вложенные в виде открытых классов в классы, генерирующие исключения. При перехвате исключений сначала обрабатываются специализированные, а затем базовые типы исключений. Если позднее появятся новые производные исключения, то существующий клиентский код все равно перехватит их через базовый тип.

52. Генерируйте исключения по значению и перехватывайте их по ссылке. Поручите управление памятью механизму обработки исключений. Если при выдаче исключений использовать указатели на объекты исключений, созданные в куче, то перехватчик должен знать о необходимости уничтожить объект, а это нежелательно. Перехват исключений по значению может привести к лишним операциям конструирования и уничтожения объектов, а повышающее приведение типа становится причиной расщепления (отсечения производных частей объектов исключений).

53. Не пишите собственные шаблоны классов без крайней необходимости. Сначала поищите подходящий шаблон в стандартной библиотеке C++, а затем в продуктах разработчиков, занимающихся созданием нестандартных инструментов. Научитесь пользоваться ими, это существенно повысит производительность вашего труда.

54. Работая над шаблоном, найдите код, который не зависит от типа, и поместите его в нешаблонный базовый класс, чтобы предотвратить излишнее разрастание программного кода. Используя наследование или композицию, вы сможете создавать шаблоны, весь код которых зависит от типа, а следовательно, абсолютно необходим для работы шаблона.

55. Не используйте функции из файла <cstdio>, такие как printf(). Вместо этого научитесь работать с потоками ввода-вывода; они безопасны по отношению к типам, расширяемы и обладают гораздо большими возможностями. Ваши труды будут постоянно вознаграждаться. Как правило, вместо библиотек C всегда следует использовать библиотеки C++.

56. Избегайте встроенных типов C. Они поддерживаются в C++ для обеспечения совместимости, но по надежности уступают классам C++, поэтому их применение затянет поиск ошибок.

57. Если вы используете встроенные типы для глобальных или автоматических переменных, отложите их определение до момента инициализации. Определяйте по одной переменной в строке одновременно с инициализацией. При определении указателей размещайте оператор * рядом с именем типа (если в строке определяется всего одна переменная, это абсолютно безопасно). Такой стиль лучше воспринимается при чтении программы.

58. Проследите за правильным выполнением инициализации во всех местах программы. Вся инициализация членов класса выполняется в списках инициализирующих значений конструктора, причем это относится и к встроенным типам (в псевдоконструкторах). При инициализации внутренних объектов списки инициализирующих значений конструктора часто оказываются более эффективными; без них вызывается конструктор по умолчанию, и для получения нужной инициализации вам еще придется вызывать другие функции класса (вероятно, operator=).

59. Не используйте форму MyType a=b; для определения объектов. Этот синтаксис часто вызывает путаницу, потому что вместо функции operator= он вызывает конструктор. Чтобы программа была более понятной, всегда четко выражайте свои намерения и используйте форму MyType a(b);. Результат получается тот же, но зато другим программистам будет проще разобраться в вашем коде.

60. Используйте явное приведение типа (см. главу 3). Операции приведения типа влияют на нормальную работу системы контроля типов и являются потенциальным источником ошибок. Поскольку явное приведение типа делит «универсальный» механизм приведения типа языка С на несколько разновидностей с понятными и однозначными именами, в процессе отладки и сопровождения кода любой программист легко найдет места, где с наибольшей вероятностью возникают логические ошибки.

61. Надежная работа всей программы возможна только при надежной работе всех ее компонентов. Используйте те механизмы, которые предоставляет в ваше распоряжение С++: ограничение доступа, исключения, константность, проверка типов и т. д. во всех создаваемых классах. Это позволит вам безопасно подняться на следующий уровень абстракции при построении системы.

62. Правильное использование ключевого слова const поможет компилятору обнаружить некоторые неочевидные и нетривиальные ошибки. Эта привычка требует внутренней дисциплины и логической последовательности, но в конечном счете она окупается.

63. В полной мере используйте средства проверки ошибок компилятора. Выполняйте все компиляции с полной выдачей предупреждений; исправьте программу, чтобы компилятор не выдавал предупреждений. Следите за тем, чтобы потенциальные ошибки обнаруживались на стадии компиляции, не переходя на стадию выполнения (например, не используйте переменные списки аргументов, подавляющие всю проверку типов). При отладке задействуйте макрос assert() и организуйте обработку исключений на стадии выполнения.

64. С ошибками времени компиляции справиться проще, чем с ошибками времени выполнения. Старайтесь обрабатывать ошибки как можно ближе к точке их возникновения. Если возможно, обработайте ошибку «на месте», вместо того чтобы генерировать исключение. Перехватывайте исключения в ближайшем обработчике, который располагает достаточной информацией. Сделайте все, что только можно сделать с исключением на текущем уровне; если

это не решит проблему, перезапустите исключение (за подробностями обращайтесь ко второму тому).

65. Если вы используете спецификации исключений (тема обработки исключений рассматривается во втором томе книги), определите собственный обработчик unexpected() при помощи функции set_unexpected(). Функция unexpected() должна регистрировать ошибку и перезапускать текущее исключение. Если существующая функция в результате переопределения начнет выдавать исключения, вы будете располагать полной информацией, что позволит изменить существующий код и обработать исключение.
66. Создайте пользовательскую функцию terminate(), которая регистрирует ошибку, ставшую причиной исключения, освобождает системные ресурсы и завершает работу программы.
67. Если в деструкторе вызываются какие-либо функции, они тоже могут генерировать исключения. Деструктор не может генерировать исключения (это приводит к вызову функции terminate(), что свидетельствует об ошибке программирования), поэтому любой деструктор, вызывающий функции, должен перехватывать и обрабатывать свои исключения.
68. Не используйте хитроумные схемы именования закрытых переменных (начальные символы подчеркивания, венгерская запись и т. д.), если в программе не используется множество заранее определенных глобальных значений. Видимость имен должна определяться на уровне классов и пространств имен.
69. Не забывайте о перегрузке. Варианты выполнения кода в функциях не должны ставиться в зависимость от значения аргумента; в таких случаях лучше определить две или более перегруженные версии функции.
70. Скрывайте указатели внутри контейнерных классов. Выводите их наружу только перед непосредственным выполнением операций с ними. Указатели всегда были одним из основных источников ошибок. При вызове оператора new постарайтесь немедленно сохранить полученный указатель в контейнере. Желательно, чтобы указатели «принадлежали» своему контейнеру и он отвечал за их уничтожение. Еще лучше, если указатель будет инкапсулироваться в классе; а если вы хотите, чтобы он выглядел как указатель, перегрузите функции operator-> и operator*. Наконец, если в программе приходится работать с «обычным» указателем, всегда инициализируйте его — желательно адресом объекта или, по крайней мере, нулем. После удаления обнулите указатель, чтобы предотвратить случайное повторное удаление.
71. Не перегружайте глобальные версии операторов new и delete; их перегрузка должна осуществляться только на уровне классов. Перегрузка глобальных версий распространяется на весь проект прикладного программиста, а такие вещи должны находиться под контролем создателей проекта. При перегрузке оператора new и delete для класса не предполагайте, что вам точно известен размер объекта; класс может использоваться в качестве базового. Применяйте переданный аргумент. Если перегруженные версии делают что-то особенное, подумайте, как это отразится на производных классах.

72. Избегайте расщепления объектов; повышающее приведение типа объекта по значению почти всегда бессмысленно. Чтобы предотвратить повышающее приведение типа по значению, включите в базовый класс чисто виртуальные функции.

73. Иногда задача решается простым агрегированием. Например, «система комфортности» на самолете включает несколько разнородных элементов: сиденье, кондиционер, видео и т. д., но все эти элементы должны существовать во множестве экземпляров. Может, имеет смысл сделать их закрытыми переменными и построить новый интерфейс? Нет, не имеет — в данном случае компоненты также являются частью открытого интерфейса, поэтому правильнее включить в класс открытые внутренние объекты. Безопасность этих объектов будет обеспечиваться наличием у них собственных закрытых реализаций. Помните, что простое агрегирование, хотя и применяется не так уж часто, иногда приносит ощутимую пользу.

Алфавитный указатель

Символы

!, оператор, 129
!=, оператор, 125
#define, 151, 187, 248
#endif, 187, 551
#ifndef, 187
#include, 74
#undef, 151
%, оператор, 124
&, оператор, 108, 126
&&, оператор, 125
&=, оператор, 126
(), оператор, 377
*, оператор, 124
+, оператор, 124
    для указателей, 148
++, оператор, 129
    для указателей, 148
-, оператор, 124
--, оператор, 129
->, оператор, 374
->*, перегрузка, 378
/, оператор, 124
<, оператор, 125
<<, оператор, 126
<<=, оператор, 126
<=, оператор, 125
=, оператор
    как закрытая функция, 391
    перегрузка, 370
==, оператор, 125
>, оператор, 125
>=, оператор, 125
>>, оператор, 126
>>=, оператор, 126
?:, тернарный оператор, 129
[], оператор, 88, 381, 509
^=, оператор, 126
|, оператор, 126
|=, оператор, 126
||, оператор, 125
~
    оператор, 126
    префикс деструктора, 216

A

abort(), 302
argc, 146
argv, 146
asctime(), 285
assert(), макрос, 153, 171
atexit(), 302
atof(), 146
atoi(), 146

atol(), 146
auto, ключевое слово, 118, 305

B

bad_alloc, 415
BASIC, язык, 61
break, ключевое слово, 99

C

C
  abort(), 302
  exit(), 302
C++
  GNU, компилятор, 63
  причины успеха, 58
  сильная типизация, 331
  совместимость, 180
  эффективность, 60
CALL, команда, 337
calloc(), 171, 403
catch, секции, 418
char, 82, 105
cin, 82
const_cast, 134
continue, ключевое слово, 99
cout, 78

D

delete, 129, 171, 405
  глобальная версия, 415
  для массивов, 412
  ключевое слово, 42
  нулевой операнд, 244
  перегрузка, 414
do-while, 98
dynamic_cast, 132, 496

E

else, 96
enum, 140
exit(), 294, 302
explicit, 392

F

false, 125
for, циклы, 88, 98
free(), 171, 403, 416
friend, 199, 406
  вложенные структуры, 201
  глобальные функции, 199
  функции классов, 199
fstream, 85

G

get(), 347
getline(), 85, 411
GNU C++, 63
goto, 101, 216, 219

I

if-else, 96
ifstream, 85, 443
insert(), 87
int, 105

J

Java, 20, 59, 65, 506

L

l-значение, 144, 257
limits.h, 105
long, 106
longjmp, 216

M

main(), 79
make, утилита, 157
malloc(), 171, 403
memset(), 202, 243, 410
mutable, 270

N

new, 129, 171
  глобальная версия, 415
  для массивов, 412, 419
  ключевое слово, 42
  массивы указателей, 409
  перегрузка, 414

O

ofstream, 85, 434
ostream, 244

P

Perl, 76
printf(), 416

private, 33, 198
protected, 33, 198, 203, 446
public, 33, 197
Python, 51, 65, 68, 472, 512

R

r-значение, 123, 509
realloc(), 171, 403
register, 305
reinterpret_cast, 132, 134
RETURN, 337

S

setf(), 343
setjmp(), 216
SGI, 87
Simula, язык, 30, 204
sizeof, оператор, 135
Smalltalk, 30, 472
static, 119, 299, 504
  const, 264
  инициализация, 318
  функции класса, 316
static_cast, 132, 497
STL, 87
string, 80, 191
struct, 137, 168, 196
  вложенные структуры, 189
  массивы, 143
  минимальный размер, 184
  размер, 183
system(), 83

T

this, 215, 272, 282, 316, 469
throw, 418
time_t, 285
toupper(), 279
true, 125
try, блок, 419
typedef, 137, 176, 305

U

UML, 32
unsigned, 106
using, 79, 308
  std, 188
  директива, 79
  объявление, 310

V

void*, 113, 154, 169
volatile, 123, 271
VPTR, 466, 484, 486
VTABLE, 466

W

while, циклы, 85, 97
width(), 343

А

абстрактные базовые
  классы, 472
абстрактные типы данных, 104, 182
агрегатная инициализация, 225
агрегаты, 225
агрегирование, 34
адрес, 107
  константы, 252
  уникальность, 184
  элемента, 108
алгоритмы, стандартная
  библиотека C++, 541
анонимное объединение, 239
аргументы
  командной строки, 146, 191
  конструктор, 215
  передача, 333
  переменные списки, 94
  по умолчанию, 232
  пустые списки, C и C++, 93
  ссылки, 332
ассемблер, 136, 337

Б

базовые классы
  интерфейс, 463
  неустойчивость, 208
байт, 109
библиотеки, 67, 75, 167
бинарные операторы, 129
  перегрузка, 362

В

векторы, 539
    push_back(), 88
    присваивание, 90, 93
виртуальная память, 404
виртуальные
  деструкторы, 487, 514, 515
виртуальные
  функции, 442, 455, 539
    dynamic_cast, 496
    virtual, 40, 462
    определения, 473
    эффективность, 471
внешнее связывание, 121
внутреннее связывание, 121, 249
возврат
    констант, 256
    оптимизация, 373
    по значению, 331
    семантика, 260
    тип значения, 435
восьмеричная система, 122
временные объекты, 258, 344, 372
    возврат, 373
встроенные типы, 104
    псевдоконструктор, 432
выделение памяти
    динамическое, 171, 373, 401
    эффективность, 414

Г

гарантированная
  инициализация, 229, 401
глобальные операторы, 383
глобальные переменные, 117
глобальные функции, 178

Д

данные
    инициализация констант, 250
    статические, 299
декремент, 104, 129
деструкторы, 216
    автоматический вызов, 216
    виртуальные, 487, 514
    порядок вызова, 433
деструкторы (*продолжение*)
    чисто виртуальные, 488
    явный вызов, 423
директивы
    using, 79, 308
    препроцессора, 69
доступ
    контроль, 196
    спецификаторы, 33
    строение объекта, 203
    функции, 281

Е

единица трансляции, 174, 318

З

заголовочные
  файлы, 74, 95, 104, 178, 234
    повторное включение, 186
    порядок включения, 551
    шаблоны, 510
закрытое наследование, 445

И

иерархии, однокоренные, 491
имена
    конфликты, 175
    перегрузка, 231
    украшение, 176
индексирование, 88, 143
инициализация, 170, 264
    гарантии, 229, 401
    конструкторы, 214
инкапсуляция, 182, 204
инкремент, 104, 129
    перегрузка, 361
интерпретаторы, 68
интерфейс, 184
    базового класса, 463
    общий, 473
    объекта, 22
    отделение от реализации, 33
итераторы, 374, 511
    вложенные классы, 376
    контейнеры, 503

К

классы, 31, 204
  абстрактные базовые, 472
  вложенные, 315
  итераторы, 374
  локальные, 315
  манипуляторы, 208
  объявление, 209
  определение интерфейса, 56
  экземпляры, 30
коллекции, 374
конечный итератор, 528, 530
константы, 121, 248
  mutable, 270
  агрегаты, 251
  адрес, 252
  внешние, 252
  финт с перечислением, 266
конструкторы, 214, 402, 487
  аргументы, 215
  возвращаемое значение, 215
  закрытые, 519
  оператор присваивания, 382
  подставляемые, 281
  псевдоконструктор, 411
  список инициализирующих значений, 430
копирующий конструктор, 318
  закрытый, 346
  по умолчанию, 344
куча, 42, 171
  выделение памяти, 401
  создание объектов, 401

Л

логарифм, 343

М

макросы
  аргументы, 277
  препроцессор, 128, 149, 276
массивы, 142
методология, 44

Н

наследование, 34, 427, 429
  VTABLE, 474
наследование (*продолжение*)
  диаграммы, 41
  закрытое, 445
  защищенное, 447
  и композиция, 442
  открытое, 447
неявное приведение типа, 122

О

обучение персонала, 62
объединения
  анонимные, 239
  экономия памяти, 141
объекты, 29, 70
  адрес, 200
  временные, 258
  динамическое создание, 42
  локальные статические, 302
  определение, 181
  передача по значению, 339
  создание в куче, 401
  уникальность адресов, 184
объявления, 71
  virtual, 462
  анализ, 155
  и определения, 184
  опережающие, 120
  переменных, синтаксис, 73
  синтаксис, 72
ООП, 204
  Simula, 30
  анализ и проектирование, 44
  основные характеристики, 30
операторы
  глобальные, 383
  перегрузка, 356
  приоритет, 103
определения, 72
  и объявления, 184
  классов, 209
  массивов, 251
  функций, 72
оптимизация
  возвращаемого значения, 373
  подстановки, 292
отладка, 69
отложенная инициализация, 514

П

память, 109
выделение, эффективность, 414
динамическое
выделение, 171, 402
затраты на управление, 406
постоянная, 270
управление, 241
перегрузка, 81
глобальных операторов, 383
конструктор, 231
оператора
++, 361
--, 361
[], 381
присваивание, 382
функций, 231
передача
по значению, 110, 339
по ссылке, 112
перекомпиляция программ С, 180
переменные
автоматические, 42, 118, 121
глобальные, 117
регистровые, 118
синтаксис объявления, 73
счетчики циклов, 218
точка определения, 218
переопределение, 462
при наследовании, 430
планирование, 49
повышающее приведение
типа, 41, 449, 460, 536
позднее связывание, 40, 462
полиморфизм, 40, 436, 459, 506
пользовательские типы
данных, 104, 182
постусловия, 552
постфиксный декремент, 104
постфиксный инкремент, 104
предусловия, 552
преобразование типа, 41
препроцессор, 69, 74, 121
макросы, 128, 149, 276
префиксный инкремент, 104
приведение типа, 108, 131, 208
прикладной программист, 32, 196
принадлежность, 438, 519
приоритет операторов, 103
присваивание, 124, 225
запрет, 391
оператор, 370
проблема неустойчивости базового
класса, 208
программирование
XP, 56
масштабное, 61
мультипарадигменное, 30
ООП, 28
программист прикладной, 32, 196
проектирование, 44
пространство
задачи, 29
имен, 59, 78, 305, 551
NDEBUG, 151
std, 79
директива using, 308
использование, 308
объявление using, 308
решения, 29
процесс, 271

Р

раздельная компиляция, 156
размер слова, 106
разыменование, 150
раннее связывание, 40, 466, 486
реализация, 32, 184
отделение от интерфейса, 33, 282
регистровые переменные, 305
реентерабельность, 337

С

С, 217
совместимость, 65
самоприсваивание, 370, 384
семантика возврата, 260
сигнатура, 436
синтаксис
объявления
переменных, 73
функций, 72
перегрузки операторов, 357

системные спецификации, 47
слабая типизация, 512
сообщения, отправка, 31, 182
составные типы, 88
спецификаторы доступа, 33
ссылки, 331, 334
  передача аргументов, 261
  повышение, 460
  эффективность, 335
стандартный ввод, 82
стандартный вывод, 78
стек, 42, 189, 402
Страуструп, Бьярн, 508, 545
сценарии, 48

Т

тестирование
  XP, 56
  автоматизация, 57
типы
  абстрактные, 104, 182
  базовые, 35
  встроенные, 104
  объединения, 141
  перечисления, 140
  пользовательские, 121
  приведение, 108
  проверка, 70, 72, 121
  производные, 35
трансляции, единица, 174

У

уборка мусора, 43, 414
указатели, 111, 121, 208, 331
  const, 134
  void, 331
  void*, 407
  на члены классов, 347
  общие сведения, 109
унарные операторы, перегрузка, 358
универсальные алгоритмы, 541

Ф

Фибоначчи, 528
фрагментация кучи, 172, 415
функции, 72
  адреса, 153, 290
  аргументы, 110
  базового класса, 429
  виртуальные, 462
  возвращаемое значение, 94
  глобальные, 178
  доступа, 281
  дружественные, 199
  затраты на вызов, 276
  конструкторы, 484
  массивы указателей, 156
  объявление, 95, 186
  перегрузка, 231
  переопределение, 37
  подставляемые, 276
  прототипизация, 92
  связывание, 461
  создание, 92
  ссылки, 332
  указатели, 154

Ц

циклический сдвиг, 128

Ч

«чеширский кот», 208

Ш

шаблоны, 503, 508
  STL, 87
  слабая типизация, 512
  список аргументов, 512
  функций, 540

Э

эволюция программы, 54
экземпляр класса, 30
экстремальное программирование, 56
элегантность, 54
эталоны, 55
эффективность, 276
  конструкторы, 484
  подставляемые функции, 282

Брюс Эккель

Философия C++. Введение в стандартный C++

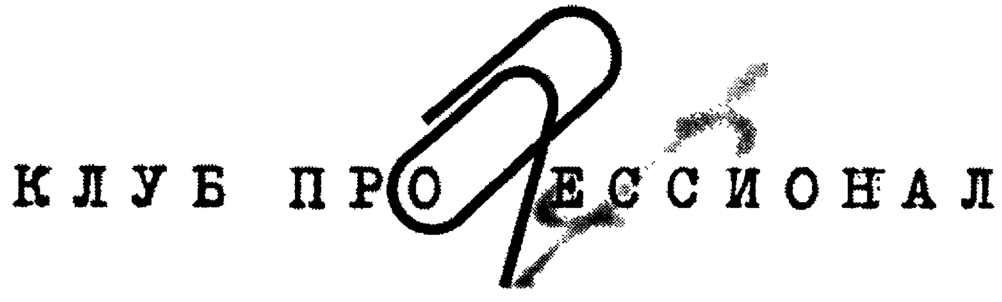
2-е издание

Перевел с английского Е. Матвеев

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*А. Крузенштерн*
Литературный редактор	*А. Жданов*
Художник	*Н. Биржаков*
Иллюстрации	*В. Шендерова*
Корректоры	*Н. Викторова, Н. Лукина*
Верстка	*А. Келле-Пелле*

Лицензия ИД № 05784 от 07.09.01.

Подписано в печать 20.01.04. Формат 70×100/16. Усл. п. л. 46,44.
Тираж 3500 экз. Заказ № 1699.
ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в.
Налоговая льгота — общероссийский классификатор продукции
ОК 005-93, том 2; 953005 — литература учебная.
Отпечатано с готовых диапозитивов в ФГУП «Печатный двор»
им. А. М. Горького Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.



В 1997 году по инициативе генерального директора **Издательского дома «Питер»** Валерия Степанова и при поддержке деловых кругов города в Санкт-Петербурге был основан **«Книжный клуб Профессионал»**. Он собрал под флагом клуба профессионалов своего дела, которых объединяет постоянная тяга к знаниям и любовь к книгам. Членами клуба являются лучшие студенты и известные практики из разных сфер деятельности, которые хотят стать или уже стали профессионалами в той или иной области.

Как и все развивающиеся проекты, с течением времени книжный клуб вырос в **«Клуб Профессионал»**. Идею клуба сегодня формируют три основные «клубные» функции:

- неформальное общение и совместный досуг интересных людей;
- участие в подготовке специалистов высокого класса (семинары, пакеты книг по специальной литературе);
- формирование и высказывание мнений современного профессионала (при встречах и на страницах журнала).

КАК ВСТУПИТЬ В КЛУБ?

Для вступления в **«Клуб Профессионал»** вам необходимо:

- ознакомиться с правилами вступления в **«Клуб Профессионал»** на страницах журнала или на сайте **www.piter.com**;
- выразить свое желание вступить в **«Клуб Профессионал»** по электронной почте **postbook@piter.com** или по тел. **(812) 103-73-74**;
- заказать книги на сумму не менее 500 рублей в течение любого времени или приобрести комплект **«Библиотека профессионала»**.

«БИБЛИОТЕКА ПРОФЕССИОНАЛА»

Мы предлагаем вам получить все необходимые знания, подписавшись на **«Библиотеку профессионала»**. Она для тех, кто экономит не только время, но и деньги. Покупая комплект – книжную полку **«Библиотека профессионала»**, вы получаете:

- **скидку 15%** от розничной цены издания, без учета почтовых расходов;
- при покупке двух или более комплектов – дополнительную **скидку 3%**;
- членство в **«Клубе Профессионал»**;
- подарок – журнал **«Клуб Профессионал»**.

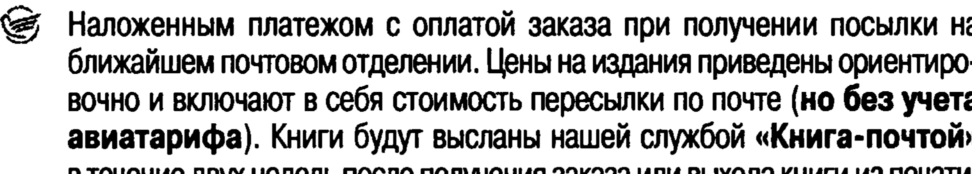
Закажите бесплатный журнал «Клуб Профессионал».

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- по телефону: **(812) 103-73-74;**
- по электронному адресу: **postbook@piter.com;**
- на нашем сервере: **www.piter.com;**
- по почте: **197198, Санкт-Петербург, а/я 619 ЗАО «Питер Пост».**

ВЫ МОЖЕТЕ ВЫБРАТЬ ОДИН ИЗ ДВУХ СПОСОБОВ ДОСТАВКИ И ОПЛАТЫ ИЗДАНИЙ:



Наложенным платежом с оплатой заказа при получении посылки на ближайшем почтовом отделении. Цены на издания приведены ориентировочно и включают в себя стоимость пересылки по почте (**но без учета авиатарифа**). Книги будут высланы нашей службой **«Книга-почтой»** в течение двух недель после получения заказа или выхода книги из печати.

Оплата наличными при курьерской доставке (**для жителей Москвы и Санкт-Петербурга**). Курьер доставит заказ по указанному адресу в удобное для вас время в течение трех дней.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, код, количество заказываемых экземпляров.

Вы можете заказать бесплатный журнал «Клуб Профессионал».

СПЕЦИАЛИСТАМ КНИЖНОГО БИЗНЕСА!

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»

предлагают эксклюзивный ассортимент компьютерной, медицинской, психологической, экономической и популярной литературы

РОССИЯ

Москва м. «Калужская», ул. Бутлерова, д. 17б, офис 207, 240; тел./факс (095) 777-54-67; e-mail: sales@piter.msk.ru

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а; тел. (812) 103-73-73, факс (812) 103-73-83; e-mail: sales@piter.com

Воронеж ул. 25 января, д. 4; тел. (0732) 27-18-86; e-mail: piter-vrn@vmail.ru; piterv@comch.ru

Екатеринбург ул. 8 Марта, д. 267б; тел./факс (3432) 25-39-94; e-mail: piter-ural@r66.ru

Нижний Новгород ул. Премудрова, д. 31а; тел. (8312) 58-50-15, 58-50-25; e-mail: piter@infonet.nnov.ru

Новосибирск ул. Немировича-Данченко, д. 104, офис 502; тел/факс (3832) 54-13-09, (3832) 47-92-93; e-mail: piter-sib@risp.ru

Ростов-на-Дону ул. Калитвинская, д. 17в; тел. (8632) 95-36-31, (8632) 95-36-32; e-mail: jupiter@rost.ru

Самара ул. Новосадовая, д. 4; тел. (8462)37-06-07; e-mail: piter-volga@sama.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10–11, т. (057) 712-27-05, 712-40-88; e-mail: piter@tender.kharkov.ua

Киев пр. Красных Казаков, д. 6, корп. 1; тел./факс (044) 490-35-68, 490-35-69; e-mail: office@piter-press.kiev.ua

БЕЛАРУСЬ

Минск ул. Бобруйская д., 21, офис 3; тел./факс (37517) 226-19-53; e-mail: piter@mail.by

МОЛДОВА

Кишинев «Ауратип-Питер»; ул. Митрополит Варлаам, 65, офис 345; тел. (3732) 22-69-52, факс (3732) 27-24-82; e-mail: lili@auratip.mldnet.com

---

Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 103-73-73.**
E-mail: grigorjan@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов.
Обращайтесь по телефонам: **Санкт-Петербург – (812) 327-13-11, Москва – (095) 777-54-67.**

Заказ книг для вузов и библиотек: (812) 103-73-73.
Специальное предложение – e-mail: kozin@piter.com

**УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ
У НАШИХ РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.**

Башкортостан
Уфа, «Азия», ул. Зенцова, д. 70 (оптовая продажа),
маг. «Оазис», ул. Чернышевского, д. 88,
тел./факс (3472) 50-39-00.
E-mail: asiaufa@ufanet.ru

Дальний Восток
Владивосток, «Приморский торговый дом книги»,
тел./факс (4232) 23-82-12.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Мирс»,
тел. (4212) 30-54-47, факс 22-73-30.
E-mail: sale_book@bookmirs.khv.ru

Хабаровск, «Книжный мир»,
тел. (4212) 32-85-51, факс 32-82-50.
E-mail: postmaster@worldbooks.kht.ru

Европейские регионы России
Архангельск, «Дом книги»,
тел. (8182) 65-41-34, факс 65-41-34.
E-mail: book@atnet.ru

Калининград, «Вестер»,
тел./факс (0112) 21-56-28, 21-62-07.
E-mail: nshibkova@vester.ru
http://www.vester.ru

Северный Кавказ
Ессентуки, «Россы», ул. Октябрьская, 424,
тел./факс (87934) 6-93-09.
E-mail: rossy@kmw.ru

Сибирь
Иркутск, «ПродаЛитЪ»,
тел. (3952) 59-13-70, факс 51-30-70.
E-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Антей-книга»,
тел./факс (3952) 33-42-47.
E-mail: antey@irk.ru

Красноярск, «Книжный мир»,
тел./факс (3912) 27-39-71.
E-mail: book-world@public.krasnet.ru

Нижневартовск, «Дом книги»,
тел. (3466) 23-27-14, факс 23-59-50.
E-mail: book@nvartovsk.wsnet.ru

Новосибирск, «Топ-книга»,
тел. (3832) 36-10-26, факс 36-10-27.
E-mail: office@top-kniga.ru
http://www.top-kniga.ru

Тюмень, «Друг»,
тел./факс (3452) 21-34-82.
E-mail: drug@tyumen.ru

Тюмень, «Фолиант»,
тел. (3452) 27-36-06, факс 27-36-11.
E-mail: foliant@tyumen.ru

Челябинск, ТД «Эврика», ул. Барбюса, д. 61,
тел./факс (3512) 52-49-23.
E-mail:evrika@chel.surnet.ru

Татарстан
Казань, «Таис»,
тел. (8432) 72-34-55, факс 72-27-82.
E-mail: tais@bancorp.ru

Урал
Екатеринбург, магазин № 14,
ул. Челюскинцев, д. 23,
тел./факс (3432) 53-24-90.
E-mail: gvardia@mail.ur.ru

Екатеринбург, «Валео-книга»,
ул. Ключевская, д. 5,
тел./факс (3432) 42-56-00.
E-mail: valeo@etel.ru

Философия C++

Введение в стандартный C++

2-е ИЗДАНИЕ

«Это — единственная книга, которую вам обязательно следует иметь, если вы всерьез занимаетесь разработкой на C++.»

Ричард Хейл Шоу,

выпускающий редактор PC Magazine

«В этой книге я постараюсь сформировать у читателя такую модель мышления, которая бы давала глубокое понимание языка вплоть до самых его основ. Встретив непонятную ситуацию, вы используете эту модель и получите ответ.
Я хочу поделиться с читателем теми соображениями, которые изменили мой стиль мышления и помогли мне "думать на C++".»

Б. Эккель

В ближайшем будущем выйдет вторая часть — «Философия C++: практическое программирование». В ней вы найдете описание целого ряда новых библиотек, а также таких нетривиальных тем, как множественное наследование, обработка исключений, построение и отладка стабильных систем.